Dynamic Web Application Analysis for Cross Site Scripting Detection

Björn Engelmann

August 2007

Dynamic Web Application Analysis for Cross Site Scripting Detection

Björn Engelmann

Diploma Thesis

Department of Informatics
Faculty of Mathematics, Informatics and Natural Sciences
University of Hamburg

submitted by

Björn Engelmann

supervised by

Prof. Dr. Joachim Posegga

co-supervised by

Prof. Dr. Norbert Ritter

Hamburg, August 2007

Abstract

Though cross site scripting (XSS) is essentially a server-side problem, in most cases users are the one who suffer. Additionally, most Anti-XSS measures developed so far are requiring either a major customization effort or modifications in the Web Application. This thesis presents a general XSS detector able to automatically derive all required Web Application specific knowledge. Data-mining techniques are employed to analyse Web Applications in a script-focused way, which only necessitates access to unencrypted HTTP traffic. Wherever this is given, the system thus can be used as a straightforwardly deployable, Anomaly-based Intrusion Detection Sensor. This can help to find XSS vulnerabilities in very different application environments, and in future, the detector may even be implemented in a browser to form a pure client-side XSS protection.

Acknowledgements

Many people have contributed to the development of this diploma thesis. First, my special thanks go to my thesis supervisor Prof. Dr. Joachim Posegga, head of the "Security in distributed systems" group at the University of Hamburg for his substantial support and constructive criticism, as well as many inspiring lectures throughout my graduate studies. His taking over the primary thesis supervision allowed me to conduct this research in the first place.

Secondly, I owe many thanks to Martin Johns, Ph. D. student at the University of Hamburg. Without his support, encouragement and guidance through the whole process, I would never have been able to accomplish this work.

I also would like to express thanks to Prof. Dr. Norbert Ritter, head of the "Database and Information Systems" group at the University of Hamburg, the co-supervisor of this thesis, for his support throughout the preparation of this diploma thesis as well as my graduate studies.

Further, my thanks and appreciation to my girlfriend, Luyue Shen, as well as my parents for their support throughout the whole process.

And finally, I am also thankful to Prof. Esko Ukkonen, Kenichi Morita, Noritaka Nishihara, Yasunori Yamamoto, Zhiguo Zhang, the Mozilla Project and all others who made the fruits of their work available for anyone and therefore crucially contributed to an environment necessary to create a thesis like this.

Contents

1. Introduction

Pages of dynamic Web Applications often contain user-supplied parts. When insufficiently filtered, malicious scripts can be injected along with these parts. Assuming, that these are executed in a user's browser, it is possible to exploit the trust relationship between user and Webserver by, for instance, compromising authentication credentials. This type of attack is called Cross Site Scripting (XSS) and, although it originates in a failure in the Web Application, clearly jeopardizes it's users.

It was discovered in 2001, and since then has become by far the most common Web Application vulnerability [7]. A lot of research was conducted for ways to develop more secure Web Applications and fix existing ones. All approaches, however, require modification of the application or other complex and time-consuming efforts. Even though new applications nowadays can use modern frameworks with integrated support for such methods, a huge number of old applications on the Internet still has to be ported. Probably due to the situation of misplaced incentives, many application providers are opting only to fix vulnerabilities they are notified about, instead of investing in the security of their sites. At present, the XSS disclosure rate is still on the rise.

In this thesis, two conceptually different approaches for detecting XSS attacks are proposed. All necessary information is derived using data mining techniques on mere HTTP traffic, thus not requiring any Web Application modifications and allowing a straightforward deployment. Additionally, both methods can be applied outside the provider's sphere of influence, as long as access to unencrypted HTTP traffic is given.

While this thesis focuses on the application as Intrusion Detection Sensor, the methods discussed may constitute a first step towards anomaly-based, pure client-side XSS protection.

Excluding Introduction, the thesis is composed of 7 Chapters. Chapter 2 starts by providing a collection of background information, as well as formal foundations necessary to follow later argumentations. Afterwards, Chapter 3 outlines the gathering of a large body of Webpages, the analysis of which led to the development of our concepts and which was crucial for evaluation. Being the principal part of this work, Chapter 4 presents both concepts, discusses them in detail and gives two possible application scenarios. A prototypic implementation is presented in Chapter 5 and used for an empirical evaluation, which is, along with it's results, discussed in Chapter 6. Chapter 7 lists a selection of related work and Chapter 8 concludes with a number of ideas for further improvement.

2. Fundamentals

The Internet is the largest equivalence class in the reflexive, transitive, symmetric closure of the relationship "can be reached by an IP packet from".

–Seth Breidbart

At the outset, this Chapter will provide background information necessary for understanding the remaining thesis. First, Section 2.1 will give a technical overview of the World Wide Web, the environment in which Cross Site Scripting attacks (described in Section 2.2) take place. They are the problem this work is trying to mitigate. After several flavors and combinations thereof have been exemplified, Section 2.3 will briefly introduce the Mozilla project, as it is fundamental for the prototypical implementation. Section 2.4 provides a basic understanding of Intrusion Detection, introducing technical terms and differentiating it's two types. As one of our concepts describes an Anomaly-based IDS, this vocabulary is used throughout the thesis and the reader should be able to familiarize himself with it. Finally, Section 2.5 defines a body of formal notions mainly used in Chapter 3 and 4.

Even though this Chapter is intended to provide a foundation for the remaining thesis, some basic notions as complexity theory, deterministic finite automata and hashtables are assumed as known. Farther afield are mathematical concepts as equivalence relations or function composition. Those can be learned from introductory textbooks on theoretical computer science, data structures and discrete mathematics.

The well-versed reader may of course safely skip this Chapter entirely or in parts.

2.1. World Wide Web

The World Wide Web (WWW) is a hypertext-system invented in 1989 by Tim Berners-Lee at CERN. It is the most commonly used application on today's Internet. A large number of so-called Webservers are offering their clients (called Webbrowsers) documents of varying types and contents. Every document is assigned one or more unique addresses (called URLs), containing, amongst others, the name of a server, from which the document can be queried. Most of this documents are Webpages written in HTML, which allows reference to other documents using unidirectional hyperlinks. In this way, a world wide network of interconnected documents is created, hence the name WWW. A number of Webpages, which are accessible under the same domain and forming a coherent set of information, are referred to as 'Website'.

The WWW (or the 'Web', as it is often called) is the technical foundation, on which both Cross Site Scripting attacks (discussed in Section 2.2) and the concepts in Chapter 4 are based. The information in this Section is crucial for understanding how vulnerable Webservers are jeopardizing their users.

The following Subsections will go deeper into the more technical details of the WWW. Subsection 2.1.1 introduces HTTP, the Web's basic protocol and Subsection 2.1.2 gives an overview of HTML, the language Webpage are written in. After dynamic Web Applications,

a special type of Website interesting in the context of Cross Site Scripting are discussed in Subsection 2.1.3, Subsection 2.1.4 will give more details on how the Session Mechanisms are implemented. While Subsection 2.1.5 gives a general introduction of client-side scripting, Subsection 2.1.6 focuses on JavaScript, the language used for the majority of Cross Site Scripting attacks.

2.1.1. HTTP

The HyperText Transfer Protocol (HTTP) is used by browsers to query documents from the servers and therefore constitutes the very foundation of the WWW. It is specifying a simple request-response-scheme, offering a variety of methods for adaption to different purposes. While the GET-method for querying documents certainly is the most commonly known and used one, there is, for instance, also a POST-method for transmitting data to the server.

Independent of the method used, however, the client is sending a URL to the server that in turn is answering with a 3-digit response-code and a document. Both request and response can additionally contain any number of name-value pairs in their header, used for implementing cookies as an example (see Subsection 2.1.4). If the request could be processed successfully, the server answers with the response-code 100 - 'success'. Otherwise, the code is used to give information about what type of error occurred. 404 means 'document not found' and 500 stands for 'authentication failed'. Additionally, it is possible to forward the browser to a different URL with the response-code 301 - 'moved permanently'. It should be stressed though, that every response contains a document, independent of the response-code. In case of errors, this is usually used to explain the error in human-readable form.

HTTP uses TCP-connections to transmit its data. However, unlike TCP, it is completely stateless. The connection is always closed after transmitting one or more request-response-pairs. Even later, when the introduction of HTTPS demanded the reuse of SSL-connections for efficiency, the statelessness was preserved for compatibility reasons.

2.1.2. HTML

The HyperText Markup Language (HTML) is the language used by Webpages in the WWW. It has been developed by the World Wide Web Consortium (W3C) until version 4.01 and is rendered by all popular browsers, more or less conforming to the standard. HTML documents are consisting of text, so-called tags and their attributes. Like XML, they are exhibiting a tree-like structure and, besides pure text, can contain markup elements (bold, italic, etc.), hyperlinks, as well as meta-information (language, author). Other documents, like images, animations, music or other Webpages can be included by their URLs. When a browser is displaying such a Webpage, it has to send out a number of HTTP-requests probably to a number of different servers.

2.1.3. Dynamic Websites

In contrast to static Webpages, dynamic ones are able to change over time. After receiving the request, their contents is generated by a program and therefore can change depending on a number of factors. It thus can happen, that a browser receives two different documents when querying the same URL twice. For this reason, dynamic Webpages are marked with special tags refraining the browser from caching them.

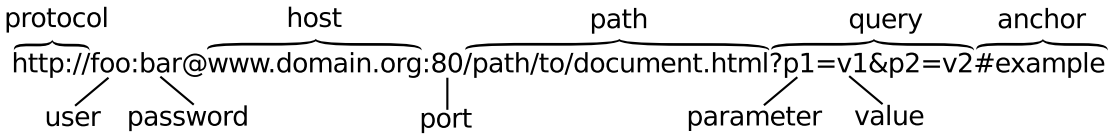


Figure 2.1.: structure of a URL according to [10]

Another possibility is to pass parameters to the generating program using the so-called URL-parameters or the POST-method. The structure of such an URL is shown in Figure 2.1. Of course, URL-parameters are encoded in a special way (URL-encoding, also known as Percent-encoding), to ensure parsability. Basically, a generating program may access any part of a request, and thus all parts of its HTTP header can also be considered parameters.

Generating programs are usually written in scripting languages, whose interpreter as able to cooperate with the Webserver. The most popular example certainly is PHP, a language specially developed for this purpose. But normal languages like Java, Perl, Python or Ruby can be used this way. Although theoretically, any data source can influence the contents of a dynamic Webpage, databases or other persistent data storages are usually used. Applications like forums, weblogs, wikis, etc. which republish inputs of their users can be created this way. Such a dynamic Website is called 'Web Application'.

2.1.4. Authentication & Sessions

For allowing access restrictions, a feature called HTTP-Authentication was already introduced in HTTP 0.9: If a browser requests a restricted document, the server responds with response-code 401 - 'unauthorised' and the information as to which 'realm' the document belongs to. Every document within this realm can only be accessed with the appropriate credentials (which the browser requests from his user) being included in the requests header.

For allowing restricted realms to be used conveniently, sessions were implemented. Once the credentials have been requested from the user, the browser stores and automatically includes them into every response for a document of the restricted realm. After a first successful authentication (log in) the user therefore does not need to bother about the restriction any more. Mechanisms working like that are referred as 'implicit authentication'.

Most modern Web Application are (mis)using a different mechanism to authenticate their users and provide authenticated sessions, though: so-called HTTP-cookies [13]. A cookie is a small chunk of data (max. 4096 bytes according to the HTTP1.1 specification [15]), included in the header of some HTTP-response ('Set-Cookie'-field) and assigned a domain as well as an expiration date. Cookies are - if permitted by the user - stored in the browser and, very much like the authentication credentials, automatically included in each request ('Cookie'-field) furthermore sent by the browser within the validity period to a server of the same domain (or a subdomain).

Many Web Applications are using cookies in the following way to implement sessions: First, the user is logging into the Web Application by for example entering his credentials into a form and sending it to the server using the POST-method. The server in turn checks the credentials for validity and if successful, includes a cookie containing a large random number into his response. This number is called sessionID and also stored in the servers database. If the Web Applications domain is specified in the cookie, the browser will include the sessionID

into each further request and the server is able to associate them with his credentials thus authenticating him.

2.1.5. Active Content

For allowing Web Applications to execute parts of their logic on the client-side, a number of active content technologies were introduced over time. In classical client-server applications, client-side logic has already been used for several years to improve the application's responsiveness. On the Web, JavaScript, Java-applets, Flash or similar technologies are also mainly used to shorten response times and interact more closely with the user. Lately, however, an increasing number of so-called 'Asynchronous JavaScript And XML (AJAX)'-applications are appearing, which are moving their entire application logic to the client-side and use servers as mere data storages or in the form of Web Services.

Common for all types of active content is that it has to be integrated into HTML-code somehow. Binary formats like Macromedia Flash or the bytecode of Java-applets are usually stored as separate documents on the server and integrated by giving their URL. The browser then queries them along with the Webpage and passes them to the appropriate plugin for execution.

Human-readable scripting languages like JavaScript of VBscript can mostly be integrated into HTML documents in several ways. As JavaScript-interpreters are often integrated into the browsers (SpiderMonkey into Firefox and JScript into Internet Explorer), a very close collaboration is possible. In recent years, browser vendors were steadily extending methods both for integration and interaction between browsers and scripts.

The next Subsection will give details for each method.

2.1.6. JavaScript

The JavaScript language was invented for the Netscape Navigator by Brendan Eich in 1995. Since then, it has been standardized as ECMAScript and is supported by all popular browsers. JavaScript is the first - and nowadays the most popular - client-side language of the WWW. It can be integrated into HTML-pages in a variety of ways[1]:

First, it is obviously possible to include external JavaScript. An empty script-tag is added to the document and the URL is specified in it's 'src'-attribute. Like in the case of Flash or Java-applets, the document from that URL is queried and it's content executed.

The second possibility is the so-called inlining. The code is enclosed by script-tags and directly inserted into the document. In both ways, the browser stops the parsing process when encountering the first script-tag, executes the code and inserts everything that was 'written' using the command `document.write()` directly after the closing script-tag. After the script's termination, parsing continues right there with what has been inserted.

Additionally, most HTML-tags have been given so-called 'events'. JavaScript is able to handle these when a so-called event-handler is specified. If a tag is given an attribute with the event's name ('onclick', 'onmouseover', 'onload', etc.) and the code is specified as it's value, it will be executed every time the event occurs (the element is clicked, touched by the mouse cursor, finishes loading, etc).

[1] The listing only includes those methods offered by Mozilla Firefox. Other browsers may offer additional ones.

Since JavaScript is able to read and write cookies, and HTML-documents from different servers can be intermixed and embedded in various ways, Netscape, along with JavaScript, introduced a security concept known as the 'same-origin-policy': Every document is assigned a so-called 'site', containing the domain name, protocol and port of it's URL. All documents corresponding in these three properties are belonging to the same site. Next, every object a document contains is assigned the document's site as an 'origin'. If some object (usually a script) wants to access another (for example a cookie), this is only granted if both have the same origin. It should be pointed out here that the origin of an external script (`http://www.scripts.com/script.js`) included into some page (`https://www.example.com/index.html`) is the page's site (`https://www.example.com:443`).

In this Section all technical details of the WWW necessary to understand Cross Site Scripting attacks were introduced. The next Section will focus on this attack and give a detailed overview about it's criticality.

2.2. Cross Site Scripting

Cross Site Scripting (abbreviated XSS in order to avoid confusion with Cascading Style Sheets) refers to the injection of alien, executable code into a dynamic Webpage. This of course requires the respective generating program to be vulnerable in a certain way. One can distinguish different types of this security vulnerability, but in any case arbitrary scripts can be injected, which are interpreted as part of the Web Application by the browser of any user visiting the page, and according to the same origin policy are granted access to any of its components. Cross Site Scripting thus is completely evading this security policy, which is exploitable in a number of different ways. Subsection 2.2.1 will detail the different attack types.

According to several Statistics [7, 8], XSS is the most common vulnerability on today's Web. OWASP is calling it 'the most prevalent and pernicious Web Application security issue' and ranks it Top 1 in their 2007 Edition of the Top 10 Web Application security issues [5]. Moreover, in recent years an alarming trend of XSS vulnerabilities found in infrastructure like Webservers [43], Web Application Frameworks [44] or scripting languages [45] could be observed.

Every XSS attack is consisting of two components: A method for injecting the script into the Webpage (called 'attack vector') and the script itself (called 'payload'). Which payloads are to be considered malicious is very controversial, which of course leads to differing definitions of the XSS term itself. Some authors [6] are calling the injection of mere HTML-tags into a Website XSS, even though it is not possible to add any active content. This standpoint may be justifiable in respect to Cross Site Request Forgery (XSRF), another class of Web Application attacks, but is only causing confusion in the context of XSS, as many Web Applications (forums, wikis) are intentionally allowing their users the controlled insertion of such tags. As this thesis is solely concerning itself with XSS, discussing non-executable payloads is out of scope.

Subsection 2.2.1 lists a number of attack vectors and Subsection 2.2.2 lists common payload types. Even though most XSS attacks can be classified as a simple combination of these two, some examples at the end of Subsection 2.2.1 demonstrate how this simple vectors can be combined to more sophisticated attacks.

2.2.1. Attack Vectors

Cross Site Scripting attack vectors commonly are categorized into 3 classes. Their names are not standardized by any means and the type-numbering is somewhat arbitrary, but consistently used in the community. After discussing the classes, we will exemplify some combination methods of this basic vectors.

Reflected XSS (Type1)

If a generating program injects the content of a parameter into the document without proper filtering, thus allowing a script embedded in this parameter to be included into the page, we are calling this a 'reflected XSS vulnerability'. The Webserver is so to speak 'reflecting' the script back to the browser. This type of vulnerability is extremely common and still regarded as harmless by many Web developers. But, since the WWW is a hyperlinked environment, the user is not at all empowered to control which URLs his browser is querying. A single click onto a crafted link is enough for executing a malicious script like this.

A typical scenario: The operator of some forum (`http://www.forum.com`) configured his Webserver to display a nice description along with the 404 - 'page not found' error page. Of course this description also includes the name of the document that could not be found. Incautiously, he forgot to properly filter the URL. Now a user of that forum (lets call him Mel) adds a new thread, telling about a great Website he recently discovered and also providing a link. The link, however, points to `http://www.forum.com/<scriptsrc='hacker.org/xss.js'></script>`. If another user clicks the link, his browser will send a request to the `forum.com`-Webserver querying for some page named `<scriptsrc='hacker.org/xss.js'></script>`. As the server does not know about any document with that name, it answers with it's nice error page including the name. The user's browser in turn will obviously execute it as an external script from `hacker.org`, which coincidentally is controlled by Mel. Any JavaScript code Mel uploads to `hacker.org` will thus be executed with it's origin set to `http://www.forum.com:80`, allowing Mel for instance to read the forum's cookies and send their contents to a Webserver of his choosing.

Stored XSS (Type2)

Subsection 2.1.3 already outlined that most dynamic sites are using some kind of database to store user entries. At least some of them will eventually resurface as part of other pages, within that respective Web Application. Insufficient input filtering can in this case lead to a script being stored in the database. Since most developers are regarding the contents of their database as trustworthy, it is very likely that the output-filtering is insufficient, too. A vulnerability like this, allows an adversary to persistently inject a payload, which furthermore will be executed in any browser visiting the page.

As an example, one can simply imagine that the forum from the reflected XSS example would allow its users to insert arbitrary JavaScript code into their postings. In this case, Mel would not have to forge a link but could include the payload directly into his posting. The attack thus would not require clicking the link any more, but viewing the post would be sufficient.

DOM-based XSS (Type0)

Embedded JavaScripts are able to modify their HTML document in a variety of ways, including the insertion of further scripts. Uncarefully written scripts that for instance insert some part of the URL into the page, therefore can also cause the execution of contained scripts. This attack vector was first described by Amit Klein in [3] and is known as 'type0' or 'DOM-based' XSS.

The payload is not necessarily extracted from the URL. Significant for this type of attack is solely, that the malicious script is not as such existent in the document sent from Webserver to browser. It is for example imaginable, that some value in the database is written into an attribute's value during generation, and, while the page is rendered in a browser, reinserted elsewhere in the page by a JavaScript. An attack, which places a script at the appropriate location in the database, would therefore be as persistent as stored XSS, but incredibly harder to detect.

Combinations

XSS attack vectors are usually easily combinable. We will go through some short examples:

Obfuscation: When an adversary is performing a stored XSS attacks or posting a link in a forum, his address may be logged by the server. As everything necessary for an attack can also be performed by a script, it is quite easy to 'chain' XSS attacks across multiple servers to obfuscate one's identity and complicate prosecution. A simple example is a reflected XSS attack whose payload is performing a stored one.

Persistence: Cookies are transmitted in the HTTP header and can therefore also be seen as parameters. In rare cases it may be possible to perform reflected XSS with the cookie's value. By setting the cookie via `document.cookie`, a reflected attack can in this cases achieve persistence.

Bypassing Access Restrictions: If an attack vector requires special privileges, a successful XSS attack on a privileged user can obviously be used to bypass such restrictions. In environments where every user has write access to a special site (community profiles, for example), applying this idea recursively can lead to viral behaviour (see [46]).

Every attack vector can be used with any of the following payloads. This also holds for combined vectors.

2.2.2. Malicious Payloads

For the reasons outlined in the beginning of this Section, only executable payloads will be mentioned here. These will be categorized by goal as this is a good way to gain an overview of what XSS is capable of. The list will be followed by a short discussion as to why XSS makes the most sense in Web Applications with sessions.

Denial of Service and Content Defacement will be omitted here, since the threats they constitute cannot be considered as serious as the ones discussed in the remainder of this Subsection. This selection, of course, does not claim completeness and a malicious payload can obviously serve the attainment of multiple goals simultaneously.

Credential Stealing

In Web Applications using sessions, it is, with some exceptions, possible to access the authentication credential with JavaScript. Cookies, for instance, can usually be accessed via `document.cookie`. Although both Internet Explorer [37] and Firefox [38] are supporting HTTP-only-cookies by now, the majority of Websites is still not using them. According to the same origin policy, this credentials thus can also be accessed by a malicious payload (like in the example in Subsection 2.2.1) and, for instance, be sent to an arbitrary server as part of a request. The credentials are then allowing an adversary to impersonate the user with respect to the Web Application.

Other information, like those entered on the page by the victim, can also be compromised, as forms can be manipulated to redirect submitted data.

Request Forgery

Implicit authentication causes browsers to automatically include credentials in every request to a restricted page (see Subsection 2.1.4). As this is done independent of how the request was initiated, all access rights of the victim can be exploited by a malicious script in his browser. Attacks causing the victim's browser to sent authenticated requests are called Cross Site Request Forgery (XSRF) and form the basis for the combination examples mentioned under 'Combinations'. In simple cases, request forgery is also possible with non-executable payloads, but scripts obviously are allowing much more powerful attacks like using the POST-method or sequences of interdependent request-response pairs. These attacks are called 'Browser Hijacking' in [1] and are used for instance in the MySpace-Worm [46].

In-Browser Propagation

In [1], two techniques for propagating XSS attacks were mentioned. The term 'propagation' in this context means that the malicious JavaScript stays active in the background, while the user continues to surf the Website. This techniques are clarifying that a single XSS vulnerability may be sufficient to compromise an entire application.

