DOM-based XSS Attacks

M.Sc. Project thesis

Zdravko Danailov zdravko.danailov@rub.de

Krassen Deltchev krassen.deltchev@rub.de

Ruhr-University of Bochum

Faculty Of Electrical Engineering And Information Technology

Chair for Network and Data Security
Horst-Görtz Institute

Prof. Dr. Jörg Schwenk

First examiner: Prof. Dr. Jörg Schwenk
Adviser and Second examiner: Dipl.-Ing. Mario Heiderich

16. April 2012

Abstract

Concerning the matter of modern Internet Security, we are obliged to understand the meaning and proper utilization of Web Application Security and Web Services Security. In this paper, we discuss one of the recent and misunderstood problems, related to Web Application Security.
In distinction to the standard Web Application Cross-Site Scripting Attack Vector, DOM-based XSS conveys a way to inject malicious payload on Layer 7 ISO/OSI Model on the client-side implementation of the Web-Application logic. Precisely, DOMXSS represents a subclass of this dominant Web Application Attack Vector – XSS.
Therefore, we observe the comprehensive environmental aspects of DOMXSS; we divide them in two layers – Programming environment, which concerns the DOM Abstraction Model, Web-Application coding language and the rendering engine of the Web client (Web-Browser); and Implementation environment – the Web Application programming logic and source code.
Because of the fact, that this subclass, should be still considered as misapprehended and consequentially – systematically avoided by the security and scientific communities, the main objectives of this project thesis are: proposing and evaluating of related categorizations and meta-model, demonstration of case studies and sanitization best practices, and further propositions across the problem – DOM-based XSS attacks.

Keywords

XSS, DOM-based XSS, DOMXSS, DOM-based XSS taxonomy, Web Application Security, Mobile Security, Web Application Attack Vector, Application Flow Analysis, Function Flow Analysis, Data Flow Analysis, Static Model Checkers, Dynamic Model Checkers, JSLint, Web Application scanners, DOMinator, Security patterns, JavaScript/AJAX Coding Patterns, JavaScript/AJAX Design Patterns, Anti-Patterns, DOM and Web-Browser Patterns, HTML5, Filters, ESAPI4JS, ECMAScript conventions, Secure ECMAScript, Direct Proxies

Contents

List of Tables

List of Figures

List of Abbreviations

0day	Zero-day exploit/attack
AFA	Application Flow Analysis
BM	Business Model
CSFU	Cross-site File Upload
CSRF	Cross-site Reference Forgery
DB	Database
DDoS	Distributed denial-of-service
Dev-Team	Developers Team
DFA	Data Flow Analysis
DFA*	Deterministic finite automaton
DOM	Document Object Model
DoS	Denial-of-service
DSTO	Defensive STO Model
e.g.	for example
ESAPI	Enterprise Security API
FFA	Function Flow Analysis
FO-Team	Forensics Team
i.e.	id est
IR-Team	Incident Response Team
ISO	International Organization for Standardization
LDA	Linear Discriminant Analysis
MS	Microsoft
MS SSDLC	Microsoft Security Software Development Lifecycle
NP-DOMXSS	Non-persistent DOMXSS
NP-XSS	Non-persistent XSS
OOP	Object-oriented programming
OS	Operating System
OSI	Open Systems Interconnection
OWASP	The Open Web Application Security Project
P-DOMXSS	persistent DOMXSS
PoC	Proof of Concept
PT-Team	Pen-testing Team

QA-Team	Quality Assurance Team
Regex	Regular expression
RE	Requirements engineering
RFI	Remote File Inclusion
RIA	Rich Internet Applications
ROA	Resource-oriented Architecture
SbO	Security by Obscurity
SDLC	Software Development Lifecycle
SE	Social Engineering
SH	Social Hacking
SOA	Service-oriented Architecture
SOP	Same-Origin-Policy
SQLIA	SQL injection attack
SSDLC	Security Software Development Lifecycle
S3MM	Sources/Sinks/Storage Meta-Model
STO	Strategical Tactical Operational Model
SVN	Subversion
TRRs	Tools' requirement rules
WAS	Web Application scanners
WASC	Web Application Security Consortium
Web-Dev community	Web developer community
WOA	Web-oriented Architecture
WOT	Web of Trust
W3C	W3 Consortium
XHR	XMLHttpRequest
XSS	Cross-site Scripting

1 Introduction

Nowadays, the differences between Web Application Security and Web Services Security can be hardly distinguished. On one hand their essence is still misunderstood, on the other these security areas could not be strictly separated apart. Therefore, the attacks and more precisely the attack scenarios, related to these areas, should be evaluated as unique, respecting the aspects of: their utilization and the approach to categorize them, which are prerequirements for a precise and complete examination and respectively for an effective sanitization.
In this project thesis, we shall observe Cross-Site scripting attacks on the client-side of modern Web applications, which consider exploits on the JavaScript implementations of the programming logic. Other representatives of such attacks on CSS and XML should not be discussed due to the scope of the paper, which is outlined in the section about the paper's limitations.
To achieve a taxonomic soundness of the thesis exposition, we feel obliged to give some short historical facts about JavaScript in this introduction.
In 1995 Netscape[1] proposes LiveScript[2] as an additional layer on top of the HTML. Thus, the static HTML pages become interactive and furthermore this scripting language is directly integrated into the Web-Browser. This methodology is well accepted and extended by two separated organizations. On one hand the Mozilla Foundation develops and maintains JavaScript[3], on the other Microsoft supports JScript[4]. Ironically, the so called “Browser Wars”[5] [6] reflects also on the standardization of this scripting language. Consequently to this fact, arises confusion in the Web Developers communities. First malicious attempts to exploit the LiveScript derivatives follow as an avalanche effect respectively. These attacks designate a new Web-Application Attack Vector – Cross-Site Scripting, or shortly XSS. Nowadays, XSS injections are presented not only on JavaScript, but also on PHP, ASP, JSP etc. The strive for an adequate and completed standard, considering the LiveScript proposal and respecting the current development stages of JavaScript and JScript, is presented by the evolution of ECMA-262 scripting language, or shortly ECMAScript[7]. For the interested reader, please consider further reading on Section 4.4 False usage of filters. We shall reckon still JavaScript and JScript for not completely compliant to the Modern Standards represented by ECMAScript.
Let's proceed more detailed with the introduction of the attacks on JavaScript. As already mentioned above, the most common and vigorous Web Application Attack Vector is considered to be the Cross-Site Scripting, XSS[8][AE11], [WHS10], [WHW11]. According to OWASP[9] and WASC[10], XSS represents a TOP 10 Web Application Attack Vector; statistically, see Appendix Figures 22, 23, 24, 25, 26 and Table 26, designated on second place as most severe Web Application threat. XSS is sophisticated, dynamically evolving and expanding as an attack vector. This justifies the fact for the existence of wide range of XSS attack paradigms and the intensive occurrence of new ones. Moreover, we observe a general categorization of the XSS subclasses in two main branches – sever-side and client-side. The common and well-known Web-based attacks concern an

1 http://www.scriptingmaster.com/javascript/javascript-history.asp
2 http://javascript.about.com/od/reference/a/history.htm
3 https://developer.mozilla.org/en/JavaScript
4 http://msdn.microsoft.com/en-us/library/72bd815a.aspx
5 http://blogs.law.harvard.edu/doc/2011/01/30/learnings-from-the-browser-wars/
6 http://www.zdnet.com.au/a-history-of-web-browser-wars-339312301.htm
7 http://www.ecmascript.org/
8 http://www.crn.de/security/artikel-8538.html, http://www.cenzic.com/resources/hackinfo/owasp-top-10-2010/, http://www.troyhunt.com/2010/05/owasp-top-10-for-net-developers-part-2.html
9 http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf
10 http://projects.webappsec.org/w/page/13246920/Cross%20Site%20Scripting

exploitation of Web-Application servers. Server-side XSS attacks are designated as follows – Persistent and Non-persistent XSS attacks. The research on these XSS subclasses in the security and scientific communities is well established. In distinction to this, the other branch of XSS attacks, client-side XSS, is stated to be incomprehensively documented, imposed upon the fact that the research information: conventions, approaches, best practices, Cheat Sheets etc. are scattered and discontinuously represented. As the major representative of client-side XSS attacks should be considered the DOM-based XSS attack. In this paper, we describe this subclass as follows: definition of the operational flow, areas of application, illustration on significant case studies, estimation of the impact factor of DOMXSS. Moreover, we illustrate some suggestions for sanitization approaches on specific implementations of DOMXSS.
In order to scrutinize the basic features and behavior, as well as some existing scenarios for DOMXSS Attacks, the structure of this project thesis proceeds as it follows: Chapter 2 focuses on the general structure of XSS Attack Vector, its separation on subclasses, including Persistent (stored), Non-Persistent (reflected) and DOMXSS attacks, and gives brief information about their specific characteristics and malicious nature. A complete analysis of the DOMXSS attacks, on behalf of the Strategical/Tactical/Operational Model, will be performed in Chapter 3 . Furthermore, Subsection 3.2.5 illustrates some known examples and case studies of DOMXSS, together with the factual course. Possible remediation techniques, such as implementation and extension of Regex(s), proper application of JavaScript patterns and filters etc., will be specified in Chapter 4 . Besides, it presents a theoretical approach on what more could be applied to the problem, in account of Function Flow and Data Flow Analysis. Chapter 5 will propose some important aspects for future work on the problem, while some of them will be outlined in appropriate parts in the exposition of the paper. In Chapter 6 , the conclusion on the topic will be presented.

1.1 Limitations of the paper

In this project thesis, we should explicitly outline the following limitations in our research and thesis exposition:

- Language requirements:
  - as given in RFC 2119[11]
  - For the purposes of this paper, we shall use He, His, which should be understood as she/he, her/his, respecting the lexicological relation to terms like intruder, attacker, user etc.
- As we discuss DOM-based XSS on JavaScript client-side Web-Applications, we shall not evaluate DOMXSS implementations, including CSS and XML.
- XML DOMXSS attacks require in most cases discussions on Web Services Security.
- If we provide limited information on the mentioned above languages, this we should utilize only to respect the consistency of the exposition.
- We use the term Web client to distinguish precisely the group of Web-Browsers, RSS Readers etc., from the other types of User Agents as: Robots, Crawlers, Spiders etc., respecting the RFC 2616, Section 14.43[12]; see also footnote[13].
- Note, that the proposals, described in our paper are not approved on strong mathematical level, which is considered as an appropriate scientific level, and are valuable topic for future work.
- In some cases, we should outline conventions and patterns, with their appropriate references, without demonstrating them in exhaustive manner.
- We should also allow us to demonstrate interesting and important techniques and taxonomies, as related works, which are considered by their developers as not completed.
- Every aspect, which is a good candidate for a future work on the topic, should be outlined in the appropriate stage of the paper exposition and explicitly in Chapter 5.

11 http://tools.ietf.org/html/rfc2119
12 http://tools.ietf.org/html/rfc2616#section-14.43
13 http://www.user-agents.org/

2 General approach to DOMXSS and limitations of the paper

In this chapter, we explain, what is DOM, in order to define it as a term and how it associates with JavaScript. Furthermore, a taxonomy of the XSS attacks will be presented, as we divide the main class (XSS) on three general subclasses: Persistent, Non-Persistent and DOMXSS attacks.

2.1 What is DOM?

The development of JavaScript is as controversial as the Browser evolution on itself. The so called "Browser wars" arouses the utilization of two different formulations of the scripting language. As mentioned in the introduction part, on one hand, Mozilla Foundation maintains JavaScript, which implementation is applicable almost everywhere, on the other, Microsoft supports the JScript, applicable only on the MS Internet Explorer. Furthermore, we outlined, that this paradigm inevitably leads to lack of standardization, concerning the proper scripting language implementation by the Web-Dev community. For these reasons, the W3 Consortium decides to propose a unifying language-independent convention, a cross-platform model, more specific the Document Object Model[14], which includes a solution, how to solve the gap between the two different scripting language dialects. DOM presents an interface between JavaScript and JScript. Moreover, in order to use both languages on behalf of HTML-enabling dynamic characteristics of the Web-content, DOM represents an instrument in the hands of the Web-Devs. The kernel in the structure of DOM is the "document" Object. Its methods grant the implementation and execution of functions in both of the scripting languages. This is well illustrated in Figure 27, see Appendix.
Nevertheless, there are still differences between the realization of W3C DOM specification, which prompts a challenge even today, to respect and to compound the specifications of JavaScript and JScript.
The lack of standardized solution hampers the Web-Dev community to implement a proper scripting code in modern Web-Applications until nowadays. As already outlined JavaScript and JScript are not fully compatible with the ECMA-262 Standard, please see Chapter 4 .
This unavoidably arises the question about the security aspects of the released code in Production environment.
The other fact, that the complexity of the Web-Applications is vastly growing, stigmatizes modern Web-Apps, as extremely attractive for malicious activities.
In the next section, we should explain more detailed the different attack techniques on modern Web-Applications and HTML based content.