AJAX Hijacking

Due to their asynchronous nature, building secure AJAX Applications is an even more difficult and currently less understood problem. Nevertheless, it could already be shown that a successfully injected script is able to hook itself into-, and therefore manipulate, any Object belonging to the AJAX Application, using a technique called "Prototype Hijacking" [2].

LAN-Scanner

All goals and techniques discussed so far do only make sense in Web Applications with sessions. Credential Stealing is nonsense if there is no authentication, Request Forgery is meaningless without access restrictions and both In-Browser Propagation and AJAX Hijacking techniques are only extending attacks to a greater scope. LAN-Scanning methods, however, clearly differ as they are really attacking the client instead of his relation to the Web Application.

Many Web surfers are accessing the WWW over home- or corporate networks. As discussed in [4], JavaScripts downloaded from the Internet are able to portscan hosts on the local

network. With some very simple tricks, they are able to identify hosts, check for Webservers and even fingerprint running Web Applications. If a local Web Service can be identified, it can be sent GET and POST requests.

JavaScript LAN-scanners work regardless of whether they were previously injected into some benign Website or just placed on one controlled by an adversary. Employing them as XSS payloads, however, may be the only method to target users protected by the noscript-plugin (see Chapter 7). As long as the vast majority of WWW-users are unprotected, and the adversary is not targeting someone in particular, there is no reason to do so, though.

XSS attacks on sites not having a session implementation of some kind thus can thus be regarded as futile.

2.3. Mozilla Firefox

Mozilla Firefox is open source browser and the basis of the prototypic implementation of our concept detailed in Chapter 5. This Section gives a conceptual overview over the inner workings of Firefox in order to enable the reader to follow the more technical details mentioned there.

The name Mozilla refers to both an Application Suite and a Cross-Platform Application Framework used thereof. Both are based on a codebase released by Netscape under a free software/open source license in 1998. Since then, it has been maintained by the Mozilla Foundation, and contains, amongst others, the Firefox browser and the Thunderbird email client. Like all Mozilla Applications, they are based on the Gecko Rendering/Layout Engine, the Necko Network Library, the XUL Toolkit, and XPCOM. While the latter is providing a Cross-Platform Development Framework for modular applications, and therefore constitutes the very foundation of everything else, XUL allows the implementation of scripted User Interfaces and is heavily relying on the SpiderMonkey JavaScript Engine. This Engine is also invoked, when Gecko encounters scripts within a page it should render, and can even be used to run entire XPCOM components, that were written in JavaScript, via a bridging technology called XPConnect. In Firefox, there are thus two types of JavaScript code, both running in the same Engine. All client-side code embedded in documents from the Web is executed sandboxed, to prevent access to internal methods and enforce the same origin policy, while everything else - called 'privileged code' - is considered a part of the application and therefore granted access to everything XPConnect provides an interface for.

After this brief overview, we will shortly detail some component important in the context of this thesis.

2.3.1. XPCOM

The Cross Platform Component Object Model (XPCOM) is a framework for developing modular applications. Basic idea is splitting an application's code into several components, interacting with each other via exported interfaces. While XPCOM is written in C++, a lot of language-bindings exist, for instance allowing it's use from Python (PyXPCOM), Java (JavaXPCOM), or, as mentioned previously, JavaScript (XPConnect). Interfaces are written in a language called Cross Platform Interface Definition Language (XPIDL), which is allowing any component, independent of similar mechanism in it's native language, to specify it's interface in a way all other components can understand.

All Mozilla Applications thus are composed of a vast number of small, reusable components, each exporting their interface in a way even JavaScripts can use. Along with XUL, XPCOM thus allows a rich environment for plugin developers.

2.3.2. XUL

All User Interfaces in Mozilla Applications are defined in the XML User-Interface Language (XUL). As the name suggests, it is based on XML and, like HTML, therefore provides a tree-like structure accessible from embedded JavaScripts via a Document Object Model (DOM). Structurally, XUL Interfaces and Webpages therefore a very similar. This can most convincingly be demonstrated by accessing the URL `chrome://browser/content/browser.xul` with Firefox, which is loading the browser window itself as a document.

2.3.3. Necko, Gecko and SpiderMonkey

During the document loading process, Necko, Gecko and SpiderMonkey are closely interacting to retrieve, decode, parse, layout and render the page, while also executing active content. Necko thereby is responsible for all tasks related to the retrieval of documents via the HTTP protocol. Afterwards, being handed the content stream, Gecko's parser is constructing a DOM-tree, from time to time invoking SpiderMonkey or other plugins to interpret executable parts. As scripts are able to modify the DOM, parsing, layouting and rendering steps have to be partly repeated and the page's layout is in a constant flow.

2.4. Intrusion Detection Systems

According to NIST [16] Intrusion Detection is "the process of monitoring events occurring on a computer system or network and analysing them for signs of possible incidents, which are violations or imminent threats of violation of computer security policies, acceptable use policies, or standard security practices" and an Intrusion Detection System (IDS) is "software that automates the intrusion detection process". Common to all IDSs is that they cannot provide absolutely accurate detection. We call an alarm raised on a benign event, a false-positive and a malicious activity remaining undetected, a false-negative.

Although a large number of different IDSs were developed over the years, only one distinction is relevant in this paper:

2.4.1. Signature-based Intrusion Detection Systems

A Signature-based Intrusion Detection System compares the state of a system against a repository of 'attack signatures', which are patterns of known attacks. This type of IDS is especially effective against the known attack types in it's repository, but completely fails to detect anything else. Without an up-to-date repository it thus is pretty useless. Signature-based Intrusion Detection, from its philosophy, is strongly related to blacklisting ("Allow anything but the following...").

2.4.2. Anomaly-based Intrusion Detection Systems

Contrarily, Anomaly-based Intrusion Detection uses a profile, capturing the normal behaviour of a system. Any significant derivation from this profile is considered malicious and reported.

Their philosophy thus is corresponding to whitelisting ("Deny everything except the following..."). An Anomaly-based IDS is able to detect unknown as well as known threats, as long as they cause a significant change in the system's behavior.

Before an Anomaly-based IDS can be applied, an initial profile has to be generated, by observing the normal system behaviour for a certain time period. This is called the *learning*-phase (or *training*-phase). The *learning*-period should be chosen sufficiently long for every normal system behaviour to occur at least once, for it would otherwise cause a false-positive. A problem of this type of IDS is that malicious behaviour occurring during *learning*-phase is incorporated into the profile and henceforth considered normal. A secure state of the initial system therefore is a necessary precondition for applying any Anomaly-based IDS.

The concept outlined in Section 4.3 in fact is an Anomaly-based IDS and the terminology introduced in this Subsection will prove to be useful for it's discussion.

2.5. Formal Primitives

To fully understand the more formal parts of this thesis, some basic notions and notations are necessary. We will briefly introduce them here even though most are part of every standard undergraduate course.

First, Subsection 2.5.1 is defining some basic notations and abbreviations. Subsection 2.5.2 then introduces strings as formal primitives. Although most readers may already be familiar with the formal treatment of character sequences, it provides a good foundation for the following definitions of Suffix Trees and Semi-Thue Systems.

Subsection 2.5.3 will briefly discuss directed graphs, a formalism used in Chapter 3 for capturing the structure of Web Applications in so-called Linkage-Graphs, as well as prerequisite of defining rooted trees in Subsection 2.5.4. Trees themselves are in this thesis only informally used in the form of DOM-trees, but required for the introduction of Suffix Trees in Subsection 2.5.5. The way they are defined, however, additionally allows a more formal treatment of different crawling techniques in Chapter 3.

Suffix Trees, as well as their generalization, are introduced in Subsections 2.5.5 and 2.5.6. This sophisticated data structures are advantageous for solving Pattern Matching problems and therefore heavily used in Section 4.2.

Finally, Subsection 2.5.7 is defining Semi-Thue-Systems, a very basic type of string-rewriting system used also in Section 4.2 to formally describe character encodings.

2.5.1. Notations

- $\exists_1$ means "there is exactly one". Formally it combines existence with uniqueness:

$$[\exists_1 x : P(x)] \leftrightarrow [\exists x : P(x)] \wedge [(P(a) \wedge P(b)) \rightarrow a = b]$$

- We sometimes use "iff" as an abbreviation for "if and only if".
- $\mathbf{P}(M) = \{M' | M' \subseteq M\}$ denotes the power set of M.

2.5.2. Strings

Definition 2.5.1 *A **string** $s = s_1 s_2 ... s_n$ over some alphabet Σ is a sequence of characters $s_1, s_2, ..., s_n \in \Sigma$. n is called the **length** of s and written $|s|$. We denote the set of all strings*

over Σ as Σ^* (Kleene closure). If the alphabet can be clearly derived from context, we just call s a string. $\epsilon \in \Sigma^*$ with $|\epsilon| = 0$ is called the empty string. $s[i] = s_i$ has proven to be a useful notation for accessing single characters.

Like sets, strings obviously feature some kind of containment relation:

Definition 2.5.2 *Let $s = s_1 s_2 ... s_n$ be some string over an alphabet Σ. For all $u, x, v \in \Sigma^*$ with $s = uxv$ we call u a **prefix**-, v a **suffix**- and x a **substring** of s (Note that u, x, v can be empty). We further introduce the function*

$$substring : \Sigma^* \mapsto \mathbf{P}(\Sigma^*) : substring(s) = \{x \in \Sigma^* | \exists u, v \in \Sigma^* : s = uxv\}$$

mapping each string to the set of its substrings, and define $prefix(s)$ and $suffix(s)$ accordingly.

A substring thus is some part of a string uniquely identified by a pair of start- and end-index. A prefix is a substring starting at the beginning and a suffix is a substring ending at the end. It therefore is obvious that $\forall s \in \Sigma^* : suffix(s) \cup prefix(s) \subseteq substring(s)$.

Definition 2.5.3 *Let $s = s_1 s_2 ... s_n$ be some string over an alphabet Σ. We call each string $q = s_{i_1} s_{i_2} ... s_{i_m}$ with $0 \leq m \leq n$ a **subsequence** of s iff $1 \leq i_1 \leq i_2 \leq ... \leq i_m \leq n$. As above the function subsequence(s) is mapping each string to the set of its subsequences.*

Subsequences can be thought of as substrings that can also skip characters. It clearly holds $\forall s \in \Sigma^* : substring(s) \subseteq subsequence(s)$.

Note that a string of length n may have up to $n + 1$ prefixes, $n + 1$ suffixes, $\frac{n^2+n}{2} + 1$ substrings and 2^n subsequences.

2.5.3. Directed Graphs

Informally, a directed graph is a set of objects called vertices which are connected by arrows called edges. This very general concept can be used to model all sorts of inter-object relationships. Of special interest in this thesis is the Linkage Graph, a directed graph where the vertices are representing hypertext documents and arrows between them are describing hyperlinks. Chapter 3 uses it to describe the functionality of a crawler. More formally, we define

Definition 2.5.4 *A **directed graph** is a 2-tuple $G = (V, E)$ consisting of a set of vertices V and a relation $E \subseteq V^2$. We say $a \in V$ is connected to $b \in V$ iff $(a, b) \in E$ and write $a \rightharpoonup b$ in G.*

The directedness of arrows directly leads to the notion of 'inbound' and 'outbound' edges as well as vertices:

Definition 2.5.5 *For every vertex a in a directed graph $G = (V, E)$ we define the sets of **inbound** and **outbound** vertices as*

$$\begin{aligned} in_G(a) &= \{b \in V | b \rightharpoonup a\} \\ out_G(a) &= \{b \in V | a \rightharpoonup b\} \end{aligned}$$

Edges obviously also provide some means of traversing the graph:

Definition 2.5.6 *out A* ***path*** *in a directed graph $G = (V, E)$ is a sequence of vertices $v_1, v_2, ..., v_n \in V$ with*

$$v_1 \rightharpoonup v_2 \rightharpoonup ... \rightharpoonup v_n \text{ in } G$$

the number of edges traversed $(n-1)$ is called the **length** of the path. If there is a path from a vertex a to a vertex b we write $a \overset{*}{\rightharpoonup} b$. The $\overset{*}{\rightharpoonup}$-relation is reflexive ($\forall a \in V : a \overset{*}{\rightharpoonup} a$) as every node is connected to itself by a path of length 0.

Definition 2.5.7 *A path $v_1, v_2, ..., v_n$ of length ≥ 1 in a directed graph G is called a* ***cycle*** *iff $v_1 = v_n$. If there are no cycles in a directed graph it is called* ***directed acyclic****.*

Note that in a directed acyclic graph $G = (V, E)$, the edge relation E must be irreflexive ($\forall a \in V : (a, a) \notin E$).

2.5.4. Rooted Trees

Trees are data structures structurally resembling their biological namesakes while their terminology is more evocative of genealogy. HTML and XUL documents can be represented as a labeled tree (a tree with a function assigning each edge a label string), which is called their DOM-tree. The following definition intentionally resembles the way a crawler treats a Web Application's Linkage Graph in order to make it efficiently retrievable.

Definition 2.5.8 *A directed acyclic graph $T = (N, E)$ is called a* ***rooted tree*** *iff*

$$\begin{array}{lll} 1. & \exists_1 r \in N : & |in_T(r)| = 0 \\ 2. & \forall a \in N : & a \neq r \rightarrow |in_T(a)| = 1 \end{array}$$

r is called the **root** of T and T is said to be rooted in r. In a rooted tree the vertices are called **nodes** and the edges are usually perceived as a parent-child relation. Consequently, if two nodes share the same parent, they are called **siblings**. A **leaf** is a node with no outbound edges. All other nodes are called **inner nodes**. For convenience we thus define the following functions

$$\begin{array}{rrll} root_T: & N \mapsto \mathbb{B}: & root_T(a) \leftrightarrow |in_T(a)| = 0 \\ leaf_T: & N \mapsto \mathbb{B}: & leaf_T(a) \leftrightarrow |out_T(a)| = 0 \\ parent_T: & N \mapsto N: & parent_T(a) = b \longleftrightarrow b \rightharpoonup a \\ children_T: & N \mapsto \mathbf{P}(N): & children_T(a) = out_T(a) \end{array}$$

If there is a path from a node a to a node b, a is said to be an **ancestor** of b. The root is ancestor of all nodes in the tree. We call the length of the path from the root to a node a the **level** of a. The highest level in a tree is called its **height**.

Note that *parent* indeed is a function due to the restriction in the number of inbound nodes, and that the tree is ensured to be connected for only the root is allowed to have no inbound edge and cycles are not allowed.

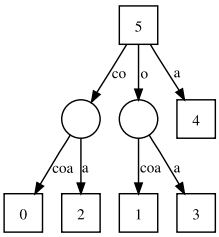


Figure 2.2.: Suffix Tree for the word 'cocoa'

Definition 2.5.9 *In a rooted tree $T = (N, E)$ each node $n \in N$ is the root of a* ***subtree*** *$T_n = (N_n, E_n)$ with*

$$\begin{aligned} N_n &= \left\{ a \in N | n \overset{*}{\rightarrow} a \right\} \\ E_n &= \{(a,b) \in E | a \in N_n\} \end{aligned}$$

When algorithms are working on trees, it is often necessary to perform some operation on each node in a tree (called visiting a node). The nodes can be visited in different orders called **traversals**. Relevant for this thesis is the distinction between **depth-first** and **breadth-first** traversals. Depth-first traversals are by far the most common, since implementing them recursively is very simple and straightforward. When visiting a node, they first traverse its entire subtree before continuing with its siblings. A depth-first traversal is called **pre-order** if the node itself is visited prior to the nodes in its subtree and **post-order** the other way around. For a large tree of n nodes, traversing it depth-first has the additional benefit of requiring memory linear with its height, as the algorithm only needs to remember the path to the current node. Contrarily, in breadth-first traversal, which means visiting all nodes of a certain level before continuing to the next, one has to remember at least all inner nodes of the last level. This in the worst case are $\lfloor \frac{n-1}{2} \rfloor$ nodes, which scales linearly with the total number of nodes.

2.5.5. Suffix Trees

An especially useful type of tree is the Suffix Tree, as it helps to efficiently solve several important problems on strings, like the longest common substring problem. Trees in general can be effectively used to store a set of strings, by labeling the edges in a way that every path from the root to a leaf corresponds to a certain string derivable by concatenating the edge labels in the order of traversal. A Suffix Tree in fact does exactly that with the set of suffixes of a certain string. We define a Suffix Tree following [34, page 90].

Definition 2.5.10 *For a string $s \in \Sigma^*$, a tree $T = (N, E)$ and an edge-labeling function*

*$label : E \mapsto \Sigma^*$, the 3-tuple $STree(s) = (N, E, label)$ is called a **Suffix Tree** for s iff*

$$\begin{aligned}
\forall e \in E : label(e) \in substring(s) \wedge |label(e)| > 0 \\
\forall n \in N : \neg leaf(n) \wedge \neg root(n) \rightarrow |children(n)| \geq 2 \wedge \\
\forall a, b \in children(n) : label((n,a))[1] \neq label((n,b))[1] \\
\forall q \in suffix(s), |q| > 0 : \exists_1 v_1 \rightharpoonup v_2 \rightharpoonup ... \rightharpoonup v_n, root(v_1), leaf(v_n) : \\
label((v_1, v_2))label((v_2, v_3))...label((v_{n-1}, v_n)) = q
\end{aligned}$$

Since a Suffix Tree like that cannot be constructed for all strings, s is usually padded with a terminal symbol not contained in it previously (denoted \$) to ensure that no suffix is prefix of some other. For quickly locating substrings, it is also common to store in each leaf the start-index of the suffix it represents. $STree(cocoa)$ is depicted in Figure 2.2.

Note that it follows directly from the definition that a Suffix Tree with n leafs has at most $n - 1$ inner nodes.

With such a tree for some string s one can for instance search for any substring q of s in time linear with $|q|$ and completely independent of $|s|$.

In [32], Esko Ukkonen presented an on-line algorithm (known as Ukkonen's Algorithm) able to construct the Suffix Tree for an arbitrary string s in time linear with the length of s.

2.5.6. Generalized Suffix Trees

A generalized Suffix Tree is a generalization of the Suffix Tree introduced in the previous chapter. Instead of storing all suffixes of one string, it stores all suffixes of a set of strings. Structurally, it is completely identical to a normal Suffix Tree, but each string needs to be padded with a unique termination symbol (or sequence) denoted $\$0, \$1, ...$ and each leaf-node has to store a pointer to the respective string along with the start-index of its suffix. This generalization is required, for example, for finding the longest common substring in a set of strings. Once the tree has been constructed, this can be easily accomplished by traversing it in post-order, while maintaining a bit-string in the following way: In each leaf node, the bit, corresponding to the string the leaf is storing a pointer to, is initialized to 1 while all other remain 0. Each inner node henceforth combines the bitstrings from all its children with a bitwise-and operation. As each node in a generalized Suffix Tree corresponds to a substring of at least one of its strings, the bitstring in each node is representing a list of strings it occurs in. For solving the longest common substring problem above, we just, during traversal, have to remember the node with the longest corresponding substring (concatenation of all labels in its path) and all bits set. Note that this algorithm can be easily adapted for finding the longest common substring present in $\geq k$ strings for some k.

With a trick outlined in [34, page 116], Ukkonen's Algorithm can be adapted to construct the generalized Suffix Tree for a set of strings $s_1, s_2, ..., s_n$ with combined length $N = \sum_{i=1}^{n} |s_i|$ in time linear with N. This construction is used in Section 4.2 to create an efficient pattern matching algorithm.

2.5.7. Semi-Thue Systems

Semi-Thue Systems are a very basic type of string rewriting systems. They are used in Section 2.2.1 to formally describe character encodings in a simple yet sufficiently powerful way to capture their important properties. All other notions (except for the inversion) are

simplifications of what Morita, Nishihara, Yamamoto and Zhang defined for Uniquely parsable Grammars (UPG) in [31]. As we are not giving a full proof in Section 2.2.1, using the full UPG formalism would unnecessarily complicate things.

Definition 2.5.11 *For some finite alphabet Σ a **Semi-Thue system** (STS) is a set of rewrite rules $R \subseteq \Sigma^* \times \Sigma^*$ written $u \to v$ for $u, v \in \Sigma^*$. **Derivations** in this system are defined as*

$$\begin{aligned} w_1 \Rightarrow w_2 &\longleftrightarrow \exists a, b \in \Sigma^* : w_1 = aub \wedge w_2 = avb \wedge u \to v \in R \\ w_0 \overset{n}{\Rightarrow} w_n &\longleftrightarrow \exists w_1, w_2, ..., w_{n-1} : w_0 \Rightarrow w_1 \Rightarrow ... \Rightarrow w_n \\ w_1 \overset{*}{\Rightarrow} w_2 &\longleftrightarrow \exists n \in \mathbb{N} : w_1 \overset{n}{\Rightarrow} w_2 \end{aligned}$$

Definition 2.5.12 *For every $R \in STS$ there exists an **inverse STS** R^{-1}. Every derivation $w_1 \overset{*}{\Rightarrow} w_2$ in R can be reverted by a derivation in R^{-1} applying the reversed rules in reverse order. Applying a reversed rule $v \to u$ from the inverse STS R^{-1} is called a **reduction** in R and written $w_2 \Leftarrow w_1$. The reflexive, transitive closure is defined identical to Definition 2.5.11.*

Definition 2.5.13 *For some string w and an STS S we define the function*

$$\varphi_S : \Sigma^* \mapsto \mathbf{P}(\mathbb{N} \times S) : \varphi_S(w) = \{[i, \alpha \to \beta] \,|\exists u, v \in \Sigma^*, |u| = i - 1 : w = u\beta v\}$$

*and call every $[i, \alpha \to \beta] \in \varphi_S(w)$ a **reversely applicable item** of S in w. A direct reduction $w_2 \Leftarrow w_1$ using the rule $\alpha \to \beta$ at index i thus can be denoted $w_2 \overset{[i,\alpha\to\beta]}{\Longleftarrow} w_1$.*

3. Data Collection

"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to", said the Cat.
"I don't much care where-" said Alice.
"Then it doesn't matter which way you go" said the Cat.
"-so long as I get somewhere", Alice added as an explanation.
"Oh, you're sure to do that", said the Cat, "if you walk long enough."

–Alice in Wonderland

First and foremost, we had to collect a large amount of data for analysis in order to get an idea about the problem and the real-world environment. Based on this data, our concepts were developed and our implementations were evaluated. However, for evaluating the XSS detector under realistic conditions, an enormous amount of real-world surfing data was necessary. More specifically, a large number of HTTP request-response-pairs, belonging to a number of dynamic Web Applications, were needed. As gathering real-world surfing data would either violate privacy protection laws, or require a tremendous amount of funding, the data had to be automatically collected by a crawler.

This Chapter consists of two parts. Section 3.1 outlines the design decisions made while constructing the crawler and Section 3.2 describes the process of gathering HTTP request-response pairs.