2.2 Types of XSS Attacks

As we outlined in the introduction section, the XSS attack vector splits into three main branches, mainly: Non-Persistent, Persistent and DOM-based XSS attacks. These types dispose, firstly regarding their art of execution: server- or client-side. Secondly, they differ in their efficiency, as well as their discoverability and countermeasures. So, before we proceed with the objectives of this project thesis, we will scrutinize in detail the basic features of the XSS attack subclasses as follows.

14 http://www.w3.org/DOM/

2.2.1 Non-persistent XSS Attacks

The Non-Persistent XSS (NP-XSS) attacks should be illustrated in our paper's exposition as the first subclass of the XSS attack vector. Since, we want to make clear how the NP-XSS works, we should describe a simple attack scenario. It proceeds in few steps, as it follows:

1. In order to execute an attack, a malicious user (attacker) should find a potential vulnerability on a site. As a vulnerability input source should be designated for example a text box, search box etc., which gives the possibility, that one can enter string (text) or an HTML-based code, an image etc., considered as an appropriate Web-content on a dynamic Web-App:
    - Find a submitted on the Web and unsanitized XSS exploit, e.g. 0day NP-XSS.
    - Use a search engine, e.g. Google source[15] and extract the URL of a currently running Web-Application, which still implements such exploit in its source code.[16]
2. In the second phase of this attack scenario, the attacker should try to force the Web server to process and execute the malicious code:
    - As the attacker locates a proper input source on the Web-App, such as input field or parameter of the GET-request, related to the URL of the unsecured site, He can start with a trial and error approach to inject malicious code.
    - Another alternative for the attacker is to launch an internal search within the site, if the specific Web-App has an integrated search engine. By simulation of a common search, the attacker can test the security filters, which are implemented on the search box, in order to sanitize the Web-Application input stream. At first, using the search engine, He does not produce any suspicious noise, but observes the error response of the system – assuming the fact, that no matter what search input is utilized, the security mechanisms on the web-server should be activated by default.
3. After completing this testing (sample-approach), the intruder gathers and organizes the feedback about the weaknesses in the security of the Web-App, with the goal of a possible attack utilization.
4. In order to increase vastly the impact of the attack, He advertises a link to the already exploited Web-App, by posting it on different blogs, forums, sending it as a spam e-mail, per VoIP (Skype) or other messenger software, such as Yahoo Messenger etc. This process can be automated and amplified, by using bots[17] for this purpose. The attacker can fulfill furthermore two different actions:
    - to make the normal[18] users to embody themselves as actual attackers ('confused deputy'[19]) and cause DoS/DDoS on the Web-App.
    - to achieve a data steal, such as: cookie theft, ID theft etc.

15 http://code.google.com/

16 Another approach is to deploy Web-Application scanners, e.g. Acunetix Web Vulnerability Scanner (Free Edition), Netsparker [Community Edition] , Qualys Freescan etc., on a particular domain (URL), which can automate the exploit detection process; this method utilizes the so called point'n'scan technique, on behalf of a list with known exploits.

17 http://cert.uni-stuttgart.de/doc/netsec/bots.html

18 We specify the term “normal user” as – an inexperienced and not security concerned user, utilizing daily Internet activities on the Web. This user can become with greater likelihood a “Confused deputy”, see CSRF[KD10].

19 http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html, see further:
https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

This abstract scenario is presented more illustrative as a process flow diagram in the Appendix part of the paper, see Figure 28.

2.2.2 Persistent XSS Attacks

The stored XSS attacks embody another subclass of the XSS attack vector. In order to clarify the methodology of the Persistent XSS (P-XSS), we will represent another simple attack scenario:

1. In the preparation phase, the attacker should find a Web site (Web-based content), on which He could post messages in HTML-based code such as: message boards, blogs, HTML-based e-mail forms etc. Before He can proceed with the implementation of a malware code, considering its proper[20] execution, one should test the security filters on the site. If the Web-App is not properly sanitized against integration of malicious JavaScript code, the intruder could make (legitimate) HTML-conform postings, where vigorous script is integrated. Its proper execution should be considered with greater likelihood, because of the continuous operating of the compromised Web-Server, which acts as a propagation environment for the malicious JavaScript code and the existence of the Browser rendering engine – interpreting HTML-conform code by default. A normal user, who has no advanced security protection, integrated within His Browser, should not be able to ascertain, that a fraudulent script is executed in background.

2. As the attacker has completed the phase of preparation and script injection, He proceeds with the process of spreading out a specially prepared link, in order to increase the impact of the attack. As it was already described in Subsection 2.2.1 , the link distribution is achieved by a dispatch over e-mail or via different instant messengers, using fake/already stolen Ids/e-mail accounts.

For the better illustration of the P-XSS workflow, refer to the process flow diagram in the Appendix, see Figure 29.

As we have already presented two fundamental attack scenarios for NP-XSS and P-XSS, we would like to describe in detail the design and general application purposes of the attacks, by discussing the differences between them. The distinguishing approach is based on the following profound criteria[21]:

- complexity of attack preparation,
- complexity of attack deployment (Reproducibility),
- application (affected entities),
- impact (damage potential) and
- discoverability.

The complexity of the attack preparation by NP-XSS and P-XSS is almost the same: the attacker has to find a currently running/0day exploit and further to search for a suitable Web-App candidate

20 We define here „proper“, considering it from the attacker's point of view, who also strives efficient and effective, intelligently implemented fraudulent code. In particular, this has nothing to do with the moral perspective, concerning the victim's damages.

21 https://www.owasp.org/index.php/Threat_Risk_Modeling

to compromise it. The complexity of deployment includes the testing phase, the utilization of XSS code injection and the consecutive propagation/spread out. While by NP-XSS, the attacker tries to locate input sources, such as: cookies, input fields (e.g. Web-App internal search engine) and GET-requests, by P-XSS, His goal is to exploit the ability to dynamically extend the HTML-based content via posting. In both cases (NP-XSS, P-XSS), the cracker[22] is proceeding further with the deployment of the attack scenario, as detecting the sanitization filters on the Web-App. If such filters are implemented, the attacker tries to escape them, starting a blinded testing[23]. He refines this iterative method, until He gains sufficient feedback, concerning the successful further implementation of the malicious XSS injection. In case of no positive results, He applies this scenario either to a different pattern in the directory structure of the same domain, a subdomain, or another detected URL. Note, that these steps could be also applied concurrent on different Web-Apps. As He succeeds to avoid the input source sanitization, the cracker injects the XSS code and can further improve the hack. Heretofore, we cannot observe extensive differences in the deployment complexity. At last, we should take in account the end stage, concerning the spread out of the attack. In distinction to NP-XSS, by stored XSS, the intruder does not need implicitly to advertise the compromised Web-App. Though, if He decides to do so, He propagates just a legal URL, because the host already includes the P-XSS in its HTML-source. This is not the case by NP-XSS, where the distributed URL implies the malicious code. On one hand, this designates the differences in the deployment complexity, on the other, it also denotes discrepancy by the level of discoverability. A legal URL, without an injection payload, cannot be directly considered as a malicious one, which impedes the proper heuristic evaluation. Thus, the discoverability of the P-XSS is limited to observation of the Web-App source by security experts, who inevitably utilize static or dynamic code-checkers for automated feedback. On the opposite side, the discoverability of the NP-XSS could demand not only code inspection, but also a complete Function and Data Flow Analysis [L2]. So, in order to detect and prevent a NP-XSS, only a source code investigation on the Web-App is not sufficient. Moreover, the attack discovery demands an extensive forensics inspection[24] of the Web-application-sever log files, made by a security professional, utilizing semi-automated approaches. Taking these facts into count, the discoverability of the NP-XSS is more complex than this of the P-XSS. We should outline, that both of the attacks cover the following server-side injections – same origin XSS[25], cross origin XSS. This designates the same level of application. At last, concerning the impact or damage potential of the attacks, both XSS types present very severe security threat on Web-application platforms.

Heretofore, we introduced the first two well-known subclasses of the XSS attack vector and clarified their differences and similarities, concerning their better illustration and understanding. Using the same pattern, we should represent the main objective of our paper – DOM-based XSS attacks, which concern especially Cross-Site Scripting injections on the client-side implementation of the Web-Application programming logic, in the next Chapter 3 .

22 In our paper, we use the term „cracker“, describing an attacker with malicious intentions.

23 In both cases, blinded testing could be utilized manually or using Web-Application scanners.

24 By P-XSS, the Forensics professional correlates at least the following evidences: the persistence of malicious code within the Web-Application source, which is deterministic, and the log file entries on the server. In distinction to this, by NP-XSS, the malicious code is just reflected by the server. Therefore, evidence is stored at most in the log files. This enforces the professional to search for other evidences, which can be correlated to the log files, with the goal to strengthen the Forensics heuristics and thus predict properly the core of the reflected XSS attack.

25 Samy Worm: http://namb.la/popular/

3 DOMXSS Attacks

In the previous chapter, we have paid attention to attacks, which exploit the technical abilities of JavaScript, but require as a jump-start a running Web server. On the contrary, DOM-based XSS presumes only the Browser engine for launching XSS attacks on JavaScript host objects[S10]. Their main representatives are the DOM-objects. These facts inevitably provoke the questions: Is a cracker capable of exploiting such JavaScript objects? Are we capable as security professionals to harden those, predict and prevent future exploits on Object-oriented level, concerning JavaScript Web-apps?

3.1 Historical reference work

Even though, DOMXSS is thought to originate from 2004/2005, the first reports about the attack are arisen in the early 2002 by Thor Larholm. For the interested reader, please check [L8].

In 2005, Amit Klein has publicized the first widespread paper on DOMXSS. The paper “DOM Based Cross Site Scripting or XSS of the Third Kind”[L3] considers the theme and make it accessible for the general public. Furthermore, it points out not only properties of the third subclass XSS attacks, but also refers to some of the fundamental differences to the other XSS types (NP-XSS and P-XSS). Based on the information back then, Amit Klein has made a basic classification, mainly: a list of cases, concerning DOM-based XSS. Let's illustrate the general XSS structure with its subclasses in Figure 1:

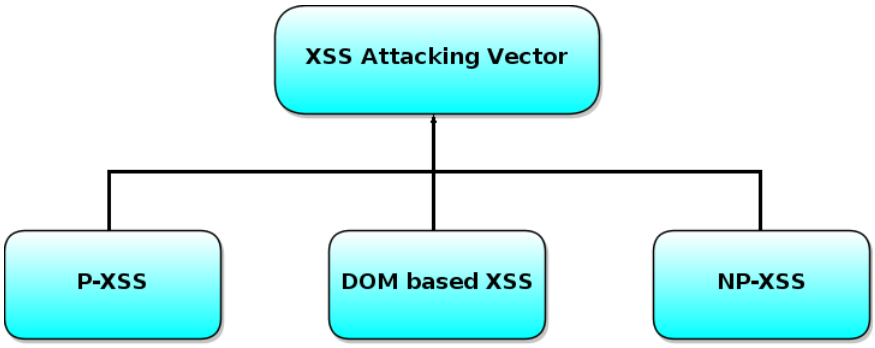


Figure 1: XSS basic classification, derived from A.Klein [L3]

In 2007, this categorization is extended and precised by Robert 'Rsnake' Hansen et al., see [JGETA07], by specifying the main DOMXSS subclasses: Persistent DOM-based XSS (P-DOMXSS) and Non-persistent DOM-based XSS (NP-DOMXSS). These inherit the features, which DOMXSS proposes, and additionally designate the core for the technical implementation, the specific DOMXSS attack is based on: if the prerequirement for the DOMXSS is P-XSS-based, we categorize the P-DOMXSS subclass, in the opposite case – NP-DOMXSS subclass.

The differentiation between NP-DOMXSS and P-DOMXSS, shown in Figure 2, is on hand of the utilized attack technique to achieve a DOMXSS exploit. In the first case, aiming an execution of DOM-based XSS, the attacker uses the methodology, applicable for NP-XSS. By advertising a specially crafted link to the potential victims, the cracker endeavors falsifying a property of a node or even extending the DOM structure with a new node with its own properties, within the document

object, loaded in the users' Browser. If the client-side logic[26] of the Web-Application is not properly sanitized, the forged document object will be conducted by the Browser engine. In the second case, utilizing P-DOMXSS, the attacker uses Persistent XSS approach. An example of stored DOMXSS, where we shall explain in detail the technical background of the attack, see Subsection 3.2.5 . Moreover, in combination with other attack techniques such as Cross-site Reference Forgery (CSRF)[KD10], Cross-site File Upload (CSFU)[L4], Remote File Inclusion (RFI)[L5][WASC10] etc., the impact of the NP-DOMXSS and persistent DOM-based XSS is vastly increasing.