3.1. Constructing the Crawler

Crawlers are programs for automatically retrieving huge amounts of documents from the WWW. This is very useful for mirroring a Website or building a search index. As most Webservers are disallowing their directory structure to be traversed, crawlers have to follow the link structure, starting from some initial set of documents. A variety of crawlers have been designed for a lot of different purposes. For ours, the following requirements were crucial:

parallelity In order to limit the impact of slow / non-responding Webservers and also to speed up the overall process and fully utilize the available bandwidth, the crawler has to crawl all Web Applications in parallel rather than sequentially.

coverage A collection of pages with maximal variety should be retrieved from each Web Application. Ideally, each type of page the Website offers should be included.

stability It should be possible to collect data in a completely automated fashion over a period of several month. The crawler therefore should not crash or stop working properly on unexpected events like grossly malconfigured Webservers, huge or corrupt documents, etc.

repeatability To simulate page revisits and gain an insight in chronological Web Application changes, it should be possible to make the crawler regather exactly the same documents retrieved in an earlier run. This obviously is severely limited if, for example, a Web Application is including sessionIDs into it's URLs.

sessions In order for the results to resemble real surfing data as much as possible, certain session-mechanisms have to be supported.

browser simulation There should be as few technical barriers as possible preventing the crawler from working accurately. Of course, the crawler will not be able to follow links embedded in a Flash-Menu or crawl AJAX-applications, but it should at least follow meta-redirects and handle frames correctly.

Filter only HTML content should be retrieved. All other content should be filtered by it's MIMEtype.

Conformity The crawler should be able, for instance, to follow HTTP redirects.

HTTP Headers As the data is intended for evaluating our concept, we need to store the entire request as well as the response header along with each document.

After testing several open-source crawlers, we were not able to find one which meets all our requirements. The following cases were encountered most often:

- prioritizing speed, many crawlers do not parse the HTML pages, but instead use some sort of pattern-matching to find the links. This algorithms were found to be mostly inaccurate (finding links even in comments or not recognizing links with additional attributes before `href`), and mostly did not include frames or meta-redirects.

- many Web crawlers are intended for use as backends of search engines. Consequently, they are optimized to find the most relevant pages first. Techniques like On-line Page Importance Computation (OPIC) are neither necessary in this context nor tend to improve page variety.

- No crawler tested, supported any session mechanisms

We therefore decided to implement our own crawler in ruby. For extracting the links (`a`, `area`, `meta`-redirects, `frames`, `iframes`), we used the hpricot-library. More Information can be found in Appendix A.

The following Subsections will give an insight in how we constructed a crawler meeting our requirements. Subsection 3.1.1 presents the crawling strategy, an algorithm for deciding which links to follow and which pages to retrieve. Afterwards, Subsection 3.1.2 introduces a robust architecture allowing the crawler to operate for a long time without human intervention. Subsection 3.1.3 describes how the crawler was able to save both requests and responses entirely and finally, in Subsection 3.1.4, a method for dealing with Web Applications, that are requiring sessions, is presented.

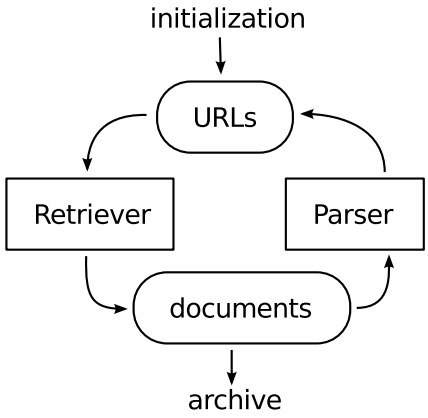


Figure 3.1.: Architectural overview of the crawler

3.1.1. Linkage Graph Traversal

A Linkage Graph is a directed graph with each page of a Web Application being represented as a node and each edge from node a to node b representing a link in page a referring to page b. A crawler basically is downloading each page it knows of, and then follows links to find more pages, until some termination condition is met. For avoiding duplicate downloads, crawlers usually remember all retrieved documents. In the graph, this would be equivalent to marking each visited node while traversing the graph's edges starting from some set of start-nodes. If also highlighting the edges used for traversal, one would end up with a forest of trees each rooted at one of the start-nodes. Note that this, apart from the multitude of root nodes, exactly follows the tree definition in Subsection 2.5.4. This is exactly the reason why we can speak of crawling strategies in terms of classical tree-traversals:

As a depth-first traversal would often get stuck in highly interconnected areas (like, for instance, myspace-profiles) and a breadth-first-traversing crawler would only crawl the surface of a complex site, we have to strike a balance between those two which can provide us with a high variety of pages in both cases.

Our implementation traverses the graph in what we call bounded breadth-first order. It only differs from breadth-first order in that only a maximum number of 5 links per page are followed. These are selected randomly from all links on that respective page. If an average of 3000-5000 pages per Web Application were crawled in each run, an average depth of at least 5 links could be assured.

3.1.2. Architecture

As providing sufficient stability was a main concern, the crawler was implemented as a number of separate processes, interacting in a transaction-based way with a mysql-database. All data-handling processes (called the workers) were memory- as well as runtime-constrained and periodically restarted by a scheduling process coordinating their efforts. All Websites were crawled in parallel, as every worker randomly chooses which page to retrieve/parse next. Retrieved pages were added to a pool, from which the parsers were selecting their next work items. During the process of parsing, a list of links was extracted, and after corrupt- as well as off-domain links had been removed, a maximum number of 5 links were randomly chosen

for addition to the crawl-list (see Figure 3.1).

3.1.3. Retrieval

We chose to use the stable HTTP document retrieval tool GNU-wget, since it automatically follows HTTP-redirects and retries unsuccessful downloads. Like all other tools, however, wget does not offer any option for retrieving the header information (neither for request nor response) along with the data. A slight modification for adding this option therefore was unavoidable (see Appendix A). Wget was run as a separate process with several parameters adjusting it's behaviour in the following ways:

Referrer the referrer-header was always set to the URL of the document containing the link. As this behaviour sufficiently mimics that of a browser, referrer-checks could be avoided.

Cookies Every crawled site was given a separate file to store cookie data. This respective file was read anytime a document of that site was requested, and updated as new cookies had been set. If a page uses sessions, this is absolutely necessary to retrieve something else than a 'your session timed out'-message.

User-Agent Meanwhile, it is quite common in the browser community to set the 'User-Agent'-header value to that of a popular browser, since many Websites are refusing to work otherwise. We made wget mimic Firefox, since a surprisingly high number of pages were answering our queries with 'sorry, your browser is not supported'-messages.

Timeouts and Retries In order to optimize our retrieval rate, we decreased several timeouts and limited the number of retries to 2.

3.1.4. Sessions

In order to deal with pages requiring session credentials for access, we implemented login- as well as logout-scripts into our crawler. Login-scripts were executed at the beginning of each crawling, and initialized the respective cookie-files after performing some login operation. Logout-scripts were called to cleanly notify the Websites that the credentials were no longer needed.

3.2. Automatically gathering HTTP Traffic Data

For evaluating the detector on a realistic data-set, a total of 95 dynamic Web Applications were crawled. To simulate an average Web-surfing experience, and at the same time enforce some diversity, we first chose suitable Web Applications from the Top 500 list of the most popular ones provided in [39], and picked the rest from the top ranked answers a search engine provided to queries for 'wiki', 'forum', 'news site', 'online shop', 'community' and 'blog'. Most of this sites were providing sufficient access for anyone, but as a few of them either severely restricted or completely denied access without an established session (e.g. orkut.com), we had to create accounts and provide our crawler with appropriate login scripts.

Given the list, the crawler has been continuously collecting data for about a month, using the following 3-day routine:

- 1st day: crawl all Web Applications and assign the result randomly to one of two groups.

- 2nd day: recrawl all results in the first group
- 3rd day: recrawl all results in the second group

This way every Web Application was crawled once every third day and for every page crawled, new versions were fetched in 3-day cycles. After one month, the crawler had gathered approximately 500.000 pages summing up to about 20GB of uncompressed HTML documents. During evaluation, however, we discovered some pathological cases:

Every user of `studivz.net`, for instance, is not only required to log in, but also has to periodically answer CAPTCHAs to proof being human. Our crawls from studivz were thus limited to the publicly available area, which was quite small.

A complete list of all crawled applications can be found in Appendix C.

The gathered data will be furthermore referred to as the 'data-set'. Based on it, we developed the concepts presented in the next Chapter.

4. Detecting XSS from HTTP Traffic

"In a staged play, even the audience is part of the act."

–Aramaki Daisuke, Ghost in the Shell SAC

The two concepts at hand are constituting a general Cross Site Scripting detector, deriving security-relevant information from HTTP traffic using data mining techniques. As both concepts are focusing on the determination of a script's legitimacy, a browser is used for finding the scripts within the HTML documents. The component doing this is called 'Script Finder' and will be outlined in the following Section, before the two concepts are discussed in detail.

Section 4.2 is presenting the first one, called 'String Matcher'. Basic idea is to detect reflected XSS attacks by correlating a query's parameters with the scripts found in the replied document. As outlined in Subsection 2.2.1, with this type of attack the payload is included in the request's parameters and should therefore - in case of a successful attack - match some script in the document. There are, however, a lot of complications due to character encodings and input filters.

The second concept is called 'Scriptbase' and detailed in Section 4.3. It is an Anomaly-based IDS on HTTP traffic. For each Web Application, a profile is derived from observed traffic, consisting of information on the scripts normally contained in it's pages. After a training period, the profile can be used to find unknown scripts in this application, thus detecting both reflected and stored XSS attacks.

Both methods, however, are not able to detect any DOM-based attacks, as these happen in the browser and are not traceable from HTTP traffic without simulating all active content.

After Section 4.2 and 4.3 provided a detailed understanding of the concepts, two application scenarios will be discussed. The idea of using the detector as a server-side IDS is detailed in Section 4.4, while Section 4.5 will focus on possible applications on the client-side.

4.1. Using the Browser as a Script Detector

The first part of XSS detection is finding the scripts. As the browser vendors develop their parsing engines further and further to include even more standards, languages and features, a constant battle is fought between those writing input-filters and those coming up with even more unexpected evasion techniques. One of the main sources of complexity in this problem is the multitude of browsers. Even if someone was both capable and willing to perform a complete, detailed analysis of every popular parsing engine, closed-source browsers would do their part in further complicating the issue. Therefore, especially if certain types of rich content should be allowed, good input filters are not available off the shelf.

As already mentioned initially, browsers are far better suited for script detection than Web Applications are. This is because they know exactly which scripts they would execute and it makes no sense to filter anything else. Utilizing the browser's parser as a script detector would therefore perfectly solve the problem. For performance, most of the other features should be

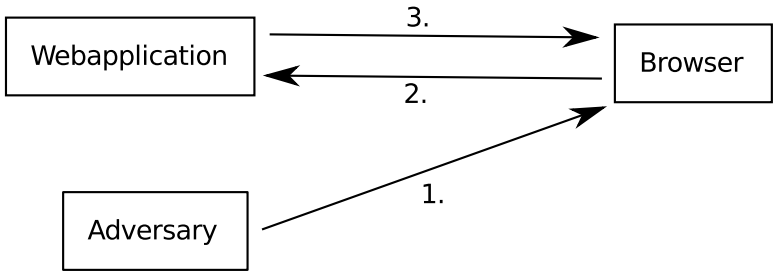


Figure 4.1.: Reflected XSS attack vector. 1. Adversary supplies malicious payload via a message board, email, instant messaging, etc. 2. When the user clicks the link, the payload is sent to the vulnerable Web Application. 3. After being reflected, the payload is executed in the victim's browser in the Web Application's context.

disabled, of course. Especially the execution of active content is undesirable, since only scripts that are included in the original page should be detected. Page layouting, rendering and loading of external content are not needed either.

Unfortunately, no browser currently supports to use it's parser for script detection. Open source browsers, however, can be patched to do so. We will develop such a patch for Mozilla Firefox as part of this thesis and Jim, Swamy & Hicks described in [18] how they managed to develop a quite similar alteration for the Konqueror and Safari Browsers. For supporting closed-source browsers, the vendors are obviously required to provide appropriate interfaces.

With this idea, getting rid of the one-browser restriction turns out to actually be a parallelization issue. Running a patched version of each browser one wants to support and feeding them the HTML-pages in parallel should be quite manageable given sufficient computational power. The results could be merged based on the script's positions in the pages.

4.2. Detecting Reflected XSS Attacks by matching Parameters against Script Codes

Figure 4.1 is depicting the attack vector of a reflected XSS attack. Usually on behalf of some user-action, a browser sends the malicious script to the server which in turn reflects it and sends it back as part of the document's body. Remarkable about this vector is the payload passing both the victim's browser and the server twice. It is part of both the request and response. This fact can be exploited for detection: One has to check all scripts contained in the response for congruence with some part of the request. Finding the scripts within the response should - as discussed in the previous Section - be taken care of by a browser.

The next Subsection (4.2.1) will derive an input filtering model to clarify the problem. It will be determined that two types of input filter activities are hindering our matching: encodings and removal filtering. Subsection 4.2.2 introduces a string normalization able to deal with the former and Subsection 4.2.3 develops a pattern matching algorithm allowing parameter-code matches despite removal filters.

name	example	items	function
HTML character entity references	< → <	&NN+;	*entity_ref*
HTML decimal character references	< → <	&#DD+;	*dec_ref*
HTML hexadecimal character references	< → <	&#xHH+;	*hex_ref*
JavaScript single escape character	< → \<	\C	*js_single_char*
JavaScript hex escape sequence	< → \x3C	\xHH	*js_hex*
JavaScript unicode escape sequence	< → \u003C	\uHHHH	*js_unicode*
URL encoding	< → %3C	%HH	*url_encoding*

Remarks: In the righthand forms NN+ represents a name assigned to each character in [12], DD+ and HH+ stand for arbitrary-length strings of decimal/hexadecimal characters referring to a Universal Character Set (UCS) code point (see [11]), C is the character itself and HH/HHHH a two/four-character hexadecimal encoding of the ASCII-code/UCS code point.

Table 4.1.: Table of relevant character encodings in the Web context

4.2.1. Input Filtering

More formally, the problem can be expressed as follows: Given the sets $P \subseteq \Sigma^*$ of all parameters in the request and $S \subseteq \Sigma^*$ of all scripts in the response, we are searching for certain kinds of similarities between at least one parameter string $p \in P$ and the code of at least one script occurrence $s \in S$. In order to exactly determine the kind of similarity we are searching for, we will shortly review what happens to a parameter string, once passed to the Web Application:

First, the Webserver surely URL-decodes all query parameters (see Table 4.1), before passing them to the generating program [10]. Depending on the Web Application, they afterwards are probably passed through various other decoding routines, in order to remove a number of encodings from any of their parts, and (hopefully) passed to an input filter for 'sanitation'. There are two basic types of sanitation performed in input filters: encoding and removal. Encoding means substituting any set of 'special' characters with a safe representation ensuring the string to be interpreted as plain text when part of the response. For HTML, three types of encoding are available: character entity reference, decimal character reference and a hexadecimal version of the latter. If the string is written into an already existing script, usually one of three JavaScript string escaping methods is applied instead (see Table 4.1).

The other method often used for input sanitation is the removal of characters or sequences. While in simple cases, it is quite easy to write a safe input filter using encoding, removal filters are very likely to leave some loophole open for evasion. Imagine, for example, an input filter removing the sequences `<script` and `<\script`. An adversary just has to supply an input like

```
<scr<scriptipt>do_something_malicious();</scr<scriptipt>
```

for successful evasion. Probably because of this - for many Web developers non-obvious - property, publicly available XSS-vulnerability lists reveal removal-filters to be the most evaded input filters only outnumbered by non-existing ones.

Our input filtering model thus is structured as depicted in Figure 4.2. The parameters are passed through some combination of decoding steps, removal filtered and afterwards applied some encodings again. While the first combination always contains the obligatory URL-decoding, the second one is allowed to be empty. The removal filter can of course be omitted as well.

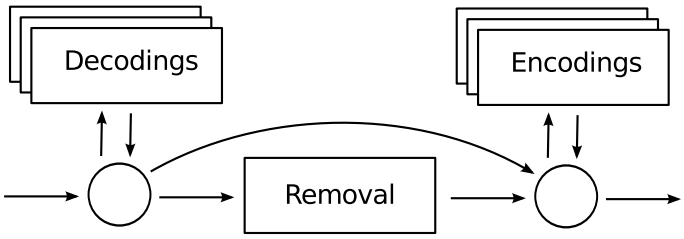


Figure 4.2.: Input Filtering Model. Parameters are passed in from the left, then applied an arbitrary combination of decoding routines, optionally removal filtered, and encoded an arbitrary number of times. The result is finally inserted into some script.

In order to make parameters and code comparable at all, we first have to find a way to deal with all the encodings before thinking of how to match them despite the removal filter. While the former (discussed in Subsection 4.2.2) is called 'normalization' the latter is detailed in Subsection 4.2.3.

4.2.2. Normalization

In this Subsection, we will define a normal form for strings allowing us to match them unrestricted by their encodings. First, a formal model of our input filters needs to be developed, though. Note that this Subsection will only concern about encodings, removal filters will be dealt with in the next.

Definition 4.2.1 *For some alphabet Σ, we define*

$$\begin{aligned} char\_encodings &= \{f : M \mapsto \Sigma^* | M \subseteq \Sigma, \forall x \in M : |f(x)| > 1\} \\ \forall E \subseteq char\_encodings : & \\ E^\Sigma &= \{g_f | f \in E\} \\ & with \quad g_f : \Sigma \mapsto \Sigma^* : g_f(x) = \begin{cases} f(x) & x \in domain(f) \\ x & else \end{cases} \end{aligned}$$

$$\begin{aligned} \forall f \in char\_encodings^\Sigma, M \subseteq \Sigma, I \subseteq \mathbb{N} : & \\ f_M : \Sigma \mapsto \Sigma^* : f_M(x) &= \begin{cases} f(x) & x \in M \\ x & else \end{cases} \\ f^I : \Sigma^* \mapsto \Sigma^* : f^I(x_1 x_2 ... x_n) &= g^I(x_1, 1) g^I(x_2, 2) ... g^I(x_n, n) \\ with\ g^I(x, i) &= \begin{cases} f(x) & i \in I \\ x & else \end{cases} \\ \forall F \subseteq char\_encodings^\Sigma : & \\ encodings_F &= \left\{ {f_M}^I | f \in F, M \subseteq \Sigma, I \subseteq \mathbb{N} \right\} \\ input\_filters_F &= \{f_1 \circ f_2 \circ ... \circ f_n | \forall i : f_i \in encodings_F\} \end{aligned}$$

The set of input filters is thus given by all functions that can be derived by combining functions from a set of character encodings. Additionally, the encodings may be restricted to an arbitrary subdomain (f_M) or applied only to an arbitrary subset of characters (f^I). Thus, if all used character encodings are elements of F, we can be certain that both the entire encoding phase and the inverse of the decoding phase from our input filter model (see Figure 4.2) are elements of $input\_filter_F$. In our case, we chose the following five character encodings

$$F_{IF} = \{entity\_ref, dec\_ref, hex\_ref, url\_encoding, js\_single\_char\}$$

which, due to their careful design, exhibit some nice structural properties we will exploit below for finding a normalization function $normalize : \Sigma^* \mapsto \Sigma^*$ satisfying the following requirements:

$$\forall x \in \Sigma^* :$$
$$1.\ \exists x' \in \Sigma^* : x' = normalize(x) \quad \text{(existence)}$$
$$2.\ \forall a, b \in \Sigma^* : [a = normalize(x) \wedge b = normalize(x)] \rightarrow a = b \quad \text{(uniqueness)}$$
$$3.\ \forall f \in input\_filters_{F_{IF}} : normalize(x) = normalize(f(x)) \quad (*)$$

While the existence and uniqueness properties are standard for normalizations, the third equation makes this one useful for our purpose. A function with this properties would enable us to match strings regardless of any combination of encoding functions applied. Before presenting a normal form satisfying all this requirements, we will have to introduce some additional notions for Semi-Thue-Systems (STS).

Definition 4.2.2 *A Semi-Thue-System S is called **strictly monotonous** iff*

$$\forall \alpha \rightarrow \beta \in S : |\alpha| < |\beta|$$

Definition 4.2.3 *For every set $F \subseteq char\_encodings$ we define a Semi-Thue-System $S_F = \{x \rightarrow f(x) | \exists f \in F : x \in domain(f)\} \in STS$ called the **associated STS** of F*

Lemma 4.2.4 *For every $F \subseteq char\_encodings$ the associated STS S_F is strictly monotonous.*

Proof 4.2.5 *For every $f \in char\_encodings$ the definition required $\forall x \in domain(f) : |f(x)| > 1$. Thus for every rule $\alpha \rightarrow \beta \in S_{\{f\}}$ it follows $|\alpha| = 1$, $|\beta| > 1$ and thus $|\alpha| < |\beta|$. The same is applicable for all $F \subseteq char\_encodings$.* □

It is important to note that any application of a function $f \in input\_filters_F$ to a string $w \in \Sigma^*$ is also a derivation in the associated STS S_F: $w \stackrel{*}{\Rightarrow} f(w)$. Applying any f^{-1} to w consequently is a reduction in S_F.

Definition 4.2.6 *A string $w \in \Sigma^*$ is called **irreducible** in $S \in STS$ if there is no reversely applicable item in it.*

$$irreducible_S = \{x \in \Sigma^* | \varphi_S(x) = \emptyset\}$$

Definition 4.2.7 *A **complete reduction** of a string $w \in \Sigma^*$ in $S \in STS$ is defined as*

$$reduce_S(w) = \begin{cases} w & if\ \varphi_S(w) = \emptyset \\ reduce_S(w')\ with\ w \overset{[i,\alpha\to\beta]}{\Longleftarrow} w', [i, \alpha \to \beta] \in \varphi_S(w) & else \end{cases}$$

the result of which is clearly irreducible due to the recursion's end condition.

Theorem 4.2.8 *For a strictly monotonic STS S, $reduce_S$ always terminates after a finite number (n) of steps if applied to a string of finite length.*

$$\forall w \in \Sigma^* : \exists w' \in irreducible_S, n \in \mathbb{N}_0 : w \overset{n}{\Leftarrow} w'$$

Proof 4.2.9 *As every rewrite rule $\alpha \to \beta$ in a strictly monotonic STS satisfies $|\alpha| < |\beta|$, every reduction is decreasing the length of the string. Thus, if the string w_0 is of finite length n, there must exists a finite number of reduction steps $w_0 \Leftarrow w_1 \Leftarrow ... \Leftarrow w_m$ with $w_m \in irreducible_S$ and $m \leq n$.* □

Theorem 4.2.10 *The STS $S = S_{F_{IF}}$ has the following property:*

$$\forall w \in irreducible_S : \forall w', w \overset{*}{\Rightarrow} w' : reduce_S(w') = w$$

We will not give a proof here as this would be overly complicated. For proving this property for a combination of the first four encodings $entity\_ref, dec\_ref, hex\_ref$ and $url\_encoding$, the interested user is instead referred to [31] as these are exhibiting all structural properties necessary for a uniquely parsable grammar. For the first three cases this is due to their distinct start- and ending-symbols (`&` and `;`) making any overlap of reversely applicable items impossible. This spacial detachment eliminates any mutual influences during the reduction process, and therefore allows all reversely applicable items in a string, to be reduced in any order without affecting the result. Even though *url_encoding* does not have an ending-symbol, there is still no overlapping possible since the very restricted character set (hexadecimal digits) after the `%` is not allowing the start-symbol to reoccur.