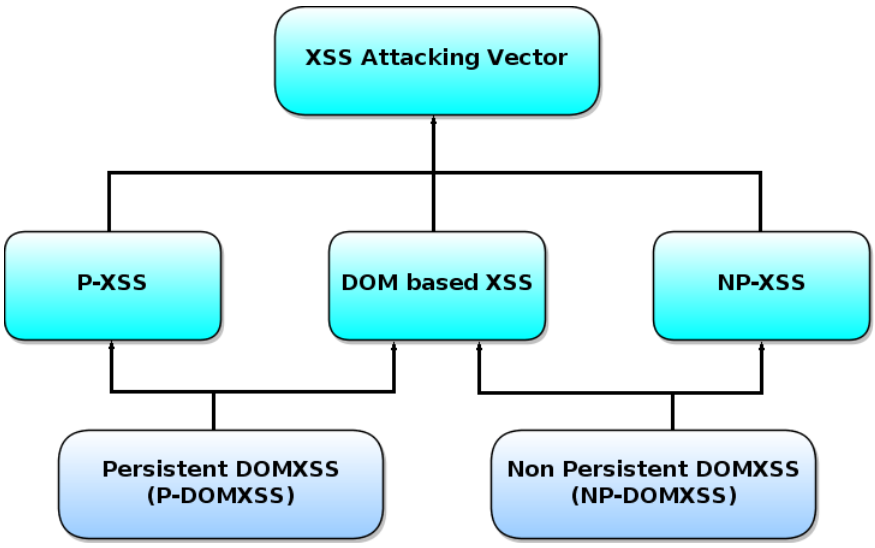


Figure 2: XSS classification, 2007, R. Hansen et al.

As Amit Klein has pointed out the basic subclasses of the XSS attack vector, in "XSS Attacks – Cross Site Scripting Exploits & Defense", the authors make classification based on the methodology, more exactly the essence and semantic of the attacks. Furthermore, they pay attention for the first time to the Function Flow Analysis (FFA) and Data Flow Analysis (DFA), considering the XSS vulnerabilities. In order to sanitize a Web-App, not only the input (input sources), but also the output (output sources) must be examined and properly defined (applying DFA), as well as the functional threads by the data processing must be carefully explored (applying FFA).

In 2007, Shreeraj Shah gives a well-illustrated approach, abstract and multilayer schemata to represent sufficient architectures of fundamental topics such as: Browser DOM engine, AJAX workflow Model, Threat Model etc.[L11]. In next year, Stefano Di Paola and Kuza55 represent the "Attacking Rich Internet Applications"[L9] and introduce a classification of sinks and sources. This extends the definitions of DOMXSS and allows more precise modular approach, which becomes a substantial part of the sanitization process.

Rafal Los [RL10] puts forward, that the "point 'n scan"-scanning of Web-Apps is not sufficient for Web-Application vulnerabilities analysis and represents a compete definition for Function Flow Analysis and Data Flow Analysis. This extends the proposals, already given by Robert 'Rsnake' Hansen et al. in 2007.

In 2011, Ory Segal [L10] enlarges the classification of Sources and Sinks by adding a new parameter, respecting the Persistent DOMXSS: Storage. On behalf of the evaluation of existing

26 This should be explicitly explained in Section 3.2.1

example, concerning the DOM-based exploit of HTML5[27] localStorage API, he proves the correctness of the separation between DOMXSS subclasses, made in 2007 by R. Hansen et al. Moreover, Ori Segal illustrates the unabated impact of these attack techniques, respecting the newest HTML Standard.

Since the history of DOMXSS can be ascertained as sophisticated, which involves implicit discussions on the terms of source code analysis, Application Flow Analysis (AFA) and further development of classifications, we feel obliged to explain the technical background of the DOM-based XSS and represent methods for source code development and analysis, as well as prevention techniques in the following sections.

3.2 Explanation of the technical background

In this section, we will describe the Threat Model, the environment, as well as the important cruxes, considering the DOMXSS attacks, in account of Sources and Sinks, URL obfuscation – encoding and decoding.

3.2.1 Environment and Threat Model

Before we depict the DOMXSS Threat Model, we should make clear, what is the general approach of an attacker and what are His main objectives, regarding Web-Applications. There are two different types of Web-Apps, respecting the E-Commerce aspect: those with implemented Business Model (BM) and such without any. Taking this matter into count, it is easy to say, that Web-Applications with a BM will be more attractive for an intruder to be exploited.

The Web-App E-Commerce deployment Model consists of two main interrelated levels – Technical (implementation) Model and Business Model. These elements can also be taken in the meaning of perception, Business interpretation: as less strongly bound – low level, and strongly bound – high level, because of their significance and internal logic, concerning the deployment and evaluation of a strong Business Plan. Considering this two layer architecture, an attacker wants to gain control over both levels, as the Business Model is with highest priority.

In order to explain the basic approach of an intruder, let us simplify the tactical scheme of the attack strategy[28]:

1. The E-Commerce Web-Application Architecture is proposed as a two-nodes-graph with bijective conjunction between the nodes.
2. The main target of an attacker is the Business Model of the Web-Application, which is accessible[29] only if the intruder controls the low level in the E-Commerce Web-App Architecture.
3. Taking both prerequirements into count, the attacker uses “divide and conquer”-approach (please, see Figure 3).

The process can be described into several steps of the four stages, Figure 3. Let's enumerate them explicitly, as follows:

27 http://www.w3.org/TR/html5/
28 Attack strategy - exploit the Business logic Model to disrupt the E-Commerce revenue.
29 Accessible - Business logic Model experiences its realization on the Web via the Technical (implementation) Model

1. The attacker has to find and exploit vulnerabilities into the Technical Model of the platform.
2. He gains control upon the low level of the Web-Application.
3. The intruder should try to disrupt the proper communication with the “Business Model”-level.
4. The attacker mitigates the bijective conjunction between the two nodes, as conquering the directed edge (Technical Model to Business Model). He disrupts the proper workflow within the E-Commerce Architecture and consequently disarranges the “Business Model”-level.
5. As a result, the attacker controls the feedback edge – from Business Model to Technical Model, which makes the last one unusable.
6. In summary, He conquers the E-Commerce Web-App, and in great likelihood any other kind of Web-Application representations.

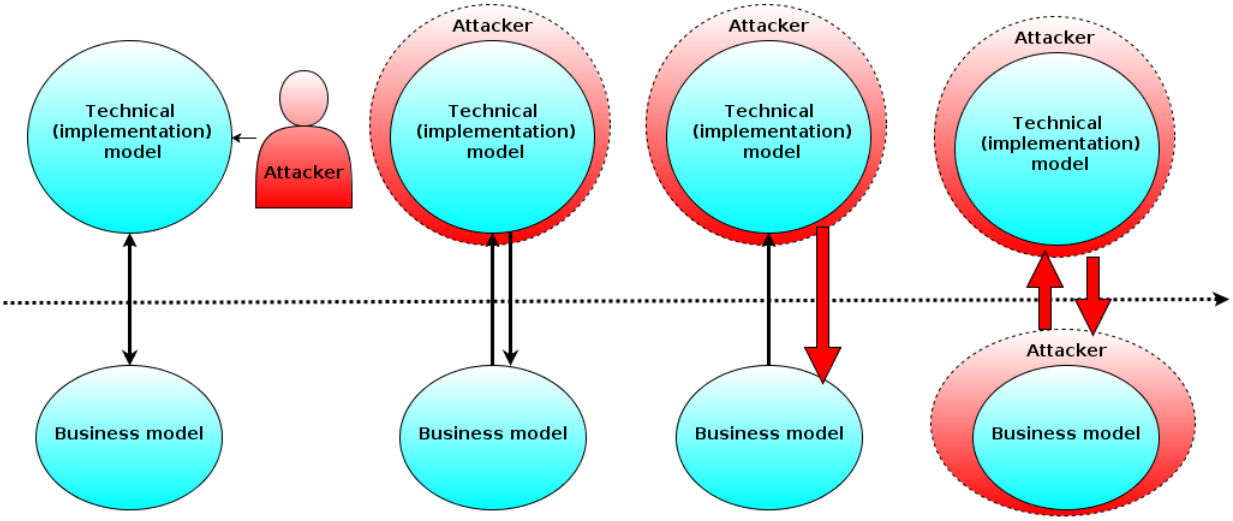


Figure 3: General attack strategy for Web-Applications

As the Business Model is getting more complex to meet the customers needs, but the periods for the release of the Web-Applications in Production environment remain the same or become even reduced, the technical implementation has to strive to keep these development rates. Thence, several important drawbacks occur:

- vastly growing load in the midrange systems – servers, due to the growing number of clients and requests to the Web-App;
- escalating concurrency into the time-to-release deadlines;
- unstructured enhancement of the existing programming code in the Production environment;
- development of programming languages for speedy application build-up;
- ignoring of established security, coding and Design patterns;
- diverge-like operating development teams, to speedy fill in the coding gaps.

These strategical changes in the Web-Application design and implementation cause numerous tactical deployments and therefore more complex operative accomplishments. Like the further development of the Web-Application Model is getting more compound, so the attacks and the Threat Models consequently adapt to it. Moreover, back in the days the implementation of the Business logic was predominantly Server-side oriented. Nowadays, we experience a migration in the Business logic implementation to the client-side, to mitigate the drawbacks enumerated above, which inevitably lead to server overload.

So, how does the modern attacks look like?

They hit on the enticing part of the Web-Application, more exactly the migrated Business logic on the client-side, because there is no need to attack directly the server, as an advantageous way to abuse the Business Model, which was a winning strategy concerning the old way of BM implementation.

What makes these new attacks conceptually so different in contrast with the “old” well-established Web attacks?

First of all, they demand advanced programming skills, including explicit knowledge of:

- Programming environment,
- semantics of the programming languages,
- programming conventions,
- Coding/Security/Design/Anti-/DOM and Browser patterns.

To summarize this briefly: the objective is not explicitly the client (user agent or end user), but the client-side implementation of the Business logic Model.

Second, a profound and complete analysis and understanding of the modules, their coherence and structural representation within the client-side environment have to be assured.

Please, check up the two examples of modern Web-Applications, demonstrating the complexity of the DOM Construction, which represent, on one hand – a modern E-Commerce B2C Portal[30], and on the other – an educational Web site. These are illustrated in the Appendix part of the paper – Examples of HTML DOM Visualizer[31], Figures 30, 31. As the reader experiences, both of the Web-Apps are highly complex. This inevitably leads to the conclusion, that a strong taxonomic approach has to be conducted, in order to deploy the already proposed Web-Application analysis. Please note, that this is executed with greatly likelihood by the modern Intelligent[32] attackers and is highly required to be awaken in the consciousness of the security professionals, as follows [Enum1]:

1. understand the fundamental structure of multi-tier SOA, concerning RIA;
2. understand the coding Implementation environment;
3. understand the Function Flow of the Web-Application;

30 http://www.akshayit.com/b2b-b2c-portals.html

31 http://www.aharef.info/static/htmlgraph/

32 In this paper, we shall propose the terms – Standard intruder (attacker) and Intelligent intruder (attacker), as follows: a Standard attacker doesn't possess personal know-how, but copies and applies well-known, published on the Web, attacks. Furthermore, He does not respect and evaluate the noise, created by the (iterative) attacks execution, and also does not consider taxonomic knowledge management. In distinction to this, an Intelligent intruder is able of creating and evaluating own attack scenarios, finding 0day-attacks and respecting the balance in the attack environment, considering minimization of the caused noise, during the (iterative) attacks execution.

4. understand the Data Flow of the Web-Application;
5. understand how to apply proper security policies to harden the preceding aspects, as mentioned in 1 – 4.

The DOM-based XSS is a good representative of such modern attacks. Let's demonstrate this, taking into count the proposed conditions, enumerated above.

At the beginning, we want to make a retrospective illustration on behalf of several figures, to explain the migration of the Business logic to the client-side and demonstrate the tight and enduring consolidation of the technical implementation with the sophisticated Business Model.

The classic Technical Model, 3-Tier Model (Figure 32), which is still usable in many Web-Applications, denotes the starting point for Business logic realizations on the Web. As well-known, it consists of: the Client Tier, Application Tier and Data Tier. The complete Business logic is integrated within the Application Tier, which explains the attack strategy – to conquer the BM, i.e. exploit the Application Tier. Some attack representatives are SQLIA[KD10], the already fully explained NP-XSS and P-XSS (please, see Section 2.2) and other, that involve as a prerequirement to exploit also the Client Tier – CSRF[33], see further [KD10].

With the reinventing of the Web[34] – the Web 2.0 evolution, the 3-Tier Model becomes insufficient, which explains the need for more refined and feature-rich model – the Service-oriented Architecture, SOA. This tendency is well-illustrated in Figure 33, where the main advantages of the SOA Model over the 3-Tier Model are enumerated.

Moreover, the unabated demand for further development of SOA and Web 2.0, engendered by the vastly growing E-Commerce and ever and anon advancing Business Model, impose a convergence between these top-level organizing principles (Figure 34).