The *js_single_char* encoding unfortunately is lacking both an ending-symbol and a character-set restriction. If added to the four encodings already discussed, however, still no righthand-side is substrings of any other, and only a single case of overlap is possible: the rewriting rule `\` → `\\` can share its last symbol with other rules righthand-sides of the same encoding. The string `\\a` for example has two overlapping reversely applicable items `\` → `\\` at position 0 and `a` → `\a` at position 1. While both reductions are yielding the exact same result, the reverse application of the latter is somewhat breaking the applicability of the former in the sense that there is still a reversely applicable item at position 0, but a different rewriting rule is used. As the theory of uniquely parsable grammars is concerned about the construction of parse-trees, and applying a different rule changes the labeling of such a tree, it is obviously not applicable here. Since we are only interested in the string resulting from a complete reduction and $reduce_S(w)$ with $S = S_{\{js\_single\_char\}}$, after substituting all special sequences (`\t`, `\n`, etc.), is equivalent to removing all \-characters in w, it is obvious that the order of removal is of no concern either.

The attentive reader probably noticed that Table 4.1 lists two JavaScript string escapings not in F_{IF}. We discovered the *js_hex* and *js_unicode* encodings only very recently and were not able to adjust the argumentation since these two are raising a lot of complications. As both fortunately are only very seldom used in input filters (in fact we have not seen a single example of that), and when used in the original string, do not harm the matching process, we will only give some ideas as to how they could be incorporated into the normalization. The following (until Theorem 4.2.14) thus may be safely skipped.

First, these two encodings are causing a lot of additional substring relationships: `\u` is a substring of all of *js_unicode*'s righthand-sides and `\x` has an equivalent relationship to the *js_hex*-encoding. Note that, according to the ECMAScript Language Specification [9] both escape sequences (`\u` and `\x`) are not allowed. Since some implementations (including SpiderMonkey) nevertheless substitute `\u` by `u` and `\x` by `x`, it is very likely that this kind of escaping has found its way into some input filters. In order to counter this, we not only have to force our reduction process to apply the rule with the longer righthand-side if multiple ones are applicable for a given index, but also have to make sure that there are no other reversely applicable items of other encodings left in the string before reducing JavaScript-encodings. This example will demonstrate why this is necessary:

Example 4.2.11 `\u0072` *can be either reduced by hex_ref to* `\u0072` *and then by js_unicode to* `r` *or by js_single_char to* `u0072` *and then by hex_ref to* `u0072`.

Another problem is that JavaScript string escaping obviously was not meant to be applied multiple times, for this is introducing ambiguity, making it impossible to correctly recover the string:

Example 4.2.12 `\\x784C` *can either be reduced by js_single_char to* `\x784C` *and then another time to* `x4C` *or it can be reduced by hex_ref to* `\x4C` *and further to* `L`.

If we, however, assume that the input filter is not applying JavaScript-encoding to the characters `x`, `u` and all hexadecimal digits (which is very reasonable since none of these has any special meaning in HTML or JavaScript), the only character of a reversely applicable item that can be reencoded would be `\`, which is not causing any ambiguity:

Example 4.2.13 *There is only one way to reduce* `\\x5Cu005Cx76` *to* `v` *by applying JavaScript encodings.*

Note that the assumption only applies to the input-filter. There still can by ambiguous encodings in the original string. These, however, do not cause any problems as long as the reduction process is deterministic and the same for both sides.

As long as the input filter makes no use of these two encodings, the above Theorem thus is very reasonable. And even if it does, the above assumptions, along with some simple precautions, still ensure reduction uniqueness except for few, very improbable situations. We will now continue to construct a normal form suitable for our purposes.

Theorem 4.2.14 *The complete reduction function* $reduce_S(w)$ *for the STS* $S = S_{F_{IF}}$ *satisfies all properties required for the normalization function.*

Proof 4.2.15 *As the existence and uniqueness properties have already been shown in the Theorems 4.2.8 and 4.2.10, we will now proof the* $(*)$*-equation. As already pointed out previously, the following also holds for* $F = F_{IF}$

$$\forall f \in input\_filters_F : \forall x \in \Sigma^* : x' = f(x) \longleftrightarrow x' \overset{*}{\Leftarrow} x \text{ in } S_F$$

as all string have a unique, irreducible representative, the following relation holds

$$x' = f(x) \overset{*}{\Leftarrow} x \overset{*}{\Leftarrow} w \in irreducible_{S_F}$$

and it follows

$$w = reduce_{S_F}(x) = reduce_{S_F}(f(x))$$

□

Our normalization function is partitioning Σ^* into equivalence classes modulo encoding, by assigning each string x a unique, irreducible representative $w = reduce_{S_F}(x)$. Note that, as the equivalence relation must be symmetric, it does not matter in which direction the encodings were applied.

For any string $w \in \Sigma^*$, we can derive two versions $w_1 = f_1(w)$ and $w_2 = f_2(w)$ by applying two different encodings $f_1, f_2 \in input\_filters_F$. Even though encodings have to be applied reversely to derive one from the other ($w_2 = f_2(f_1^{-1}(w_1))$), both still share the same, irreducible representative $reduce_{S_F}(w_1) = reduce_{S_F}(w_2) = reduce_{S_F}(w)$.

The reason why *url_encoding* is a member of F_{IF} therefore is that the parameter string can be URL-encoded an arbitrary number of times, not all of which must necessarily be removed by the input filter (aol.com-parameters for example are URL-encoded up to 4 times). A parameter string p, parts of which have been encoded n times, passed through an input filter removing the first $m < n$ of these encodings and applying m' new ones, yielding p', can still be matched this way.

This normalization thus allows us to completely ignore both the encoding- and decoding step in our input filter model (see Figure 4.2), leaving only the optional removal filter to be dealt with.

4.2.3. String Matching

Representing the equivalence classes by strings has the additional benefit that substring- and subsequence- relations can be specified modulo encoding as well. By applying it to the normalized forms, we are now able to use any pattern-matching-algorithm without concern about encodings. Nevertheless, we still need an algorithm suitable for matching parameters against script code, even if the input filter removed arbitrary character sequences. Before defining our problem formally, we will give a short overview about two classes of string matching algorithms relevant in this context.

Longest Common Substring

The longest common substring problem can be described as given two strings s_1 and s_2, finding a string s, that is a substring of both and longer than any other string with that

property. With a Generalized Suffix Tree (see Subsection 2.5.6), even the more interesting general case of n strings $s_1, s_2, ..., s_n$ can be solved in time linear with their combined length $N = \sum_{i=1}^{n} |s_i|$. Using Ukkonen's Algorithm, a Generalized Suffix Tree for all of them can be constructed in $O(N)$ time [32]. Afterwards, the whole tree has to be traversed, searching for the deepest nodes, representing a substring of all n strings ($O(N)$ time). This has already been exemplified in Subsection 2.5.6. Of course there can be multiple results of equal length.

When traversing the tree, the condition a node has to satisfy can be arbitrarily chosen. Partitioning the strings into two groups (parameters and script codes) and then searching for a string of length $\geq t$ with $t \in \mathbb{N}$ being some threshold, that is substring of at least one string of each group is definitely a lot closer to solving our problem. Rejecting all strings that are substrings of some longer one can be accomplished on-the-fly.

Anyway, this algorithm is not useful to solve our problem, since input-filters can remove character sequences. As in the example in Subsection 4.2.1, sequences, that are known to be removed, can be used to disguise the existence of other keywords. While in the example, this was only applied to the script-tags (which are not part of the code anyway), an adversary can easily apply this technique to the code of his payload, splitting it into an arbitrary number of arbitrarily short strings (possibly only one character each), thus bypassing any filter searching for substrings of some minimum length.

Longest Common Subsequence

A string of length n can have a maximum of $\frac{n^2+n}{2} + 1$ distinct substrings, but a maximum of 2^n distinct subsequences. That may be one of the reasons why algorithms finding the longest common subsequence (LCS) of two strings s_1 and s_2 require $O(|s_1| * |s_2|)$ time [30]. Additionally, to the best of our knowledge, there is no easy way to generalize this concept to find the longest string, being a subsequence of at least 2 from a set of n strings, without matching them pairwise.

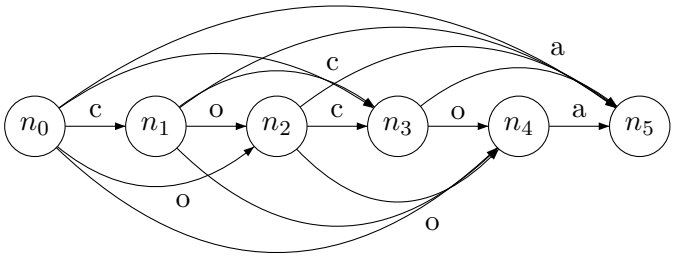
But even if these algorithms were less expensive, LCS-matching would yield a huge number of false-positives, because, given a long enough script code, practically every parameter itself would be a subsequence. However, this type of matching cannot be bypassed by splitting the code. Our problem thus seems to be somewhere in between substring-based and subsequence-based matching. The former is matching not enough while the latter is matching too much.

Subsequence Automata Matching

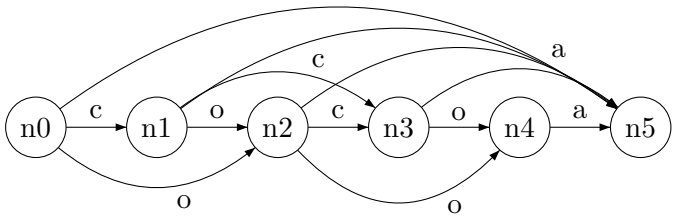
If an input filter is removing character sequences then some characters present in the parameter are missing in the code. While finding common substrings does not allow any removal at all, common subsequence matching allows removal in both directions (or clearer: removal and insertion). As we are not expecting the input filter to insert anything into the parameter, we can finally write our problem as follows:

Problem 4.2.16 *Given a set of strings $T = \{t_1, t_2, ..., t_n\}$ and a set of patterns $P = \{p_1, p_2, ..., p_m\}$ find all strings s, that*

1. $\exists t \in T : s \in substring(t)$
2. $\exists p \in P : s \in subsequence(p)$
3. $|s| \geq k$ *with* $k \in \mathbb{N}$ *being some threshold*



(a)



(b)

Figure 4.3.: Examples for subsequence automata: (a) N_{cocoa} and (b) D_{cocoa}. In both automata, all states are final.

Now that we have defined the problem, we can create an algorithm. In [33] Ricardo A. Baeza-Yates and Gaston H. Gonnet introduced a way to use a Suffix Tree over some text for matching a regular expression in time expected sublinear to the length of the text. They did so by traversing the tree, while simulating a Deterministic Finite Automaton (DFA) along all edge labels. We will thus try to find an automaton accepting the set of all subsequences of a given string. The following NFA can be constructed for any string

Definition 4.2.17 *Given a string* $w = w_1w_2...w_n \in \Sigma^*$*, the NFA* $N_w = (Q, \Sigma, T, s_0, F) \in NFA$ *with*

$$\begin{aligned} Q &= \{s_0, s_1, ..., s_n\} && \text{(states)} \\ T &= \{(s_i, w_j, s_j) | i < j\} \subseteq Q \times \Sigma \times Q && \text{(transitions)} \\ F &= Q && \text{(final states)} \end{aligned}$$

accepts exactly the set subsequences(w)

For every state $s_i \in Q$ and every character $c \in \Sigma$, it is completely sufficient to have one edge to the nearest node s_j with that character c, as every node $s_{j'}$ with $j' > j$ and $w_{j'} = w_j = c$ would just have less edges than s_j, thus unnecessarily limiting the choices. As DFAs can be simulated more efficiently than NFAs, we remove all unnecessary edges to obtain one.

Definition 4.2.18 *Given a string* $w = w_1w_2...w_n \in \Sigma^*$ *the DFA* $D_w = (Q, \Sigma, \delta, s_0, F) \in DFA$ *with*

$$\begin{aligned} Q &= \{s_0, s_1, ..., s_n\} && \text{(states)} \\ \delta &: Q \times \Sigma \mapsto Q : \delta(s_i, w_j) = s_j \leftrightarrow i < j \wedge \left[\nexists j' : i < j' < j \wedge w_j = w_{j'}\right] && \text{(transitions)} \\ F &= Q && \text{(final states)} \end{aligned}$$

accepts exactly the set subsequences(w)

The examples N_{cocoa} and D_{cocoa} are depicted in Figure 4.3.

Putting everything together, the final matching algorithm works as follows:

1. preprocess both code and parameters to remove encodings as discussed in the last Subsection.

2. build a Generalized Suffix Tree for all code strings. With Dan Gusfield's trick from [34, page 116], Ukkonen's Algorithm [32] can be directly used to do this with minimal additional runtime costs. For strings $s_1, s_2, ..., s_n$ of combined length $N = \sum_{i=0}^{n} |s_i|$, this takes $O(N)$ time and produces a Suffix Tree S with $2N - 1$ nodes, N of which are leafs.

3. for every parameter p do

   a) build the subsequence DFA D_p according to the Definition 4.2.18. This is possible in $O(|p|)$ time.

b) traverse S in post-order while simulating D_p along all path as described in [33]. When D_p rejects, backtrack, and when it accepted it's kth character on an edge to node b, continue with one of the three following cases. As a DFA can be simulated in time linear to the length of the accepted string, and during traversal it has to be simulated along all path in the tree up to character k, the worst case time for this step is $O(N * k)$. This, however, is only the case when the DFA is accepting everything, and the tree is of height 1 with N leaf nodes directly connected to the root.

   i. if the match's start index is required, the complete subtree rooted at b has to be traversed for finding all l occurrences of the substring. After adding them to the result-list, the above traversal can backtrack. This takes $O(l)$ time since the subtree has $2l - 1$ nodes from which l are leafs.

   ii. in case both start- and end- indices are required, the DFA simulation must continue along all edges of the subtree rooted at b, until it rejects or a leaf is reached. In both cases, the accepted string must be added to the result-list and the traversal can backtrack. As the simulation does not stop after at most k steps, the above calculation must be corrected in this case. If accepting everything, the automaton would have to be simulated along all path in the tree. In the very same worst-case scenario as above, with the tree additionally representing only a single string, the path-lengths would sum up to $\frac{N^2+N}{2}$ characters, which is a quadratic term. The worst case complexity for the above step would thus rise to $O(N^2)$.

   iii. If no exact matches are necessary, this step can be avoided altogether by preprocessing the tree in $O(N)$ time. If, like in Subsection 2.5.6, every node is added a bitstring representing all strings it's associated substring is contained in, all that has to be done is mark all scripts in b's list as bad.

Checking n script codes $s_1, s_2, ..., s_n$ with combined length $N = \sum_{i=0}^{n} |s_i|$ for occurrences of m parameters $p_1, p_2, ..., p_m$ with combined length $M = \sum_{i=0}^{m} |p_i|$ therefore in the worst case requires $O(m * N * k + M)$ time. Finding the starting indices of all L occurrences requires an additional $O(L)$ time and finding both start- and end-indices is possible in $O(m * N^2 + M)$ time, independent of L.

This algorithm is able to find parameters within the script codes, regardless of any input-filter complying to our model. Additionally, it is not possible to evade this matching by exploiting a removal filter. While the third version (no match data) is completely sufficient for detecting reflected XSS attacks, an ergonomic implementation may also want to supply it's users with at least the match's beginnings and thus use the first version. The second one is only intended for evaluation purposes, due to it's impact on the algorithms complexity. In order to derive statistical match-length-distributions, it is necessary to know the exact length of each match.

The concept presented in this Section is matching query parameters against script codes. It is able to detect reflected XSS attacks unhindered by eventual encodings or removing input filters. Additionally, it can be applied wherever access to unencrypted HTTP traffic is given. As it is not able to detect any stored attacks, though, we developed the second concept presented in the next Section.

4.3. Detecting XSS attacks by learning Web Application Script Profiles

From a victim's point of view, a stored XSS attack is far more difficult to detect. Like any other script, the payload is received as part of a completely normal response without having passed the victim's computer previously. However, we already noticed in Subsection 2.2.2 that Cross Site Scripting attacks do only make sense on dynamic Websites with some kind of session implementation. If a User registers for an account on such sites, he usually is visiting them on a regular basis or at least multiple times. Apart from financially sensitive ones, one can even speak of a proportional relationship between the importance of an authentication credential and the total time spent on the respective site. The mere existence of sites like `www.bugmenot.com`, where people exchange one-time accounts, is convincing proof that not all credentials are considered precious.

Especially sensitive, with respect to XSS, are Websites making extensive use of active content technologies, thus forcing their users to allow JavaScript execution both in their browser and noscript-configuration for that domain. Blocking unwanted scripts, from domains either rarely used or not requiring JavaScript, is already solved by the noscript-plugin [24] in a completely sufficient way. However, an XSS-protection for sites frequently used and extensively relying on client-side scripting currently is completely missing. The rapidly increasing number of highly dynamic sites, the so-called Web 2.0, therefore clearly poses a threat to be mitigated.

A straightforward way for detecting attacks in such applications is to remember repeatedly visited Websites along with all their active contents and to warn about all unknown scripts on successive visits.

One may note, that this type of detector is not classifying found scripts into harmless and malicious ones (for convenience called *good* and *bad* furthermore), but is only recognizing known ones and associating them with their previous occurrences. This, however, enables us to persistently mark scripts, and thus deduce their periliousness based on previously associated data like a user classification.

As this clearly falls into the reign of Anomaly-based Intrusion Detection (see Subsection 2.4.2), the operation of a detector like this should also be divided into a *learning*- and a *protection* phase. While in *learning* phase (or *learning* mode) the detector is not performing any classification, but is considering each and every *unknown* script as good and add them to it's database (which we will call 'Scriptbase') accordingly. If any of these scripts is reencountered, it has to be classified as *known_as_good*.

After some time (usually when no more *unknown* scripts are found), the detector is switched to *protection* mode, now classifying all scripts passed to it as either *unknown*, *known_as_good* or *known_as_bad*. The handling of *unknown* scripts highly depends on the environment the detector is applied in. If a user is available, a good choice would be to provide him with all available information and prompt for a classification decision. In non-interactive environments, *unknown* scripts should be considered *bad* and logged. Whoever is reviewing the log afterwards, should then be provided with some interface to edit the Scriptbase in order to competently correct false classifications and avoid them in future. No matter if immediate or time-delayed feedback is used, the Scriptbase will store the decision along with the script.

After Subsection 4.3.1 clarified the use of some Intrusion Detection terminology, subsequent Subsections will each detail a certain problematic aspect of this concept. Each will

be discussed in detail and a solution will be given. Subsection 4.3.2 establishes a reasonable partitioning of the space of all Webpages, in order to limit the pages a script can reappear on. Subsection 4.3.3 gives an overview about different script types encountered in the data-set and outlines the few problematic ones. Subsections 4.3.4 and 4.3.5 are each presenting a solution for one of them. Finally, Subsection 4.3.6 will propose a number of additional policies that can be applied to further adapt the Scriptbase to a specific Website. These are not part of the original concept as they impose a certain workload on site administrators. The work necessary to tailor a Scriptbase detector to a specific Web Application, however, can be considered much smaller than the setup of a Web Application Firewall as an example.

4.3.1. False-positives and False-negatives

It should be stressed, that our detector's ability to recognize known scripts can also be applied to scripts known as bad. Having a ternary classification, we will have to refine the common notions of 'false-positives' and 'false-negatives':

False-positives are benign scripts belonging to the Web Application, but classified as *unknown*. Only these are counted as false-positives, since they are triggering a user-dialog or a log-entry. Scripts *known_as_bad* are silently disabled.

False-negatives are malicious scripts, that were classified *known_as_good* for some reason.

4.3.2. Scriptbase Granularity

The very first obvious question, arising from the above concept, is "when exactly are two pages the 'same' ?". An equivalence relation is needed, partitioning the space of all possible pages. From a security standpoint, it makes sense to minimize the partition size in order to allow a known script to appear on as few pages as possible (principle of least privileges). Thinking more practical, one would rather demand the opposite, since any too narrow partitioning may yield a lot of false alarms caused by unrelated occurrences of already known scripts on new pages.

In theory, a Webserver could use any information, sent in the request (including cookies, POST-data etc.), to decide which page to answer with. Even data from other sources / random data might be used. Anyone surfing the Internet, however, would instantly agree that URLs are giving a pretty good indication of page equality. Two documents retrieved from the same URL, usually are practically identical.

It would, however, be fatal to include all URL-parts into such an comparison. There are cases, especially among more complex dynamic Websites, where the sessionID is passed as a parameter, as an example. Another argument against including parameters in the comparison is the tendency of users, not to revisit exactly the same URLs. On a news-site, for instance, a user will usually prefer reading the most current news rather than visiting old ones again. While the pages and the scripts on them are identical, except for some textual changes, the URLs typically differ in at least one parameter.

As anchors should also be stripped for obvious reasons, we are left with the scheme-, host-, port- and path-parts of the URL. But, even though the path is commonly used by Webservers to identify which generating program to execute, the cookieless-session implementation of ASP.NET chose to include it's sessionIDs into the path [25] and so-called rewriting engines like UrlRewriting for ASP [26] or mod_rewrite for Apache [27] are allowing even more arbitrary

manipulations. Other common examples are 'access denied' or 'your session timed out'-pages which are sent instead of the requested document and thus can appear under any URL. If these pages (as observed in the data-set), are containing a login-form along with some JavaScript performing quick input validation, every session timeout would trigger a false alarm, if the path was relevant for page equality.

Analysis of the data collected by the crawler yielded that a large percentage of the scripts are appearing in more than one page and some are even appearing in all pages of the respective Web Application. In order to facilitate a fast learning progress and avoid data redundancy, we chose a relation equivalent to the one used in the same-origin-policy: two pages are identical if and only if their URLs share a common scheme, host and port. Host in this context means either 'top- and second-level-domain' (for instance `uni-hamburg.de`) or 'IP-address'. In some cases (`.co.uk`-domains) the third-level-domain should be included as well as specified in [14].

4.3.3. Script Types

As this is the first of several Subsections about script matching, there are some general notions to introduce:

Definition 4.3.1 *A **script** in a dynamic Website is some part of it's generating programs, producing a result that can be interpreted as active content by a browser. We will also refer to this part as the script's **generating function**, although this term may not in all cases be accurate. The result of each execution of this generating function is called an **occurrence** of the script, the set of all possible occurrences it's **occurrence-set**.*

This definition is very fundamental as it captures the detector's main idea. For a browser, a script is just some executable part of a Webpage, completely independent of any other. The Scriptbase, however, is shifting this view by focusing on the relationships between this executable parts, regarding them as instances of a much smaller number of abstract objects, associated with parts of the generating program. The recognition of occurrences as belonging to known scripts is of course rising a big problem. As our tool is lacking any knowledge of the internal workings of the Web Application, we have to find another way to figure out the mapping. Once again, an equivalence relation has to be defined, resembling the original mapping as much as possible. Too small a partition-size would again yield false-positives, while too coarse a relation might fail to detect payloads, forged to resemble existing scripts in the Web Application. We will denote this relation $\simeq$.

Before going into details of the script matching process, we start by giving an overview of the different script-types, encountered in our data-set, and the various features used for classification. This way, the reader may familiarize himself with the problem prior to understanding the solution.