At Figure 35 is shown the SOA Model. Compared to the 3-Tier Model (Figure 32), the reader should notice the differences between the Client Tier and the Presentation Layer – SOA implies the migration of the BM to the Presentation Layer, where the Web clients include not only HTML, but also AJAX and RIA.

The outlined migration of the BM on the client-side, is realized adequately on the level of Object-oriented programming (OOP) and DOM. Thus, the client-side coding layer is technically implemented in SOA via the Presentation layer, e.g. AJAX and RIA, as shown in Figure 35. Exactly these two components will be the main target of modern Intelligent intruder attacks. Please note, as already stated in the limitation section of the paper, the scope of our research is AJAX-/JavaScript-based DOMXSS.

To complete this retrospection, we should illustrate the end stadium, which we are aware of, and should be most probably extended in the future. The complete convergence of Web 2.0 and SOA defines a new term – Web-oriented Architecture, WOA. This definition proposes extensive features. For an instance, the Data Tier of the 3-Tier Model becomes Resource-oriented Architecture (ROA) in the WOA Model. The complete modern taxonomy is precisely illustrated in Figure 36. Concerning the basics in the technical implementation of WOA, Figure 37 gives a good overview. Finally, we would like particularly to demonstrate the complete emerging process of Web 2.0 and SOA with the explicit allocation of WOA as a derivative of this process (Figure 38).

We shall extend these tendencies with more detailed schemes, related to our scientific research on the problem. We shall proceed in top-down manner to reach the particular level of a correct

33 http://shiflett.org/blog/2007/mar/my-amazon-anniversary
34 http://www.oracle.com/us/products/applications/atg/b2b-ecommerce-reinventing-333314.pdf

technical explanation of DOMXSS.

On behalf of Enum1, we completed Enum1.1. As next, we would like to cover Enum1.2.

We should specify the SOA Web client, as the Web-Browser, due to the fact, that DOMXSS exploits the client-side Business logic via the Web-Browser DOM rendering engine. Furthermore, any kind of other SOA Web clients like RSS readers etc., based on Web-Browser rendering engines, reduces the attack complexity to the level of DOM-based XSS on Web-Browsers. Let's designate explicitly the Programming/attack environment, concerning Enum1.2, as follows:

- all kinds of Web clients, implementing different DOM-able Browser engines,
- JavaScript Programming language and AJAX,
- any type AJAX-based frameworks, JavaScript libraries,
- SOA Web-Applications based on AJAX.

Shreeraj Shah proposes a simplified scheme of WOA, pointing out in particular the Web client as a Browser with the SOA client-side BM implementations, AJAX and RIA. We shall call this scheme (Figure 39) sufficient for the technical exposition of DOM-based XSS. As next, Figure 40 gives a general overview of the Browser rendering engine and most of the integrated SOA-related implementation techniques as components. Important in this scheme is the demonstration of the basic attack workflow, following the top-down approach – DOMXSS can start[35] even on the Presentation layer, e.g. exploiting an HTML5 APIs (such example, should be demonstrated in the Subsection 3.2.5 : Attack scenarios), and proceed further with its execution on the Process&Logic layer, where we notice, that the Browser renders the JavaScript (AJAX) code and the DOM/Browser Events. If the DOM-based XSS attack scenario implies a Browser redirection to a malicious Phishing site(s), this could be achieved via the “XMLHttpRequest” (XHR) object in the Network&Access layer. In order to bypass the Same Origin Policy (SOP), the intruder should exploit the Core Policies layer.

A comprehensive attack scenario with a prerequirement – Social Engineering (SE), could be also applied to DOMXSS attacks. This is illustrated in the next Figure 41, which represents another differentiation of the Browser components, concerning the User Layer and Browser Internals layer. The AJAX and RIA, HTML/DOM Interface and UI Logic components belong to the User layer, which explains the fact, that the user is able to interact, change and modify these by His own.

Therefore, a SE is possible and amplifies not only the impact of the attack, but also the complexity of the defense and sanitization process(es) against DOM-based XSS. Furthermore, these components implicitly interact with the DOM rendering engine, which abstract implementation in a Web-Browser is demonstrated in Figure 42, from the Browser Internals layer. Hence, the already presented basic workflow is feasible.

We would like also to demonstrate the simplified DOMXSS Threat Model, see Figure 4.

To complete the Enum1.2, we should outline the JavaScript/AJAX coding specifications, more detailed in Section 4.3 JavaScript/AJAX patterns – false usage of patterns and conventions ,where we shall explain in particular the questions, concerning the related conventions and patterns. At last, let's illustrate a conclusive scheme (Figure 43), proving the proposed strategic differentiation between DOM-based XSS and the classic XSS variations, i.e. DOMXSS is accessing the client-side implementation of the Business logic Model. More precisely, the client-side components: DOM,

35 This is related especially to stored DOMXSS attacks; though, NP-DOMXSS can start directly on the Process&Logic layer of the Browser engine.

HTML5, RIA (Flash, Silverlight etc.), overlap the server-side components, as: Presentation layer, Business layer and the Data Access layer, from the Web-Application Business logic.

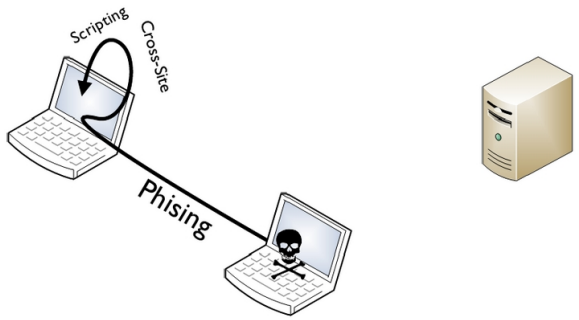


Figure 4: DOMXSS Threat Model [L13]

3.2.2 Classifications

As our goal is to represent a complete description of the technical background of DOM-based XSS, we utilize a well-known model – Strategical, Tactical and Operational Model (STO). In Subsection 3.2.1 , we covered the strategical level of the STO, explaining Enum1.1 and Enum1.2. More precisely, we outlined the strategic discrepancy between the classic XSS attacks and DOMXSS. Let's proceed further with the exposition of the tactical level, covering Enum1.3 and Enum1.4:

- understand the Function Flow of the Web-Application,
- understand the Data Flow of the Web-Application,

and how these are related to the technical implementation of DOM-based XSS.