Location

First, we will define the only feature shared by all occurrences:

Definition 4.3.2 *In a parsed HTML DOM-tree, we define the treepath of a node as the concatenation of all edge-labels along it's path from the root, in the order of traversal and separated by `/`. The **treepath** of an occurrence is defined as the treepath of the node it is*

contained in (usually a script-tag). An onload-eventhandler in the body-tag would for instance have `HTML/BODY` *as it's treepath. A script's treepath is also called it's* ***location****.*

The treepath provides a notion of locality within a page. We defined it to capture our impression, that many scripts are always occurring 'in the same place'. Although appearing on very different pages throughout their Web Application, all their occurrences seem to share a common 'location'. To base our notion of locality on the HTML DOM-tree like this, brings the following benefits:

- It is oblivious of insertion or removal of entire subtrees. Even if two documents share only a small part, scripts within this part usually have a stable treepath.
- It is oblivious of data changes. Even though pages, which were generated from database records, usually are structurally identical, they may still exhibit enormous differences in terms of character positions. The tree structure therefore is much more appropriate than any notion based on counting characters or lines from the document's beginning.
- It is oblivious of repeated structures. It is quite common, that a script has an occurrence in each item of a repeated structure. Blog entries, for instance, often contain event-handlers for various tasks. As the items normally are realized as subtrees of a common node, all this scripts share a common treepath.
- It can be efficiently computed. As the browser anyway is building a DOM-tree during the parsing process, and provides us with a DOM-node for each occurrence, extracting treepath locations is not causing any overhead.

Of course, not all scripts are always occurring with the same treepath. An example from the test-set are scripts for decoding email-addresses on the client-side. These obviously must occur wherever there is such an address. We thus clarify:

Definition 4.3.3 *A script is called* ***movable*** *if not all its occurrences share the same location.*

About 168.000 script occurrences found on 32.000 pages of 6 different sites were matching at least one other occurrence in that respective site in type and code. 7.6% of these, however, were not sharing the same location with their counterparts, giving significance to the above definition.

Compared to other features, the location of an occurrence is quite insignificant, since the locations of movable scripts show very few similarities apart from the first (root-most) few hops that are shared by almost all nodes in the tree. Due to this property, locations are not used for mapping occurrences to scripts, but only for the detection of movable scripts in the 'no new movable scripts'-policy (see Subsection 4.3.6).

Type and Event

The Script Finder differentiates four kinds of scripts: external scripts, inline scripts, event handlers and JavaScript-URLs (see Table 4.2). Event handlers, as their name suggests, are always assigned an event on which their code is to be executed. The first two features, that will be part of a formal definition for the $\simeq$-relation, can already be defined:

Script type	Item	Features
external script	ES	location, url
inline script	IS	location, code
event handler	EH	location, code, event
JavaScript URL	JU	location, code

Table 4.2.: type value assignment

Definition 4.3.4 *the **type** and the **event** of an occurrence o are assigned by the script detector according to Table 4.2 as*

$$type(o) \in \{ES, IS, EH, JU\}$$

$$event(o) \in \begin{cases} events_{JS} & if\ type(o) = EH \\ \{nil\} & else \end{cases}$$

with $events_{JS}$ being the set of all JavaScript events (onload, onclick, onmouseover, ...)

There were only very few scripts in the test-set, having occurrences of different types or events, and due to the small number of types/events, it is still possible to learn these as a small number of distinct scripts. Thus, it is reasonable to require all occurrences, belonging to the same script, to have the same type and event.

$$a \simeq b \longrightarrow type(a) = type(b) \wedge event(a) = event(b)$$

Code

Apart from external scripts, all are associated with some string called their code. As this part is about code matching, external scripts will be excluded from this point on. They will be dealt with in Subsection 4.3.5.

All occurrences are dynamically generated by the generating function of their respective script. Movable scripts thus are the result of this function being invoked multiple times (possibly conditionally) within the generating program. However, there is no reason for this function to yield the exact same result on every invocation. In fact, it is a very common technique in Web Application development to pass some information to the client-side by making the generating program write it directly into an occurrence's code.

Definition 4.3.5 *A script is called **dynamic**, if not all its occurrences share the same code.*

Note the similarity of this definition to the notion of dynamic Webpages. We will also refer to non-dynamic scripts as **static**, in future.

It is theoretically possible for two different scripts, to have occurrences with the exact same code. So on the one hand we have scripts generating multiple different codes and/or locations, and on the other hand we have occurrences, possibly belonging to several different scripts both according to their code and location.

So how to determine, whether a specific occurrence is an element of some known script's occurrence-set? For most dynamic scripts, this set is not finite and even if it were, we could

never be sure that every possible element was encountered while in *learning* mode. In order to adequately answer the question, we therefore have to extrapolate the sets.

It must be possible to extrapolate the occurrence-set based on a single occurrence, as it is needed for finding further ones. We will call this first known occurrence the **prototype** of a script.

Another desirable property of this extrapolation would be the preservation of harmlessness: Any occurrence of a harmless script should not possibly do something malicious. As proofing this would clearly require a formal definition of what is 'malicious', we approximate this criterion by stating that the behaviour of no occurrence should significantly differ from those of the prototype.

But what to do if the occurrence-sets are not disjoint and a single occurrence is element of multiple ones? Well, as stated at the beginning of this Subsection, we are searching for an equivalence relation on the set of occurrences. The occurrence-sets are corresponding with the partitions of this relation. And as every equivalence relation has to be symmetric, an occurrence a of a script A, that is also element of the occurrence-set of another script B, automatically means that every occurrence b of B also is a member of A's occurrence-set. If our occurrence-set extrapolation is defined the right way, the partitions thus are either disjoint or identical. If two scripts have an identical occurrence-set, what's the point in distinguishing them? Even if one of them is *known_as_good* and the other one was injected by an adversary, applying the criterion from the last paragraph yields that either the adversary injected a harmless script[1], or there is a malicious script being classified *known_as_good* in the Scriptbase. While the former clearly is of no concern, the latter cannot be solved by the detector itself, as it can only be caused by a malicious script being learned during *learning*-phase, or a user's misclassification. The problem of accidentally learned malicious behaviour is common to all Anomaly-based Intrusion Detection Systems (see Subsection 2.4.2).

In the next Subsection, we will derive a feature for matching dynamic scripts, matching all this requirements.

4.3.4. Dynamic Code Matching

The most important feature for deriving the mapping of occurrences to scripts certainly is what was called the 'code' of an occurrence in the previous Subsection. Although not constant within occurrence-sets of dynamic scripts, it still exhibits a great deal of similarity. Unfortunately, processing it's current form as a string, to detect and measure these similarities, requires a lot of computational power. As pointed out by Graham A. Stephen in [35], for two strings u and v, a lower bound of $\Omega(|u||v|)$ has been proven for any string edit distance calculation. Algorithms solving the related longest-common-subsequence problem, which can be used to compute string edit distances, are used in the Unix diff-tool to derive an optimal alignment of two strings. The general problem with n strings, however, has been proven to be NP-hard by David Maier [29]. Since the quadratic 2-string comparison would have to be performed at least once between the prototype of every known script and every new occurrence, we considered this solution infeasible.

Instead, a JavaScript tokenizer was used to transform the code into a list of tokens. As we had assumed, the purpose of parameter passing usually limited the changes to constants. It does not make sense to change the name of a function or anything about the control flow if

[1] or at least a script behaving very similar to some known benign one

one just wants to pass some information. Hence, we are extrapolating the occurrence-set of any dynamic script as the set of all scripts that can be derived from it's prototype by changing the constants. A matching algorithm allowing this can easily be derived by substituting all tokens representing constants (Strings, Numbers and Regular Expressions) in the tokenlists by singleton-tokens representing their classes (:string, :number and :regexp), prior to comparison.

This criterion, defined on tokenlists, is a simplification of what Zhendong Su and Gary Wassermann [17] defined on parse trees to detect command injection attacks. However, since we do not exactly know which parts of a dynamic script are static and which are dynamic, using a full parser and applying the criterion from [17] is not an option.

As it will be a very important part of the final relation, the properties of this comparison should be discussed in detail: It is clearly an equivalence relation and the occurrence-set can be easily derived from every single occurrence of a script. But does it preserve harmlessness? Or better: how much can the behaviour of a script be changed by just manipulating its constants?

Certainly, the greatest influence can be achieved by altering a constant string that is passed to the eval()- or document.write()-functions. Although this clearly falls into the realm of type0-XSS, it's impact can be severely limited by applying the 'restricted dynamic code matching'-policy (see Subsection 4.3.6). See Chapter 8 for more information on how this issue could be resolved in future.

Another noteworthy variant is manipulating constants in conditionals. As many if-clauses or loop-conditionals compare a variable against a constant, the control-flow of such a script can be manipulated by altering this constant. It is thus often possible to alter if and how often something is done. The worst-case scenario of infinite repetition, however, is severely limited due to the runtime-restriction in all modern JavaScript interpreters.

Data of course can be most easily manipulated: If a script is searching for a specific name in a list or sending data to some URL, the name and the URL can obviously be manipulated arbitrarily. Again, all this can be severely restricted using the 'restricted dynamic code matching'-policy. Constants in conditionals are usually not changing dynamically and therefore can be protected, and even if URLs are changing dynamically, it is very common that at least their first part (protocol, host, port) stays constant among all occurrences.

After discussing all this possible manipulations, it is important to note that the functional components of a script can not be altered. It is not possible to make a script read the cookies and send their contents to some URL, unless the script already did exactly that, thus containing all the necessary functional components. Since it is very unlikely for a Web Application to contain such a script (it is anyway receiving the cookie on every request) the chances of an adversary to perform this type of attack are very limited. This very same argument holds not only for credential stealing-, but also for browser hijacking-, in-browser-propagation- and LAN-scanning attacks. In fact, an adversary is not even able to perform an advanced request forgery with POST-requests unless there is a script manipulating and submitting a form to an arbitrary URL. The operating range of a malicious payload thus is very likely limited to content defacement, denial-of-service and simple request forgery, all of which are also possible when injecting non-active content as discussed in Section 2.2 (see Table 4.3).

Definition 4.3.6 *For every occurrence o, the* ***code*** *and the* ***tokens*** *of o are assigned by the script detector as*

$$code(o) \in \Sigma^* \; with \; type(o) = ES \longrightarrow code(o) = \epsilon$$

content defacement	`document.write("foobar")`	`foobar`
denial of service	`while(true){}`	`<!--`
simple request forgery	`document.location = 'somewhere.org';`	`<iframe src='somewhere.org'> </iframe>`

Table 4.3.: payloads for performing simple attacks with or without active content

$$tokens(o) \in tokens^*_{JS}$$

$$tokens(o) = substitute(tokenize_{JS}(code(o)))$$

with $tokens_{JS}$ *being the set of all possible JavaScript tokens,* $tokenize_{JS} : \Sigma^* \mapsto tokens^*_{JS}$ *being a JavaScript tokenizer and* $substitute : tokens^*_{JS} \mapsto tokens^*_{JS}$ *a function substituting all string tokens by :string, all numerical tokens by :number and all regular expression tokens by :regexp.*

We finally demand, that

$$a \simeq b \rightarrow tokens(a) = tokens(b)$$

4.3.5. External Scripts

Since external scripts do not have any code to compare, they have to be dealt with differently. A simple approach is to just remember their URL. There are, however, dynamic cases even among external scripts. Flickr.com, for example, often uses the script's filename to pass along a number of parameters. However, in general it can be observed that this "dynamic URLs" are in most cases only used within the own domain.

All external scripts, sharing the Website's domain, can be considered safe (note that this also includes relative URLs), because an attacker, able to add arbitrary JavaScript-files to the attacked domain, usually is also able to spawn new HTML-pages and thus has complete control over their contents and no need for techniques like XSS. Our approach is to classify all external scripts, who's URL satisfies the same-origin-policy, as *known_as_good* and remember the URLs of the rest.

Definition 4.3.7 *For every occurrence o the* ***url*** *is assigned by the script detector as*

$$url(o) \in \Sigma^* \; with \; type(o) \neq ES \rightarrow url(o) = \epsilon$$

Keeping in mind that scripts of the same domain were already filtered out, we can add

$$a \simeq b \rightarrow url(a) = url(b)$$

and finally, we can define our equivalence relation as

$$\begin{aligned} a \simeq b \leftrightarrow type(a) &= type(b) \wedge \\ event(a) &= event(b) \wedge \\ tokens(a) &= tokens(b) \wedge \\ url(a) &= url(b) \end{aligned}$$

This relation is partitioning the space of all possible script occurrences in a way, that each script's occurrence-set is very likely to be a subset of some partition, without introducing troublesome ambiguities. Of course this relation is based on a number of assumptions, whose compliance needs to be carefully evaluated.

4.3.6. Policies

When applying the above concept, an adversary is still able to inject an arbitrary number of scripts anywhere into any page of the Web Application, without being noticed, as long as they are equal to some native script of that application with only constants altered. Provided with enough (or better: with the right) native scripts to choose from (many modern Websites contain several thousand distinct scripts), it is theoretically possible that the adversary could be able to combine several such scripts to perform some kind of malicious action.

In order to mitigate this risk, there are several easy-to-implement policies applicable in *protection* mode. They are meant as additions to the above concept. Each of them may increase the number of false-positives on some sites.

no new movable scripts When the *learning* phase is over, do not allow any non-movable scripts to become movable any more. This policy is limiting the number of scripts available to the adversary, since he usually is not able to inject a non-movable script into the correct location.

restricted dynamic code matching For each constant in each dynamic script, remember if it matched multiple values during *learning* phase or was constant among all known occurrences. Do not allow such constants to match other values in *protection* mode. A lot of restrictions may even be applied to those values not constant: length- or character set restrictions are two examples. This policy is limiting the adversary's freedom in manipulating the scripts he can choose from, increasing the difficulty of altering their behaviour.

restricted number of occurrences For each script remember whether it occurred more then once in any page encountered during *learning* phase. Do not allow single-occurrence-scripts to occur multiple times in a page. This can be crucial, since many scripts rely on other objects on the same page (a form, for example). Injecting the script into some page without this object does not make sense, since it would only cause an error. This policy effectively prohibits the injection into a page containing the needed object, since the script usually is already present on such pages. It is not decidable which of the two (or more) scripts is the attack. Thus, for applying this policy in interactive environments an additional user-interface is needed, asking the user not only to decide whether the script is good/bad, but also which one to execute. In non-interactive settings, this is much less problematic.

A second concept was presented in this Section. It specifies an Anomaly-based IDS on HTTP traffic, which learns a script-profile (called Scriptbase) for each Web Application and based on that is able to detect both reflected and stored XSS attacks. Like any other Anomaly-based IDS, it is not able to guarantee absolute detection accuracy, but it is tailored to allow only a minimal number of false-negatives connected with script resemblance and DOM-based attacks, which anyway cannot be observed in HTTP traffic. In order to mitigate this risk,

a number of optional policies were presented, which can be used to adapt the detector to a specific Web Application.

The next Sections will outline two application scenarios in which the detector could be applied.

4.4. Application 1: Server-side Intrusion Detection System

The pages learned during the *learning* phase, are not required to belong to a single browsing session. They only have to cover as much of the Web Application as possible. For obvious reasons, this can be achieved much easier and faster on the server-side where request/response pairs from hundreds of users, surfing the site simultaneously, are available for learning. After a sufficiently long learning period, the detector can be switched to *protection* mode and serve as an IDS sensor for XSS incidents.

Every *unknown* script found, should be considered bad and logged. All scripts classified as *known_as_bad* are mere repetitions of these and therefore should only trigger warnings.

Since a site's administrator usually knows when his Web Application is changed (either because he is part of the development team, or because he knows when updates are applied), it is quite easy to switch the detector back to *learning* mode in such situations, in order to prevent a flood of false-alarms.

Another benefit of applying the detector on the server-side is that there is only one site to learn, allowing the use of very restrictive policies (see Subsection 4.3.6), which can be trimmed to fit this particular site much more precise.

One could also use the detector as an output filtering mechanism, but due to the overhead induced by parsing each page with multiple browsers, this would cause a significant increase in the site's response time.

If false-positives continue to occur in certain pages, despite a sufficiently long learning period, this can also be used as an indication that this certain part of the Website seems not to be used very frequently.

If the String Matcher is applied to the output of a Web Application, it can detect reflected XSS attacks and help to find and correct this type of vulnerability in the application. Independent of the detection technique, however, this application theoretically allows a certain amount of users to be affected, before the vulnerability can be fixed. As every attack vector needs to be found, explored and tested prior to exploitation, practical cases would still give a Web Application's administrator a realistic chance to fix the vulnerability in time, though.

4.5. Application 2: Client-side Browser Enhancement

If the detection algorithm is implemented with performance in mind, and a browser is either open-source or provides a sufficient interface, it is possible to enhance the browser, enabling it to learn the Websites it is displaying, and check each script in a page for legitimacy while the page is loaded. Each Website would start in the *learning* phase and would be switched to *protection* mode automatically when the site's Scriptbase ceases to change. In *protection* mode, the user would be prompted for each *unknown* script found, to decide whether this is a false alarm or an attack. Of course, he should be supplied with sufficient information to competently decide. All scripts classified as *known_as_bad* should be silently disabled.

An important point to notice is, that most browsers (including Firefox) are internally working asynchronously and that therefore receiving, tokenizing, parsing, script execution, layouting and rendering of the page are not subsequent steps, but executed intermixed while new data arrives. This is the reason, why the composition of a slowly loading Webpage can actually be observed on the screen. One can think of it as a pipeline that every piece of received data has to pass through.

Since JavaScripts are executed when parts of the page are already rendered, and while other parts are still being received and parsed, it is not possible to load and parse the complete page, then reason about the scripts, disable the malicious ones, and pass it on to the layout engine. One could, however, first load the page in some hidden window with active content execution disabled, then apply the detector, remove all malicious scripts, and reload the altered page into the browser window. As this solution would, in the eyes of the user, significantly reduce the Web Application's responsiveness, a much better approach is to integrate the detector into the pipeline. The decision about a script's admittance therefore has to be taken immediately on its occurrence in the parser. At this point, it's execution can be prevented quite easily by just refusing to call the JavaScript-engine or register the event-handler. The script furthermore will be handled as a normal HTML-tag and parsing continues.

As the Scriptbase anyway handles the scripts one after another, it is well-suited for this approach. The String Matcher, however, is usually building a Generalized Suffix Tree from all scripts before iteratively matching each parameter against it. Adapting it thus is inevitable.

When the results of each parameter-match are stored within the tree nodes, it is possible to iteratively add the strings to the tree, each time only continuing the matching process in the newly added areas. While this would certainly increase the memory required to hold the entire tree, the time complexity of the algorithm should not increase significantly.

Unfortunately, there are other, currently unsolved, issues with this application scenario making it rather impractical for the average user. More information can be found in Chapter 8.

5. Implementation

"Security risks can never be totally eliminated, only managed."
–Prof. Dr. Joachim Posegga

This Chapter will address the implementation of an XSS detector according to the concepts discussed previously. All three parts are to be implemented, along with an evaluation environment to test them. The detector itself is split into two independent parts: the Script Finder and the Model. While the Script Finder is corresponding to the Firefox modification announced in Section 4.1, the Model subsumes both String Matcher (see Section 4.2) and Scriptbase (see Section 4.3). Of course, the implementations can be separated and are evaluated independently.

Each part will be discussed in a subsequent Section on it's own. Section 5.1 will detail the Script Finder's implementation, Section 5.2.1 presents a prototype of the String Matcher and Section 5.2.2 discusses the structure of an efficient Scriptbase IDS. The evaluation environments are afterwards discussed in Section 5.3.

5.1. Script Finder

As explained in Section 4.1, the principal task of finding scripts in HTML pages is performed by the browser's parser. The component we call Script Finder is an extension of Mozilla Firefox, providing the necessary interface to use it this way. Firefox is build highly modular and offers a wide range of plugin interfaces (see Section 2.3). We used this existing infrastructure by implementing a large part of our extension as a plugin, only making changes to Mozilla's codebase where absolutely necessary, ensuring our modification to be easily portable to future releases.

The basic idea is to make Firefox offer a network-based service, accepting HTML documents and returning script lists. All logic necessary could be implemented in an XPCOM component, called the 'Script Manager' (Subsection 5.1.2). Practical problems, however, forced a XUL-Overlay to be added (Subsection 5.1.3) and convenience demanded a ruby-library offering object-oriented access to the Script Finder's functionality (Subsection 5.1.4).

Firefox's HTML parser was the only component not offering a sufficient interface. The changes made to it's code are discussed initially in Subsection 5.1.1.

5.1.1. Patch

First, we will discuss the changes in Mozilla's codebase, necessary to use Firefox for detecting scripts.

In [18], Jim, Swamy and Hicks shortly explained their modification to the Konqueror browser, making it detect all scripts executed in a Webpage. They hooked all calls to the

JavaScript engine and were automatically informed about every script executed. For several reasons this method is not applicable to Mozilla Firefox:

- As mentioned in Section 2.3, a large part of Firefox itself is written in JavaScript (about 22% according to `ohloh.net`). The XUL-GUI embeds a large number of small scripts and nearly all available plugins are adding some code via the overlay technique. Additionally, it is possible to implement entire XPCOM components in JavaScript. All this so-called privileged code is also executed by the SpiderMonkey engine and has to be distinguished from the sandboxed client-side scripts embedded in HTML pages.

- As we only want to detect scripts already present in the page, we have to disable JavaScript execution (see Section 4.1). In order to save resources, Firefox does not even bother to parse the scripts when JavaScript is disabled. Thus, the scripts we are looking for are never passed to SpiderMonkey at all.

- While in [18], it was completely sufficient to know about a script prior to its execution, our goal is to find all scripts in a page, not just the executed ones. In Firefox, both event-handlers and JavaScript-URLs are only parsed and executed when the event was triggered / the link was clicked, making script execution hooks not sufficient for our purpose.

A much better approach is to place the hooks directly in the parser. This way, they are executed every time a script is found in an HTML document, independent of whether it will be executed or not. We were able to identify the appropriate locations to hook script-tag parsing. As Event Handlers are not executed directly but registered for a certain event, we hooked the registration process instead. No way for detecting JavaScript-URLs could be found in time. These are treated as normal URLs throughout the parsing process and are executed within a protocol-handler component, assigned the task of handling URLs for the protocol 'javascript', while the browser tries to resolve the location it is directed to by a user's click.

It is also possible to embed JavaScript into other inlined content like SVG, for example. For detecting such scripts, it is necessary to modify the parsers for all script-enabled content-types in the very same way as our prototypic implementation did with the one for HTML.

Once the hook-routines are called, they are using a component called 'Observer-Service' to pass their collected information on to all observers registered for the topics 'event-handler parsed' or 'script-tag parsed'. The information passed is sufficient to locate the respective script in the DOM-Tree.

5.1.2. Script Manager

The Script Manager is a XPCOM component implementing a server running within the Firefox browser and providing an HTML-parsing service to anyone connecting to its socket. Implementing this in JavaScript was possible since the components code is privileged (see Section 2.3), and therefore granted access to capabilities like the ServerSocket.

When a URL is passed to the socket, a new tab is opened in the topmost browser window using the `tabbrowser.addtab` method. While the page is loaded, the patched browser's parser passes all script-parsing events to the Manager, where they are collected, filtered and converted. Unfortunately, the Firefox browser handles page loading asynchronously and

therefore returns from all loading calls (including `addtab`) instantly. For figuring out when exactly a page has finished loading, one has to register an event listener with the appcontent components of the respective browser window. As this can be established much easier from within a XUL overlay, that is anyway loaded into every browser window, we moved this part into its own component called the Notifier.

The Script Manager is also responsible for preventing Firefox from loading any external content required by the loaded document. This was implemented using the *nsIContentPolicy* interface (which is described in [36]).

Once the loading process is finished, all data is properly formatted and sent to the client side. The tab is removed.

5.1.3. Notifier

The Notifier is a small XUL Overlay embedding some JavaScript to notify the Script Manager about onload events in any of the browsers tabs. These are observed by registering proper event handlers in all appcontent-parts of the browsers XUL interface. For communication with the Script Manager, we defined our own XPIDL-interface named *nsIScriptManager*.

5.1.4. Client Library

Finally, a ruby library was added, encapsulating the actions of connecting to the server, passing the HTML content, receiving an answer and parsing the results into object-oriented components that can be conveniently used in the ruby environment. When a patched Firefox browser is running with the detector component installed, a single ruby instruction is able to find all JavaScripts in an HTML-document. The library was named Client Library as it implements the client-side of a small protocol for communicating with a Script Finder instance. This way, the Model discussed in the next Section is accessing the Script Finder's abilities.

The Script Finder component, whose implementation was outlined in this Section, is enabling the Model to use Mozilla Firefox as a script detector. The information necessary was directly extracted from Firefox's parser and passed to a ScriptManager component through XPCOM. For synchronization, it was necessary to additionally introduce a XUL overlay. Following the script's path we will now discuss the Model's implementation in the next Section.

5.2. Model

Focusing on rapid prototyping abilities to test different algorithms rather than speed, the Model was written in ruby and uses the Script Finder via the Client Library. Apart from a small abstraction layer making it usable both in IDS- and browser-mode, it is consisting of two submodels, implementing the two concepts introduced in Sections 4.2 and 4.3 respectively. Each component will be discussed in a subsequent Subsection. The String Matcher is outlined in Subsection 5.2.1 and the Scriptbase in Subsection 5.2.2.

5.2.1. String Matcher

Implementing the concept developed in Section 4.2, the String Matcher is checking script codes and parameters for similarities in order to detect reflected XSS attacks.

From the three versions of the algorithm discussed in Section 4.2, two were implemented: The first version, finding the start-indices of all matches and running in $O(mNk + M + L)$ time, was used for detection and the second version, additionally finding the end-indices, but running in $O(mN^2 + M)$ time, was required for evaluation. It is useful for creating statistical match-length distributions, as the actual match-length can be computed by subtracting start- from end-index.

In order for the algorithms to work a number of components were required:

Normalizer

While in Subsection 4.2.1, normalization of a string was defined as its complete reduction in a Semi-Thue-System, Theorem 4.2.10 states that the order of reduction steps is unimportant. When grouping the reduction steps by their encoding functions, we thus get a series of decoding steps (reverse applications of $f \in input\_filters_F$ with $F = F_{IF}^{\Sigma}$). The Normalizer hence can simply be implemented as a loop applying all five decoding functions until the string ceases to change. While URL-decoding is part of ruby's standard library (`CGI.unescape()`) and all three HTML-decodings were available in the htmlentities library [41], the *js_single_char* encoding was implemented by hand.

Generalized Suffix tree

Our Generalized Suffix Tree implementation was directly derived from [32]. Leaf- and inner-nodes are not distinguished. Instead, a list of start indices is stored in each node that is representing some suffix's end. It is therefore not necessary to add an empty leaf-node if a suffix is prefix of another. Note that the lists are necessary as a suffix can occur in multiple strings.

Subsequence Automaton

A rather simple DFA implementation able to store / restore the current state, and creating the DFA from a string according to Definition 4.2.18.

5.2.2. Scriptbase

The Scriptbase keeps track of all scripts found in a Web Application. For each document, it is passed a list of scripts found in it, the respective URL, and a callback function. The Scriptbase was implemented using several hashtables, containing the already known scripts. Each script in the list is passed to a ternary classifier categorizing it as *known_as_good*, *known_as_bad* or *unknown*, based on the hashes. Every *unknown* script is furthermore passed to the callback function for a decision if it is good or bad. It is then added to the hash tables accordingly, and will be classified as either *known_as_good* or *known_as_bad* in future. In Browser-mode, the callback function is expected to prompt the user for a decision, and in IDS-mode to classifies all scripts as good while in *learning* phase and as bad otherwise, resulting in a behaviour typical for Anomaly-based Intrusion Detection Sensors (see 2.4). In the end, a list of bad scripts is returned, including all scripts either classified as *known_as_bad* or rated as bad by the callback function.

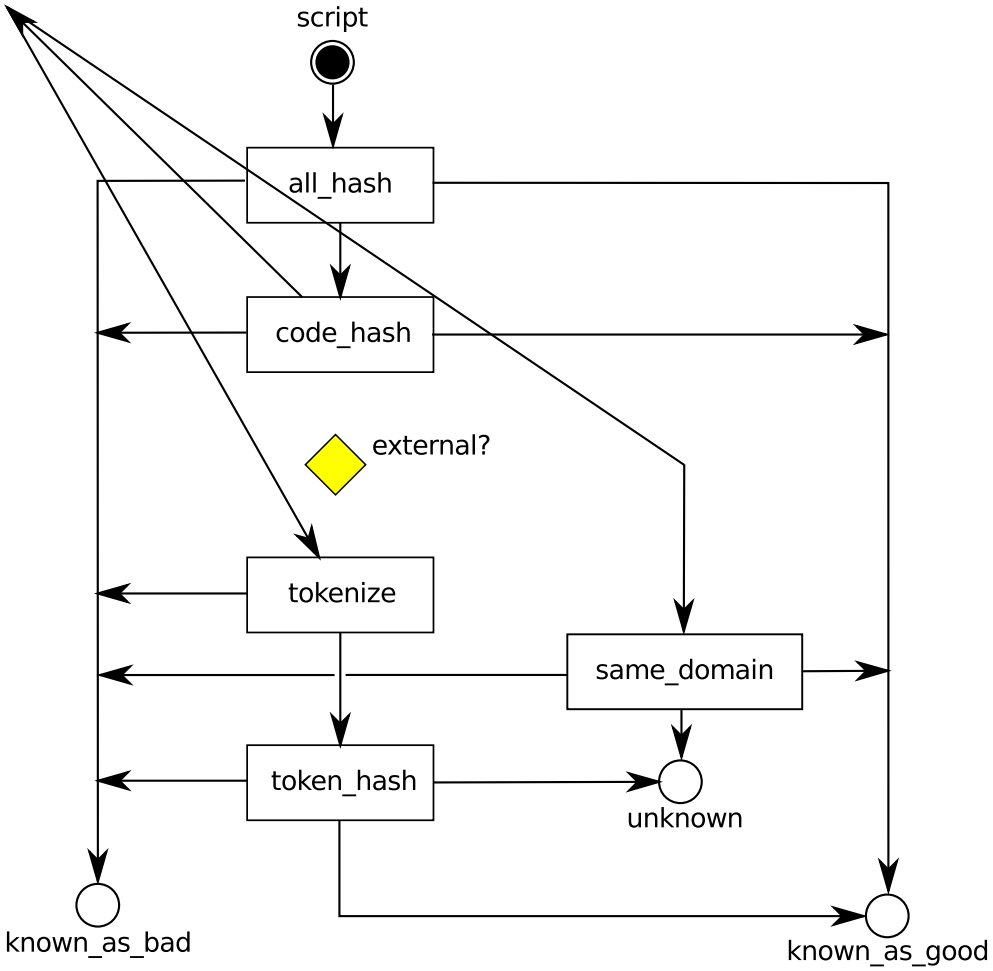


Figure 5.1.: schematic diagram of the classifier

Classifier:

Figure 5.1 schematically displays the Classifier. The scripts are passed through a number of filters each rating them either *known_as_bad* or *known_as_good* or passing them on to the next one. The 'tokenize' step as a notable exception invokes a JavaScript tokenizer, to perform the necessary tokenization required for the token_hash. The reason why it is also able to 'classify' scripts as bad is its error-handling routine. If a script is not tokenizable, it is treated as *known_as_bad* and therefore silently disabled. The same applies for the handling of parsing errors in general. If the parsing of a URL fails during the same_domain step, the script is also regarded as *known_as_bad*.

The token_hash and same_domain procedures are implementing the classifications outlined in Subsection 4.3.3 and 4.3.5 respectively. After a script has been tokenized, the tokens are substituted by unique meta-tokes representing their keywords. JavaScript terminal tokens (Strings, Numbers and Regular expression) are substituted by meta-tokens (`:string`, `:number` and `:regexp`), representing their respective class. A token representing a certain left bracket is for instance substituted by a token representing all left brackets and a token representing the string "hello" is substituted by the `:string` token representing all strings. Using a hashtable, this substitution can be accomplished in time nearly linear with the number of tokens and simplifies dynamic script matching to comparing tokenlists, which can also be very efficiently implemented with a hashtable since all tokens are unique representations. As the token_hash maps lists of meta-tokens to scripts, the filter is reduced to a simple hashtable lookup operation.

Being only passed external scripts, the same_domain filter does nothing more than parsing their URLs and classifying them according to the same origin policy. If the host-part has an equal IP-address as or shares the last two domain parts with the documents URL, the script is classified as *known_as_good* or as *unknown* otherwise. As no domains in the test-set required any special treatment, we safely could ignore the CIAC recommendations [14].

While the part covered so far can be considered the matching policy, the remains serve as optimization. Tokenizing a large script is a rather expensive operation and should be avoided as often as possible. For minimizing the burden of processing repeatedly occurring scripts, the all_hash as the first filter is mapping a string containing all features of a known script to a boolean value. If a script's feature-string is a key in this hash, it means the script has already been encountered with the value indicating whether it was good or bad. Both purpose and structure of the code-hash are very similar, the only difference being that the tree-path is not included in the feature string. It therefore filters all scripts hat already occurred at a different location. This property is useful for marking scripts as 'movable' and thus required for implementing the 'no new movable scripts'-policy. As indicated by their names, both filters can also be very efficiently implemented with a hashtable.

For splitting the script codes into token lists, we used a slightly modified version[1] of rb-Narcissus [28], a ruby port of Mozilla's JavaScript parser, provided by Paul Sowden.

The Model whose implementation was described in this Section subsumes both concepts from Chapter 4 and therefore can be seen as an implementation of what was called the 'XSS detector' before. The next Section will discuss two evaluation environments used to test both concepts and implementations.

[1] see appendix A for details

5.3. Evaluation Environment

In order to properly evaluate both concept and implementation, several additional components were required, forming an environment capable of simulating the expected application scenario. We will discuss each component separately in a subsequent Subsection.

5.3.1. String Matcher Evaluation

For evaluating the String Matcher, we used the evaluation-version or the algorithm, mentioned in Subsection 5.2.1, to analyse all documents in our data-set. The search was restricted to all string matches with length > 3, in order to limit the number of results. From these results, a match-length distribution was derived and all results of length > 15 were logged as 'false-positives'. No 'malicious' scripts were injected in this first run, in order to prevent the injected parameter from yielding matches with other scripts as well.

In a second run, we used the normal version of the algorithm with threshold $k = 15$ to analyse several Web Applications and on each page added a parameter containing some part of a script in the page with length > 15. If the match was not found, a false-negative was logged.

5.3.2. Scriptbase Evaluation

Evaluating the capabilities of the Scriptbase requires training it in *learning*-mode with a variable number of documents from a certain Web Application, then switching to *protection*-mode, and presenting it a large number of other documents from that same application, containing a lot of stored XSS attacks. We are therefore splitting each Web Application's document set into a training- and a test-set of equal size. As the Scriptbase never gets in touch with the HTML documents directly, it is sufficient for evaluation purposes to let the Script Finder parse every document and inject the attacks into the script-lists afterwards. After the Scriptbase has been trained with some percentage of the training set, each script-list of the test-set is being primed with 0 to 5 'malicious' scripts, injected into arbitrary positions, before it is passed to the Scriptbase. These scripts have randomly generated locations within the document (treepath) and their code is resembling those of typical payloads like DoS-Attacks or Credential Stealing, with at least one part being randomly generated. Event-handlers are assigned an event randomly drawn from a list, and the URL of external scripts is also partly randomized.

When the Model is processing scripts from the test-set, a list of scripts passed to the callback function is produced. It is called the `bad_list` and contains all scripts that were classified as *unknown*. The return value is called the `disabled_list` and additionally contains all scripts *known_as_bad*. Each script in the `bad_list` that has not been injected is considered a false-positive. Every injected script not in the `disabled_list` is counted as a false-negative.

The evaluation itself is based on the large data-set, whose collection is described in Chapter 3, and will be covered in detail in the next Chapter.

6. Evaluation

"Welcome to the real world!"
–Morpheus, The Matrix

After detailing our concept as well as various aspects of its implementation, we will now shed some light on how well everything is suited for practical use. A number of minor shortcomings will be discovered throughout the course of this Chapter. In the end we will sum them up along with a theoretical discussion of current limitations before Chapter 8 will provide ideas and concepts to resolve them.

Section 6.1 is testing if our Script Finder implementation from Section 5.1 is able to find all kinds of scripts. Section 6.2 then is checking the prototypic String Matcher for false-negatives, and determines realistic false-positive rates by deriving match-length distributions from the data-set. In Section 6.3, both false-positives and false-negatives are determined by letting our implementation learn the data-set. Concluding, Section 6.4 will give the theoretical discussion mentioned above.

6.1. Script Finder Evaluation

For testing the Script Finder's detection ability, we needed several payloads utilizing filter evasion techniques. The usual XSS disclosure lists turned out to be insufficient in this case, as all disclosed attack vectors were trivially detectable. In fact, most of them apparently did not even have to evade any filters, while the remaining applied the method exemplified in Subsection 4.2.1 for bypassing removal filters, which leaves no traces in the injected payload. We thus gathered a number of non-trivial test-samples from the XSS Cheat Sheet [40], a rather comprehensive list of known filter evasion techniques.

The browser-dependence of our detection method has already been discussed in Section 4.1. As Firefox obviously is not able to detect any payload it - under normal circumstances - would not execute, we limited our search to techniques known to work for this platform and tested each of them on a patched Mozilla Firefox 2.0 browser as well as an unpatched installation of Iceweasel 2.0.0.3.

From 25 attack vectors gathered, 5 did not work on our system. No discrepancy was found between the two browser versions. From the remaining 20, 5 were not detected by the Script Finder. The reasons for this are twofold:

JavaScript URLs

Three vectors were variants of JavaScript URLs. As stated in Section 5.1 we were not able to implement the appropriate parser hooks in time. This is not a conceptual problem but can be solved with some time and effort.

Embedded Content

The two remaining undetected vectors were embedding some other content encapsulating the payload. In this cases CSS, XUL and SVG were used. This problem was also stated in Section 5.1 and constitutes another resolvable implementation issue.

A complete list of all 20 payloads used, can be found in Appendix B. The 5 false-negatives are marked in red.

6.2. String Matcher Evaluation

The String Matcher was evaluated in two steps. Step 1 used the version of the matching algorithm, providing both start- and end-index of each match (see Subsection 5.2.1), to produce match-length distributions for all Web Applications in the data-set. The second step used a threshold of $k = 15$ to test the detection capability of the normal String Matcher version on a subset of these Web Applications, by letting it detect obfuscated script parts. As expected, no false-negatives were found in the second step. We will thus focus on discussing the results of the first step in this Section.

In a very early stage of evaluation, it was noticed that including the referrer-header or cookies into the parameter-set yielded a very large number of false-positives. In case of the referrer, this was due to the large number of URLs in the documents, which for obvious reasons were each sharing a considerable prefix with the referrer. The cookie-values did exhibit a completely different problem, though. As it also emerged with certain URL-parameters, we will postpone it's discussion to the general problems list. We opted not to include both values into the set of parameters, since we have never heard of any XSS attacks using the referrer, and cookie-XSS is very rare and requires a previous successful attack for exploitation.

Based on their match-length distributions, the Web Applications were grouped into 3 disjoint classes: While members of the first one (class A) barely showed any matches at all (see Figure 6.2), the algorithm was able to find quite long matches in class B on a regular basis (Figure 6.3). All matches with length ≥ 15 characters, however, were dynamic string matches as explained below. Members of Class C were additionally exhibiting other matching types of considerable length (Figure 6.4). We will discuss the reasons for such long matches one by one:

Dynamic string match

This very common kind of similarity is caused by a dynamic script, being passed the contents of a parameter using the method discussed in Subsection 4.3.3. Dynamic string matches are readily identifiable, as the script's substring in always part of a string token. This is because the passed data has to be interpreted as a constant by the JavaScript interpreter. Due to this property, it is quite easy to filter these matches using a tokenizer.

Keyword match

In some rare cases it was found that a URL-parameter's name or value was coinciding with the name of a JavaScript function. Of course this type of match is also possible for variable

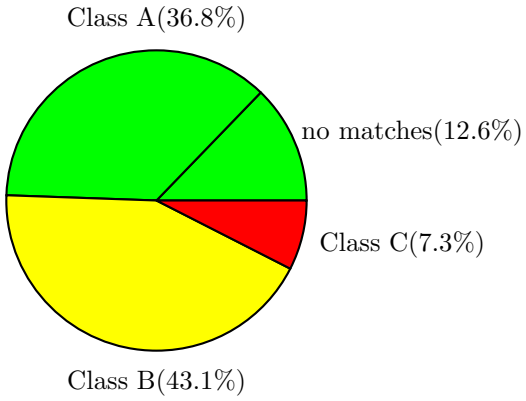


Figure 6.1.: Relative class percentages of String Matcher evaluation.

names or JavaScript keywords. As most of these names are usually quite short, the function-name-case seems to be the only one causing problems. This case can also be handled by using a tokenizer and requiring a match to span multiple tokens in order to cause an alarm.

Whitespace match

In one Web Application, some scripts were containing > 15 consecutive whitespace characters. Whenever a parameter consisted of several words with whitespaces between them, the algorithm produced a match. This case is somewhat rare, as usually tabulator characters are used for indentation, but can also be easily filtered by ignoring 'between-token' matches.

Long parameter match

The number of subsequences of a string grows exponentially with it's length. The probability of a coincidental match thus grows with the parameter length. In our data-set, it was observed that some very large parameters (> 400 characters) were from time to time matching random script parts of length > 15. For this reason, cookies (which can be up to 4kB in size) are causing such a tremendous amount of matches.

Statistics

Relative percentages of the classes are depicted in Figure 6.1 and for each class a graph of it's match-length distribution is shown in 6.2, 6.3 and 6.4, respectively. For the Web Applications in class A, the String Matcher is not causing any false-positives and can thus already be used. Since dynamic string matches can be filtered easily, an additional tokenization- and filter-step should prepare it for use on class B members. Some Ideas for dealing with class C applications can be found in Chapter 8.

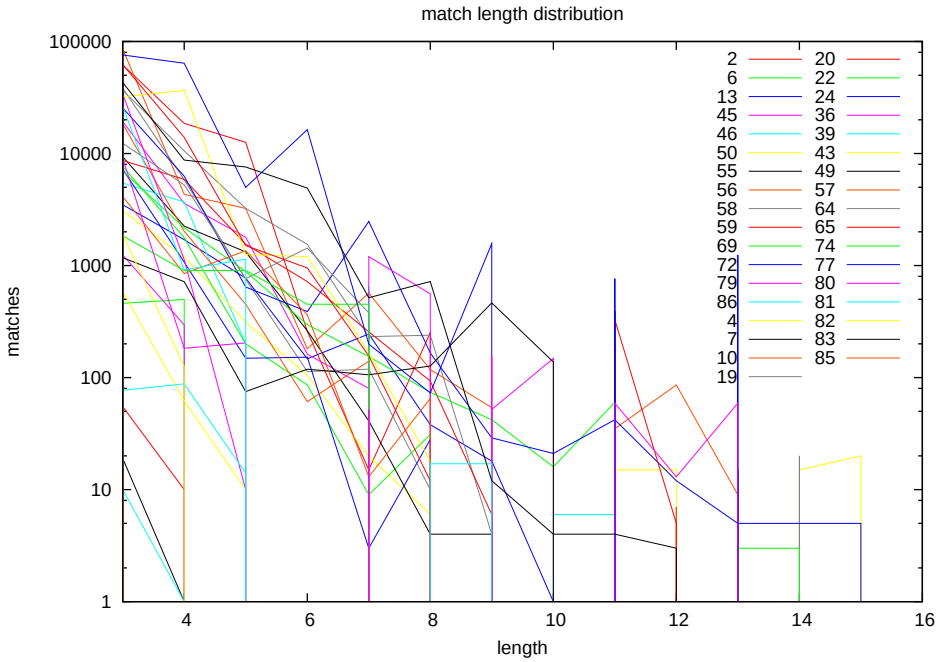


Figure 6.2.: Matchlength distribution of all members of class A in the String Matcher evaluation. Note the ordinate is scaled logarithmic. The numbers are referring to the respective entries in the list given in Appendix C.

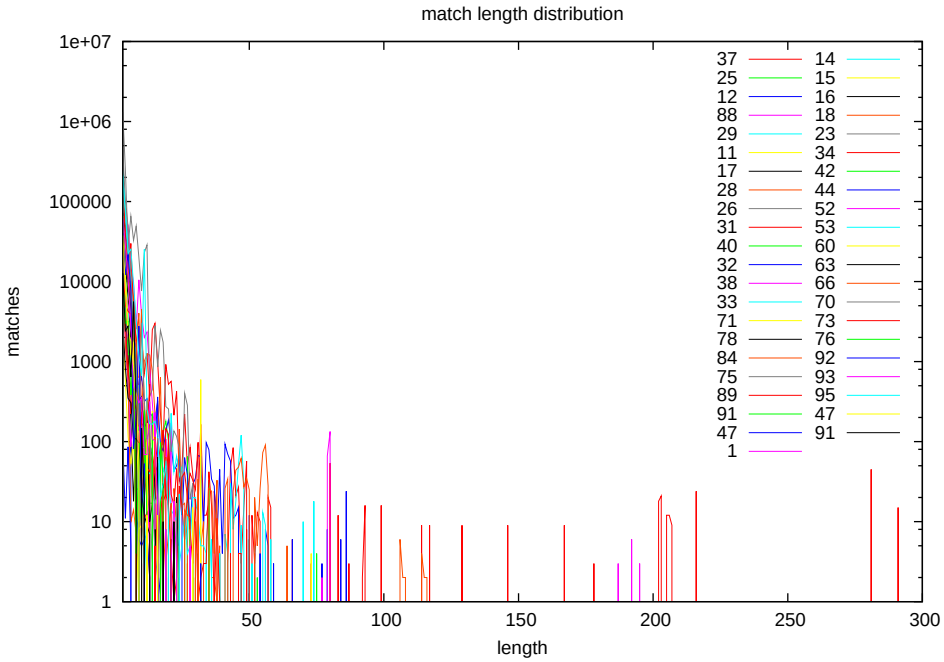


Figure 6.3.: Matchlength distribution of all members of class B in the String Matcher evaluation. Note the ordinate is scaled logarithmic. The numbers are referring to the respective entries in the list given in Appendix C.

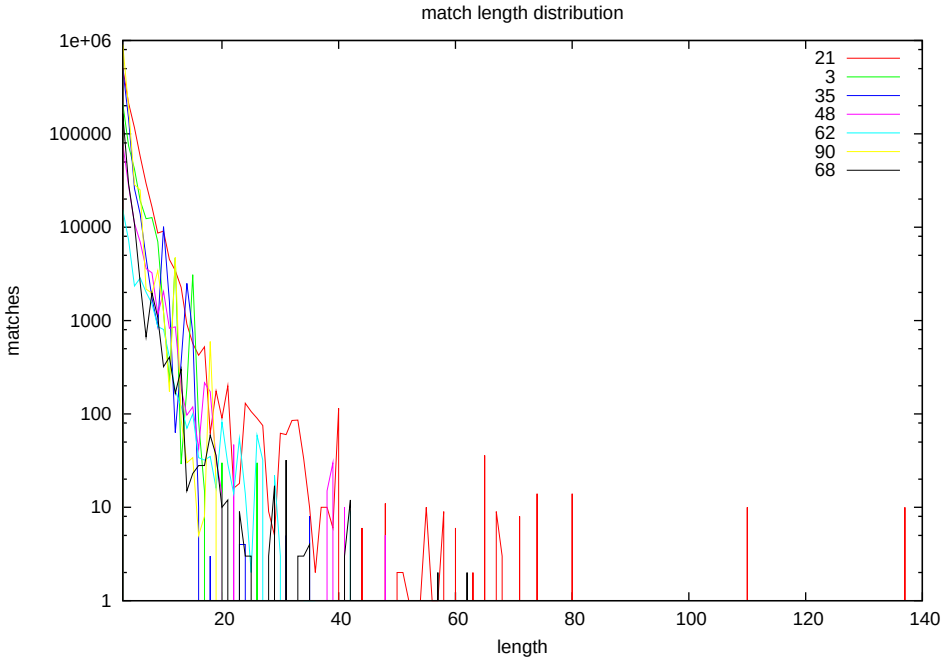


Figure 6.4.: Matchlength distribution of all members of class C in the String Matcher evaluation. Note the ordinate is scaled logarithmic. The numbers are referring to the respective entries in the list given in Appendix C.

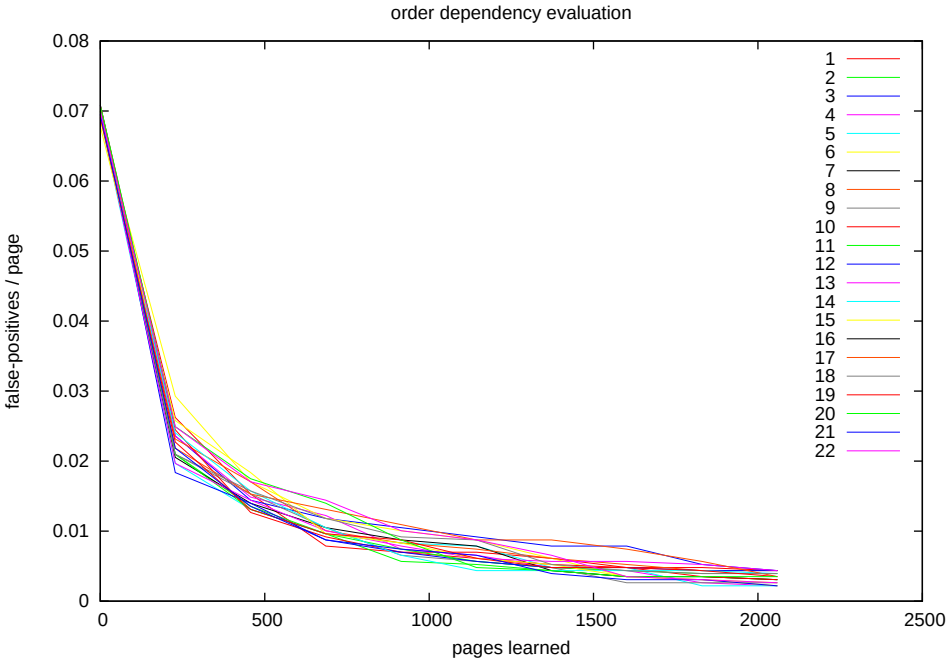


Figure 6.5.: Scriptbase evaluation of flickr.com with 10 different randomized document orders / test-set selections

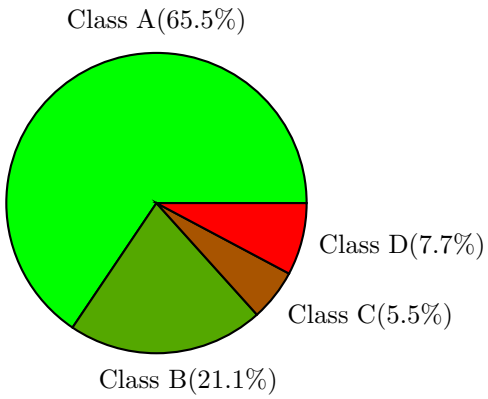


Figure 6.6.: Relative class percentages of Scriptbase evaluation.

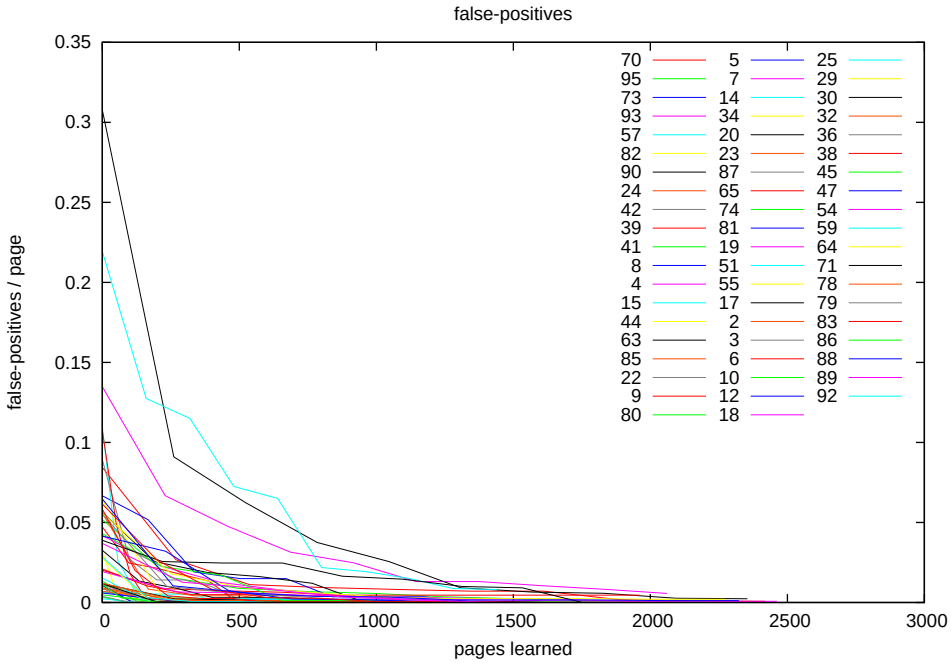


Figure 6.7.: Scriptbase evaluation of class A Web Applications. The numbers are referring to the respective entries in the list given in Appendix C.

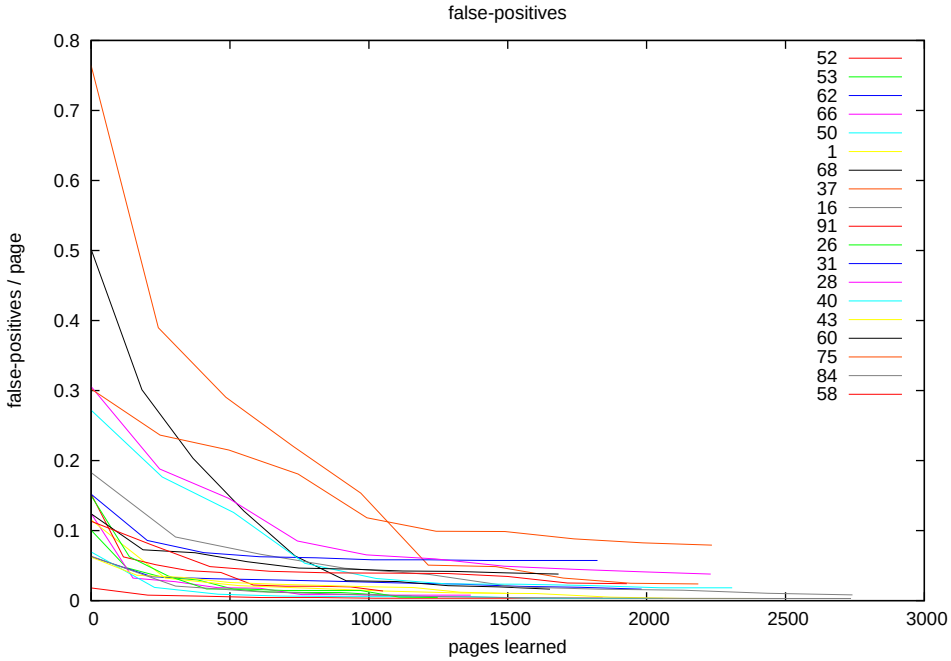


Figure 6.8.: Scriptbase evaluation of class B Web Applications. The numbers are referring to the respective entries in the list given in Appendix C.

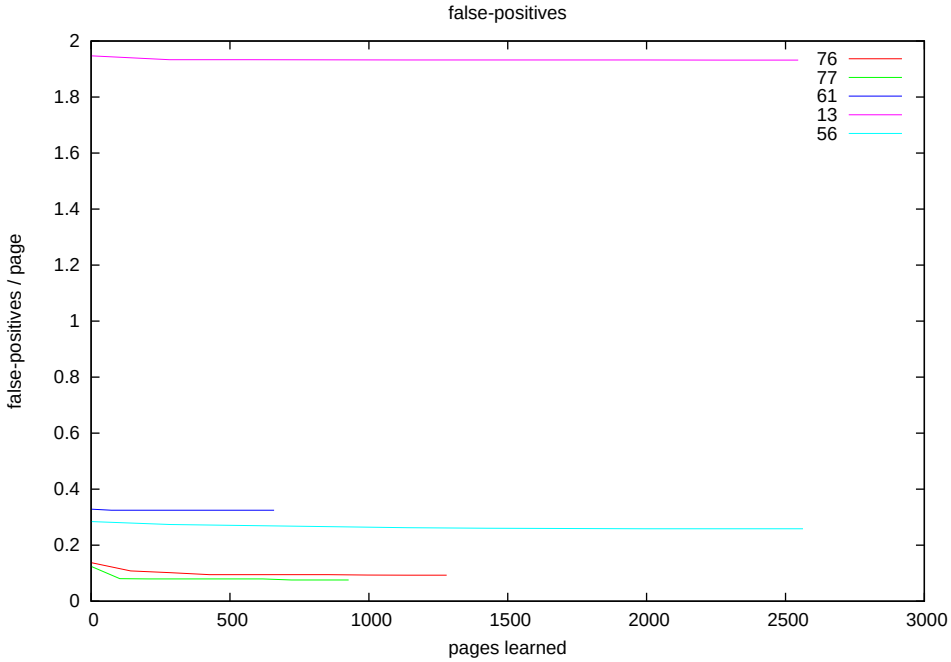


Figure 6.9.: Scriptbase evaluation of class C Web Applications. The numbers are referring to the respective entries in the list given in Appendix C.

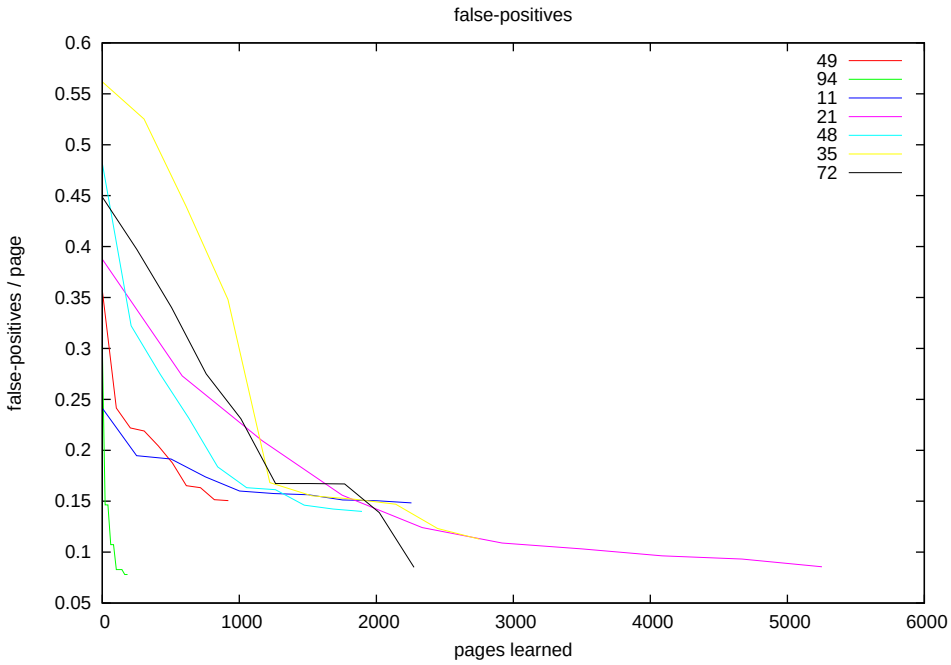


Figure 6.10.: Scriptbase evaluation of class D Web Applications. The numbers are referring to the respective entries in the list given in Appendix C.

6.3. Scriptbase Evaluation

We evaluated the pure Scriptbase without any policies applied. One may argue this to be an unrealistic setting, but as it is the basic algorithm, we definitely have to evaluate it first, in order to correctly measure the policy's impact.

Even though the policies were not implemented, we will present the basic evaluation results, as they give a good insight in the correctness of our assumptions. Additionally, they give good hints for deciding how to improve the concept.

Subsection 5.3.2 already discussed what kinds of malicious scripts were generated and used for evaluation. As none of these was using any DOM-based tricks, our tool was, due to its design, obviously able to produce a constant zero false-negatives along all evaluations, even without applying the 'restricted dynamic code matching' policy. We thus will omit the graphs as they do not contain any new information.

The documents in the database are in the order of their retrieval by the Crawler. When dividing them into training- and test-set, the documents in the training-set are thus older. In order to determine the impact of the document order / training set selection, we conducted additional evaluations on a small number of Web Applications, with the documents being reordered randomly and the training-set thus being drawn randomly, too. Figure 6.5 shows the example of `flickr.com`. As all graphs still exhibit the very same asymptotic behaviour, we concluded that given a sufficiently large training set, the detector's performance is practically independent of the document order.

A last note about exceptions: there were about 20 exceptions during the whole evaluation process apart from a number or parsing errors that were handled as described in Subsection

5.2.2. They were caused by malformed data in the database due to crawler crashes during data collection. The affected request-response pairs were excluded from the evaluation.

6.3.1. Statistics

The 95 Web Applications used for evaluation (a list can be found in Appendix C) could be categorized into 4 classes based on how well our tool worked on them. Figure 6.6 shows the relative percentages of the respective classes.

Class A (the greenmost) holds all the Web Applications, on which the tool worked fine. As can be seen in Figure 6.7, false-positive-rates are asymptotically approaching zero and within the test-set already fall way below 0.1% (which equals 1 annoying popup / log entry every 1000 pages). On this class all our assumptions held and the tool is ready for practical use already.

Class B contains all the applications that were breaking one of our assumptions occasionally, and therefore exhibited slightly unexpected behaviour regarding the matching of their scripts. Typically, members of this class are showing false-positive-rates not asymptotically dropping to zero but approaching some constant slightly above (Figure 6.8). As with all following classes, the reasons for this will be discussed in more detail in the next Subsection.

The next class is called the 'constants'. Applications belonging to this class were so extensively breaking one or more of our assumptions that the Scriptbase was not able to improve it's matching any more, yielding a constant rate of false classifications (Figure 6.9).

We believe it is possible to improve our matching in order for the Scriptbase to perform on all cases mentioned so far as it already did on the ones in the first class. Some ways to do this will be mentioned in the next Subsection, while a more detailed discussion can be found in Chapter 8.

The last class, however, while also breaking a lot of our assumptions, contained a vast amount of scripts that were related in a way not even apparent to us as human observers. These scripts either were the result of some very advanced code generation techniques or this sites hired a lot of people writing them by hand. The results of this class are depicted in Figure 6.10.

6.3.2. Unexpected Cases

After the statistical results have been discussed, we will now review the different types of unexpected behaviour found in the applications of classes B-D as this will hopefully yield some useful information for improving our algorithm.

Dynamic external scripts from other domains

We had expected external scripts to be dynamic, but only allowed this when the scripts were originating in the same domain as the document (same origin policy). As it turns out, a number of sites were using script-servers registered under a different domain (for example ebaystatic.com in the case of ebay.com). Additionally, advertising companies were causing dynamic external scripts to be included in their client's pages, and a number of smaller sites were using external usage tracker services, passing information about visited pages in the parameters.

An easy way to improve the matching would thus be extending the same origin policy to a known origin policy, keeping track of all domains external scripts were included from and alerting only on new ones.

Looped dynamic scripts

A very common cause for false-positives were loops in generating functions. This often appeared when the data passed was an array or a more complicated data structure. Cases where a certain function was called several times with different sets of parameters were also encountered, resulting in a lot of scripts only differing in the number of iterations.

Fixing this problem certainly is non-trivial, but could be done based on repeated-substring detection on the token-lists. The matching process, however, would become a lot more complicated.

Dynamic variable names

If scripts could possibly have multiple occurrences on a single page, namespace collisions can cause major problems. Instead of applying techniques from object-oriented programming that are supported in JavaScript and were invented to solve this issues, some developers resort to appending unique numbers to their variable names. Unlike looped dynamic scripts, a script like this is able to generate an incredible number of pairwise unmatching occurrences, and - if used extensively - causes class C false-positive rates.

Of course, canonizing all variable names prior to comparison would be quite easy, but the ability to change variable names could easily be used for passing data from one script to another. As a formal basis is totally missing, we can not be sure what can be accomplished by combining several benign scripts and passing data between them.

Optional parts

Conditional branching (if-clauses) in a generating function can cause code-parts to be included in some occurrences while missing in others. While a single part like this is obviously not a problem, a script with n optional parts means 2^n possible occurrences.

In theory, this can be viewed as a special case of looped dynamic scripts with zero to one iterations, but we cannot apply repeated substring detection here, because optional parts are not repeated.

Fortunately, this kind of dynamic script is very rare and the maximum number encountered for n was 4.

Dynamic ordering

Sometimes, the passed data was either a mapping or a long list of variable assignments (which is quite similar). In this case, it could be observed that not only was the number of items changing, but also was their order.

We believe this phenomenon is caused by the script being partly generated from the contents of a hashtable which - in most implementations - is not preserving order. Again this type of dynamic script is quite rare, but matching it could pose a major challenge.

Malformed JavaScript

On one site, several false alarms were caused by strings containing multiple unescaped quotes. These were interpreted by the tokenizer as multiple strings and thus did not match the prototype any more. It is quite amusing, that our XSS-detector is also detecting failures in input-/output filtering.

On `msn.com`, we found a lot of different HTML-code enclosed in script-tags and therefore causing false-positives. We were, up to now, completely unable to find an explanation why anyone would want to do this, as - at least in Firefox - this code is just causing a parse-error and therefore ignored. Speculations on this Firefox-behaviour itself being the reason are beyond the scope of this thesis.

Code generation

A definite example of code generation was encountered on lawblog.de. The script was performing some mathematical computation and stored the result in a hidden form field replied back to the server. Noteworthy about this was the complete computation changing every time the page was revisited. Either the author was using his visitors for performing some distributed computation or it was just a very elaborate way of testing whether a browser supports JavaScript.

6.4. Limitations

When reviewing all limitations discussed so far, three major problem categories are emerging. We will in the order of impact devote each of them a paragraph for summing up their various implications:

Conceptual Problems

The current conception is completely ignoring a Webpage's dynamic nature by scanning it only once. This conceptual weakness of course has implications for the detection ability. In AJAX applications, client-side code, that is dynamically loaded at runtime, obviously cannot be secured, which may let stored attacks in this content go undetected. Additionally, the detector fails to observe any kind of DOM-based XSS attack vectors as well as combinations thereof with conventional ones. While a server-side IDS sensor can use fine-grained matching policies to severely limit an adversaries freedom of action, a similar modus operandi would elsewhere result in rapidly increasing false-positive rates.

Trading scope versus security

Revising the matching algorithms for supporting a greater number of Web Applications is a balancing act between scope and security. On the one hand there are algorithms finding loops, not caring for variable name changes and only raising an alarm on very improbable multi-token parameter matches. While this kind of algorithm would surely be 'applicable' to most Web Applications out there in the Internet, there surely would still be some exceptions. The contrary side holds a lot of very elaborate policies, tailored to every single niche that can be found in Web Application development. Each surely able to bring a very high level of security to a special subset of Web Applications, but applicable to these only.

Implementation issues

Next to the conceptual problems, there are some 'matters of time' left in the implementation. These are not severe but definitely should be solved prior to any practical applications.

This Chapter thoroughly evaluated all three components of our XSS detector. A number of shortcomings were revealed, many of which will be matched with Ideas for their resolution in Chapter 8. We will next have a look at a selection of related concepts, also developed to counter the XSS threat.

7. Related Work

The field of Web Application security is very large and rapidly evolving. The selection of work presented in this Chapter can be considered only a fraction of the overall work undertaken, and is by no means exhaustive.

Cross Site Scripting is essentially an input filtering failure. Consequently, methods have been developed to target malicious inputs even before they reach the Webserver.

Traditionally, Web Application Firewalls are either scanning for attack signatures in the parameters passed to the Web Application [20] (including POST-parameters, cookies, etc.), or require an administrator to manually specify a ruleset to match requests against [21]. Both ways can be regarded as a very sophisticated second input filtering layer.

A First Anomaly-based Intrusion Detection System for Web Applications was proposed by Kruegel and Vigna in [22]. Their system is deriving a number of statistical characteristics from observed HTTP requests, regarding the parameter's length, character distribution, structure, presence and order. Like the previous approaches, however, it concentrates solely on the query parameters.

After the Web Application generated the pages, the problem has shifted to finding all active content and determining the illegitimate ones. While the Web Application knows which scripts are legitimate, but fails to detect the others, the browser is perfectly detecting all scripts, but unaware of their legitimacy. From this insight, some cooperative approaches have been put forward, requiring Webserver and browser to communicate about the scripts:

With Browser-Enforced Embedded Policies (BEEP)[18], the Webserver is supposed to include a whitelist-like policy into each page, allowing the browser to filter unwanted scripts. As the policy itself is a JavaScript, this method is very flexible and for instance allows the definition of regions, where scripts are disallowed. Content Restrictions, a mechanism proposed by Gervase Markham in [23] is based on that latter idea, but uses a finer-grained policy language.

Although such methods definitely are capable of solving the XSS problem for Web Applications applying them correctly, they do little to mitigate the risk emanating from the majority that does not. Thus, a number of pure client-site solutions have been developed:

The noscript-plugin for Firefox [24], prevents the browser from executing active content from domains not whitelisted. It can be understood as a domain-based access control for the active content execution. Even though this definitely is a commendable policy, it unfortunately fails to address XSS issues in heavily used sites which are extensively relying on client-side scripting. In noscript, these are usually whitelisted and - from the user's point of view - their credentials are the ones most worthy of protection.

Hallaraker and Vigna in [19] chose a completely different way: They modified Mozilla's SpiderMonkey Engine as to track the behaviour of every client-side JavaScript executed by Firefox. The activity profile of each script then is matched against a set of high-level policies for detecting malicious behaviour.

8. Conclusions and Future Work

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

–Alan Turing

In this thesis, we have presented two novel approaches for detecting Cross Site Scripting attacks in dynamic Web Applications using a modified browser as a script detector. These approaches are not requiring knowledge about the specific Web Application, as they are inferring the wherewithals using data-mining techniques on HTTP traffic. They therefore can also be applied outside the Web Application provider's sphere of influence and may along with [19] constitute a first step towards Anomaly-based, pure client-side Cross Site Scripting protection.

Both methods were implemented and evaluated, which revealed a number of shortcomings summed up in Section 6.4. Future work will therefore focus on improving the matching algorithms in a number of ways listed henceforth.

8.1. Client-side Browser Enhancement

In order to make the XSS detector practically usable on the client-side, three major enhancements are necessary:

8.1.1. Dynamic Matching

Firstly, as strict application of the 'restricted dynamic code matching' policy is due to high Web Application diversity not an option, the need to overcome the problem of dynamically injected code is much more pressing on the client-side. Fortunately, the browser is in a unique position to do exactly that. If the XSS detector is integrated into the browser's asynchronous pipeline as outlined in Section 4.5, it would also become possible to dynamically scan scripts introduced at runtime. As such scripts would also be added to the Scriptbase, it would both make the detector applicable to AJAX applications, and enable it to detect all flavors of DOM-based XSS.

8.1.2. Behavioral-based Matching

Another nice thing, possible only in the browser, is circumventing the problematic tradeoff between scope and security outlined in Section 6.4. The root of this problem is the difficulty to judge whether a certain change in a script's code will cause it's behaviour to change dramatically or not.

In [19], Hallaraker and Vigna introduced a method enabling Mozilla's SpiderMonkey engine to track the behaviour of every script executed within a Webpage. A misuse-based IDS was used comparing these behaviours to high-level policies, for instance preventing a script that

accessed a cookie from initiating HTTP requests. While such policies usually are applied globally, and require great effort to be adapted for every single Web Application, it was also stated in [19] that "the auditing system [...] does not restrict the IDS to be of any specific type".

A combination of this approach with ours, would not have to concern itself with behavioral impacts of syntactic changes, but could match actual behaviours directly. Instead of with a user's classification, a script occurrence could in the Scriptbase be associated with some behavioral fingerprint derived from observing it during execution. Dynamic code matching could thus be allowing loops, variable name changes, etc. without causing security loopholes, as an alarm would be raised if the script deviated from its previous behaviours too much. This combination would thus yield an Anomaly-based Web Application IDS for XSS.

8.1.3. Automatic Web Application Partitioning

Unlike on the server-side, coverage of the Web Application during the *learning*-phase usually is far from complete, as only a single user is using it. Most users tend to visit certain parts of a Web Application over and over again, while completely ignoring others. The Scriptbase would thus very quickly reach a non-changing state in this important parts, while not containing any information about the rest. Deciding when to switch to *protection*-mode therefore becomes a dilemma: While switching early can protect the important parts quickly, it also yields tremendous amounts of false-positives would the user decide to explore the Web Application later on. On the contrary, waiting for better coverage would never protect anything if the user does not change his behaviour.

A much better solution would be partitioning the Web Application into known and unknown parts. This way the detector could already protect the known ones while still learning the rest. The problem obviously lies in finding a way to derive this partitioning that

- represents actual structures present within the Web Application
- can not be influenced by an adversary to switch the detector back into *learning* mode withing a known part

Subsection 4.3.2 gives an explanation as to why it is very hard to do this on URLs alone. This argumentation, however, is only valid for the set of all Web Applications. Even though there are no 'globally' applicable rules, every Web Application exhibits certain structural principles easily detectable for individuals with a certain amount of experience in Web Application development. By searching for interactions between URLs and documents it may be possible to detect such structures automatically.

Additional information could be derived from the Linkage-Graph or the user's surfing behaviour. This certainly is an interesting problem with applications in Web Application fingerprinting.

8.2. Server-side IDS Sensor

Before applying the presented concepts for an IDS sensor in any practical environment, it certainly is necessary to implement the policies from Subsection 4.3.6 and evaluate their impact on the false-positive rates. Our proof-of-concept implementation, however, should not

be used practically on the server-side, since it is not throughput-optimized and may therefore be much too slow even for small Web Applications.

But another rather interesting application of the String Matcher algorithm has not been brought up by now:

8.2.1. Stored XSS Forensics

Whenever a stored XSS payload is found in an application, it usually is much too late for any investigation as to who injected it and using which methods. Even if all HTTP-requests were routinely logged, the effort of reviewing these logs for a large Web Application usually dwarfs the benefit. By letting the String Matcher search for similarities between the payload and parameters from all previous requests, this task can be automated easily. Once the IDS sensor detects a stored attack, the attack's method and source-IP address can therefore be investigated much faster.

8.3. String Matcher Improvements

Future work on the String Matcher should add a tokenizer to adequately handle dynamic string matches and keyword matches. It is also advisable to investigate calculating a probability for the removed characters to actually be the product of an input filter. This task is quite difficult as the filter could not only be sensible to characters but also to sequences. There, however, should be a difference between filtering for JavaScript keywords and removing random sequences.

Another issue not yet examined fully is the problem of wrongly reported or unknown character encodings. As shown in [42], input filters of short Webpages that are not stating their encoding in the HTTP header, can sometimes be evaded by using UTF-7 encoding. It is not clear whether this also applies to our String Matching algorithm. The possibility of an input filter changing a strings encoding should be considered as well as the two additional JavaScript-encodings (hexadecimal and unicode) mentioned in Subsection 4.2.1.

8.4. Scriptbase Improvements

In order to improve the Scriptbase, the proposals from Section 6.3 should be realized firstly. External scripts should be handled according to the known origin policy and loops should be detected during dynamic code matching.

For practical use, it is additionally relevant to make the implementation store the Scriptbase's data on disk instead of holding it in memory. While the average PC's memory is completely sufficient even for Models of large Web Applications, a realistic client-side implementation should be able to protect a number of these in parallel.

8.5. Script Finder Improvements

There are currently only two open implementation issues in the ScriptFinder. When these are resolved it should be able to find both JavaScript URLs and scripts embedded into content other than HTML. As VBScript is not available in Firefox, and all other active content should

be listed in either `document.embeds` or `document.applets`, the detector then can easily be extended to provide lists of all active content.

Bibliography

[1] Martin Johns: "SessionSafe: Implementing XSS Immune Session Handling", European Symposium on Research in Computer Security (ESORICS 2006), Hamburg, Germany, September 2006, Gollmann, D.; Meier, J. & Sabelfeld, A. (ed.), Springer, LNCS 4189, pp. 444-460, http://www.informatik.uni-hamburg.de/SVS/papers/2006_esorics_SessionSafe.pdf, accesses 2007.05.21

[2] Stefano Di Paola, Giorgio Fedon: "Subverting AJAX", 23rd Chaos Communication Congress (23C3), Berlin, Germany, December 2006, http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf, accessed 2007.05.21

[3] Amit Klein: "DOM Based Cross Site Scripting or XSS of the Third Kind", Website, http://www.webappsec.org/projects/articles/071105.shtml, accessed 2007.05.21

[4] SPI Labs: "Detecting, Analysing, and Exploiting Intranet Applications using JavaScript", SPI Dynamics Research Brief, 2006, http://www.spidynamics.com/assets/documents/JSportscan.pdf, accessed 2007.08.24

[5] Open Web Application Security Project: "OWASP Top 10 Web Application Security Issues 2007", Website, http://www.owasp.org/index.php/Top_10_2007, accessed 2007.07.20

[6] CERT: "Malicious HTML Tags Embedded in Client Web Requests", Advisory CA-2000-02, Website, February 2000, http://www.cert.org/advisories/CA-2000-02.html, accessed 2007.08.28

[7] Steve Christey, Robert A. Martin: "Vulnerability Type Distribution in CVE", Common Weakness Enumeration, version 1.1, technical white paper, May 2007, http://cwe.mitre.org/documents/vuln-trends/index.html, accessed 2007.07.23

[8] Michael Sutton, Jeremiah Grossman, Sergey Gordeychik, Mandeep Khera: "Web Application Security Statistics", Web Application Security Consortium, Website, December 2006, http://www.webappsec.org/projects/statistics/, accessed 2007.07.23

[9] ECMA: "ECMAScript Language Specification", 3rd Edition, December 1999, Website, http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf, accessed 2007.05.24

[10] T. Berners-Lee, L. Masinter, M. McCahill: "Uniform Resource Locators (URL)", RFC 1738, December 1994, http://www.faqs.org/rfcs/rfc1738.html, accessed 2007.08.06

[11] The Unicode Consortium: "The Unicode Standard, version 5.0", 5th edition, Addison-Wesley Professional, November 3, 2006, ISBN: 0321480910, http://www.unicode.org/standard/standard.html, accessed 2007.08.13

[12] World Wide Web Consortium HTML Working Group: "HTML 4.01 Specification", Website, http://www.w3.org/TR/1999/REC-html401-19991224/, accessed 2007.08.15

[13] D. Kristol, L. Montulli: "HTTP State Management Mechanism", RFC 2965, October 2000, http://www.faqs.org/rfcs/rfc2965.html, accessed 2007.08.28

[14] Computer Incident Advisory Capability: "Internet Cookies", Information Bulletin I-034, March 1998, http://www.ciac.org/ciac/bulletins/i-034.shtml, accessed 2007.08.28

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee: "Hypertext Transfer Protocol – HTTP/1.1", RFC 2616, June 1999, http://www.w3.org/Protocols/rfc2616/rfc2616.html, accessed 2007.08.28

[16] Karen Scarfone, Peter Mell: "Guide to Intrusion Detection and Intrusion Prevention Systems (IDPS)", Recommendation of the National Institute of Standards and Technology, Special Publication 800-94, http://csrc.nist.gov/publications/nistpubs/800-94/SP800-94.pdf, accessed 2007.08.20

[17] Zhendong Su, Gary Wassermann: "The Essence of Command Injection Attacks in Web Applications", Annual Symposium on Principles of Programming Languages (POPL'06), Charleston, January 2006, http://www.cs.ucdavis.edu/~su/publications/popl06.pdf, accessed 2007.05.24

[18] Trevor Jim, Nikhil Swamy, Michael Hicks: "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies", 16th International World Wide Web Conference (WWW 2007), Banff, Alberta, Kanada, May 2007, http://www.cs.bell-labs.com/cm/cs/who/pfps/temp/web/www2007.org/papers/paper595.pdf, accessed 2007.07.11

[19] Oystein Hallaraker, Giovanni Vigna: "Detecting Malicious JavaScript Code in Mozilla", 10th IEEE International Conference on Engeneering of Complex Computer Systems (ICECCS 2005), Shanghai, China, June 2005, IEEE Computer Society Press, USA, ISBN 0-7695-2284-X, pp. 85-94 http://doi.ieeecomputersociety.org/10.1109/ICECCS.2005.35, accessed 2007.08.19

[20] Amit Klein: "Cross Site Scripting Explained", whitepaper, Sanctum Security Group, June 2002, http://crypto.stanford.edu/cs155/CSS.pdf, accessed 2007.05.28

[21] D. Scott, R. Sharp: "Abstracting application-level web security", 11th ACM International World Wide Web Conference, ACM Press New York, NY, USA, 2002, pages 396-407, http://doi.ieeecomputersociety.org/10.1109/ICECCS.2005.35, accessed 2007.08.19

[22] Christopher Kruegel, Giovanni Vigna: "Anomaly Detection of Web-based Attacks", 10th ACM Conference on Computer and Communications Security (CCS'03), Washington D.C., USA, 2003, ACM Press, NY, USA, pages 251-261, ISBN 1-58113-738-9, http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=948144, accessed 2007.08.28

[23] Gervase Markham: "Content Restrictions", Website, version 0.9.2, March 2007, http://www.gerv.net/security/content-restrictions/, accessed 2007.08.28

[24] Giorgio Maone: "noscript Firefox Add-on", version 1.1.6.16, Website, https://addons.mozilla.org/de/firefox/addon/722, accessed 2007.08.28

[25] Dino Esposito: "Cookieless ASP.NET", MSDN, May 2005, Website, http://msdn2.microsoft.com/en-us/library/aa479314.aspx, accessed 2007.06.20

[26] Albert Weinert, Thomas Brandt: "URLRewritingNet.URLRewrite Documentation", revision 2.0, 2006, http://www.urlrewriting.net/download/UrlRewritingNet20.English.pdf, accessed 2007.06.20

[27] The Apache Software Foundation: "mod_rewrite reference documentation", Website, http://httpd.apache.org/docs/2.0/de/mod/mod_rewrite.html, accessed 2007.06.20

[28] Paul Sowden: "rbNarcissus - A JavaScript Parser ported to Ruby", Website, http://idontsmoke.co.uk/2005/rbnarcissus/, accessed 2007.07.10

[29] David Maier: "The Complexity of Some Problems on Subsequences and Supersequences", Journal of the ACM, Volume 25, Issue 2, Pages 322-336, ACM Press, New York, NY, USA, April 1978, DOI: http://doi.acm.org/10.1145/322063.322075, accessed 2007.08.01

[30] D.S. Hirschberg: "A linear space algorithm for computing maximal common subsequences", Communications of the ACM, Volume 18, Issue 6, Pages 341-343, ACM Press, New York, NY, USA, June 1975, DOI: http://doi.acm.org/10.1145/360825.360861, accessed 2007.08.01

[31] Kenichi Morita, Noritaka Nishihara, Yasunori Yamamoto, Zhiguo Zhang: "A Hierarchy of Uniquely Parsable Grammar Classes and Deterministic Acceptors", Acta Informatica, Volume 34, Number 5, Pages 389-410, 1997, http://citeseer.ist.psu.edu/morita97hierarchy.html, accessed 2007.08.07

[32] Esko Ukkonen: "On-line construction of suffix trees", Algorithmica, Volume 14, Number 3, Pages 249-260, 1995, http://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf, accessed 2007.08.09

[33] Ricardo A. Baeza-Yates, Gaston H. Gonnet: "Fast Text Searching for Regular Expressions or Automaton Searching on Tries", Journal of the ACM, Volume 43, Number 6, Pages 915-936, November 1996, http://portal.acm.org/citation.cfm?id=235810&coll=portal&dl=ACM, accessed 2007.08.09

[34] Dan Gusfield: "Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology", Cambridge University Press, New York, USA, 1997, ISBN 0521585198

[35] Graham A. Stephen: "String Search", Technical Report TR-92-gas-01, School of Electronic Engineering Science, University College of North Wales, October 1992, http://wwwcdf.pd.infn.it/localdoc/string_search.ps.gz, accessed 2007.08.28

[36] Doug Turner, Ian Oeschger: "Creating XPCOM Components", article, Mozilla Developer Center, 2003, http://developer.mozilla.org/en/docs/Creating_XPCOM_Components, accessed 2007.08.27

[37] Microsoft Corporation: "Mitigating Cross-site Scripting With HTTP-only Cookies", MSDN, http://msdn2.microsoft.com/en-us/library/ms533046.aspx, accessed 2007.08.24

[38] Stefan Esser: "httpOnly Firefox Add-on", version 0.5, Website, https://addons.mozilla.org/de/firefox/addon/3629, accessed 2007.08.24

[39] Alexa Internet: "Alexa Traffic Ranking Global Top500", Website, http://www.alexa.com/site/ds/top_500, accessed 2007.08.13

[40] Robert Hansen: "XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion", Website, http://ha.ckers.org/xss.html, accessed 2007.08.17

[41] Paul Battley: "HTML Entities for Ruby", Website, http://htmlentities.rubyforge.org/, accessed 2007.08.18

[42] Yair Amit: "XSS vulnerabilities in Google.com", security advisory, Web Security Mailing List, November 2005, http://www.webappsec.org/lists/websecurity/archive/2005-12/msg00059.html, accessed 2007.08.20

[43] mattmurphy: "Apache HTTP Server SSI Error Page XSS", security advisory, October 2002, CVE ID: 2002-0840, OSVDB ID: 862, Bugtraq ID: 5847, http://osvdb.org/displayvuln.php?osvdb_id=862, accessed 2007.07.23

[44] BCC: "to_json cross site scripting security issue (XSS)", Ruby on Rails bug report, Website, May 2007, http://dev.rubyonrails.org/ticket/8371, accessed 2007.07.28

[45] Stefan Esser: "PHP ext/session HTTP Response Splitting Vulnerability", Hardened-PHP project, Security Advisory, January 2006, http://www.hardened-php.net/advisory_012006.112.html, accessed 2007.07.28

[46] Samy: "Technical explanation of The MySpace Worm", Website, http://namb.la/popular/tech.html, accessed 2007.05.21

A. List of Annotations to used Software

This lists gives version information for all software of which the used version might be interesting. In some cases annotations to eventual modifications are added.

rbNarcissus v0.1: a bug in the string matching routines had to be fixed to correctly handle escaped quotes within string tokens.

wget v1.10: added additional command-line options for accessing the URL, request and response-header.

hpricot v0.5.115: since the latest stable version kept crashing on some very large document, we had to use the latest SVN version.

htmlentities v4.0.0

mysql v14.12

ruby v1.8.6

Mozilla Firefox 2.0rc3 release

B. List of XSS Payloads used for Evaluation

The following payloads where taken from [40] and checked for Firefox compliance. They are demonstrating filter evasion techniques and not implementing any concrete attacks. External files are listed at the end. All payloads not detected by the proof-of-concept Script Finder implementation are marked in red.

1. `<SCRIPT SRC=xss.js></SCRIPT>`
2. `<IMG """><SCRIPT>alert("XSS")</SCRIPT>">`
3. `<SCRIPT/XSS SRC="xss.js"></SCRIPT>`
4. `<SCRIPT/SRC="xss.js"></SCRIPT>`
5. `<<SCRIPT>alert("XSS");//<</SCRIPT>`
6. `<SCRIPT SRC=xss.js?<B>`
7. `<SCRIPT>a=/XSS/`
   `alert(a.source)</SCRIPT>`
8. `<BODY ONLOAD=alert('XSS')>`
9. `<STYLE>BODY{-moz-binding:url("xssmoz.xml#xss")}</STYLE>`
10. `<META HTTP-EQUIV="refresh" CONTENT="0;url=javascript:alert('XSS');">`
11. `<META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html;base64,`
    `PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">`
12. `<IFRAME SRC="javascript:alert('XSS');"></IFRAME>`
13. `<EMBED SRC="`
    `A6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB4bWxuczOiaHR0cDovL3d3dy53My5vcmcv`
    `MjAwMC9zdmciIHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hs`
    `aW5rIiB2ZXJzaW9uPSIxLjAiIHg9IjAiIHk9IjAiIHdpZHRoPSIxOTQiIGhlaWdodD0iMjAw`
    `IiBpZD0ieHNzIj48c2NyaXB0IHR5cGU9InRleHQvZWNtYXNjcmlwdCI+YWxlcnQoIlh TUyI`
    `pOzwvc2NyaXB0Pjwvc3ZnPg==" type="image/svg+xml"`
    `AllowScriptAccess="always"></EMBED>`
14. `<SCRIPT SRC="xss.jpg"></SCRIPT>`
15. `<SCRIPT a=">" SRC="xss.js"></SCRIPT>`
16. `<SCRIPT =">" SRC="xss.js"></SCRIPT>`
17. `<SCRIPT a=">" '' SRC="xss.js"></SCRIPT>`

18. `<SCRIPT "a='>'" SRC="xss.js"></SCRIPT>`
19. `<SCRIPT a=">'>" SRC="xss.js"></SCRIPT>`
20. `<SCRIPT>document.write("<SCRI");</SCRIPT>PT SRC="xss.js"></SCRIPT>`

xss.js

```
alert("XSS !!!");
```

xssmoz.xml

```
<?xml version="1.0"?>
<bindings xmlns="http://www.mozilla.org/xbl">
  <binding id="xss">
    <implementation>
      <constructor><![CDATA[alert('XSS')]]></constructor>
    </implementation>
  </binding>
</bindings>
```

C. List of Web Applications, that were crawled for Data Collection

1. about.com
2. apfelwiki.de
3. arstechnica.com
4. blog.beetlebum.de
5. blog.fefe.de
6. blog.mabber.de
7. blog.s9y.org
8. blog.sevenload.de
9. blog.webmaster-homepage.de
10. board.gulli.com
11. de.news.yahoo.com
12. de.wikinews.org
13. digg.com
14. en.wikipedia.org
15. events.ccc.de
16. forum.freenet.de
17. forum.golem.de
18. forum.herr-der-ringe-film.de
19. forum.ksta.de
20. freifunk.net
21. geocities.yahoo.com
22. itblog.eckenfels.net
23. moinmoin.wikiwikiweb.de
24. neopets.com
25. news.google.com
26. news.google.de
27. news.yahoo.com
28. nytimes.com
29. shop.manager-magazin.de
30. shop.zdf.de
31. shop.zeit.de
32. slashdot.org
33. sourceforge.net
34. wiki.genealogy.net
35. www.amazon.com
36. www.anaconda-game.de
37. www.aol.com
38. www.ard.de
39. www.basicthinking.de
40. www.bbc.co.uk
41. www.blog-frosta.de
42. www.blogger.com
43. www.buha.info
44. www.de-bug.de
45. www.deutsches-museum-shop.com
46. www.dream-multimedia-tv.de
47. www.dvdboard.de

48. www.ebay.com
49. www.facebook.com
50. www.flickr.com
51. www.forum-sozialhilfe.de
52. www.fotolog.com
53. www.friendster.com
54. www.germnews.de
55. www.golem.de
56. www.heise.de
57. www.imageshack.us
58. www.kampfkunst-board.info
59. www.kinonews.de
60. www.klamm.de
61. www.lawblog.de
62. www.livejournal.com
63. www.maennerseiten.de
64. www.matheboard.de
65. www.medizin-forum.de
66. www.monster.com
67. www.motor-talk.de
68. www.msn.com
69. www.musiker-board.de
70. www.myspace.com
71. www.nachtwelten.de
72. www.netzwelt.de
73. www.orkut.com
74. www.ot-forum.de
75. www.otto.de
76. www.pogo.com
77. www.recht.de
78. www.rockshop.de
79. www.shop.de
80. www.speisekarten-seite.de
81. www.stpauli-forum.de
82. www.studivz.net
83. www.study-board.de
84. www.target.com
85. www.theofel.de
86. www.thinkgeek.com
87. www.tiddlywiki.com
88. www.tigerdirect.com
89. www.walmart.com
90. www.weather.com
91. www.welt.de
92. www.winfuture-forum.de
93. www.xanga.com
94. www.yahoo.com
95. www.youtube.com

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als soche kenntlich gemacht.
Weiterhin bin ich mit der Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den 10. November 2015,

(Björn Engelmann)