We should deploy this, by specifying and describing the fundamental abstract constructs, considering the following simple schema:

- explaining what are Sources, Sinks and Storage (HTML5) and how are they relevant to DOMXSS
- explaining the fundamental matter of URL encoding/decoding and how this applies to DOM-based XSS, concerning the aspects of attack payload obfuscation, as well as tainted input data filtering (please, see Section 4.4)

3.2.3 Sources and Sinks

We demonstrate in Subsection 3.2.1 Environment and Threat Model, that the execution of DOMXSS passes through multiple layers in the Browser engine, which corroborate the fact, that the evaluation and analysis of the attack is highly complex. So, is it possible to find a simplified representation, a Meta-Model, which can give an answer to the basic questions, considering an attack [Enum2]:

1. Where are the vulnerabilities in the Web-Application?
2. How to exploit them?

3. Is the attack feasible or not?

As already stated, the attack exploits the client-side implementation of the Web-App Business logic Model. Let's illustrate the client-side environment in Figure 5. It is obvious, that taking into count all related aspects of the Programming environment, will make this simplified representation unnecessary complicated. Though, a Meta-Model, which reproduces the Implementation environment, can give an answer to the questions from Enum2 and sustain its simplicity and compactness. In general an attacker is essentially interested in – whether the attack works on this particular Web-App or not.

In 2005, as already mentioned, Amit Klein proposes, that there should be a third subclass of the XSS attack vector (DOM-based XSS) and outlines different tactical implementations for the successful completion of DOMXSS. In his online contribution (document), he enumerates these tactical differences in a list, and points out the characteristic aspects, considering their separation. Nevertheless, he notes that, the list is not exhaustive and will be a subject of further improvement and extension. This should be considered as the starting point for the creation of a descriptive Meta-Model, related to the tactical implementation of DOM-based XSS (see Enum2).

In 2007, the authors of [CW07] propose the terms: Source and Sink, according to the proper modular static code analysis.

These two facts are used by Stefano Di Paola, who generalizes and completes the first fundamental

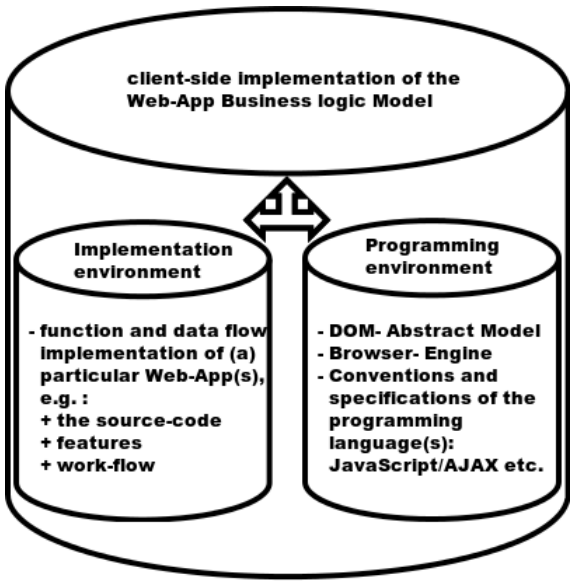


Figure 5: Client-side implementation of the Web-App Business logic Model

classifications of DOMXSS Sources and Sinks, related to the Implementation environment of Web-Applications. Hence, we should propose his classification from 2008 [L7], as a foundation for the Meta-Model, explaining the tactical characteristics of DOM-based XSS, Enum2. We should call this model the Source/Sink/(Storage)[36] Meta-Model, or S3 Meta-Model[37] (S3MM). Let's enumerate its main parameters: a Prerequiremet(s); a Source, a Sink, a possible Storage; a Payload (or just a PoC-

36 We should explain in the further exposition the existence of Storage in the S3 Meta-Model.
37 A complete illustration of the S3MM is given in Figure 67, Appendix.

Payload) and an Impact (factor). These S3MM parameters should be briefly explained in the Appendix. On behalf of these six parameters of the S3 Meta-Model, the workflow of the DOMXSS should be completely described, according to its representation on the Implementation environment of the client-side Web-App Business logic Model[38], Enum2. Let's illustrate as a graph, the basic scheme of S3MM (Figure 6).

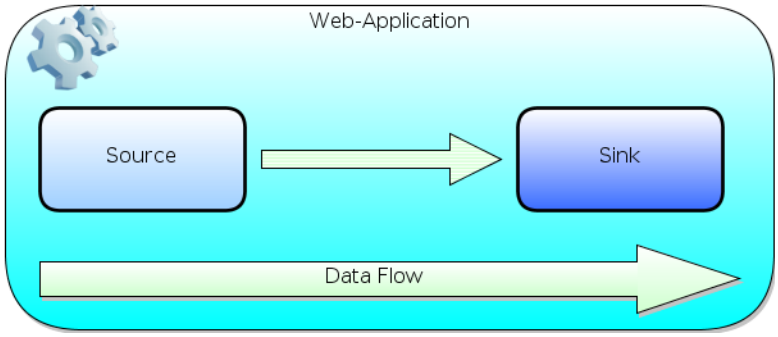


Figure 6: Processing of Source and Sink

Both – Source and Sink, should be considered as an indispensable part of the Web-Application data and function flow. In the Implementation environment (Figure 5), the Source denotes a code fragment, which embodies a vulnerable node from the DOM abstraction, implemented on the client-side of the Web-App. It represents the start node of a possible data flow, where an untrusted, malicious input data is collected and with greater likelihood acquired for processing of the Web-Application. The Sink determines the possible functional exploitability of a vulnerable source, e.g. how a processed tainted data could be exploited by an attacker and applied to the Source.

On behalf of the Stefano Di Paola's proposed classification of Source/Sink, according to their processing, there are specified three main types of Sources – direct, indirect and other object Sources, as shown in Table 1.

Let's explain briefly, how Source's types differ from one another. The direct object Sources reference to the DOM objects instantly and can be modified by the user/attacker.

As next, there are two different ways to induce indirect object Sources:

- values of direct object Sources are stored and used later to cause vulnerabilities and incompleteness of the data, or
- an already stored server side data from direct object Sources is exploited and executed by an attacker, causing a malicious input in the Web-App. [L7]

A taxonomic overview of Sinks is represented in Table 2.

As a most common and therefore dangerous type of Sinks are denoted the execution Sinks. Other important representative, which will be described more detailed in 4.4 False usage of filters, is the "cssText" Sink.

Further valuable type in this classification is the Location Sink. As stated in [L3], both "window.location", "document.location" and all their representatives are Sinks and Sources. Moreover, the Location Sink can be divided into:

38 http://blogs.msdn.com/b/simonince/archive/2008/07/18/wcsf-application-architecture-7-remote-logic-with-wcf-services.aspx

- object assignment and
- location methods .

For further information, please visit [L7].

Main types (Source)	Sub-types	Representatives
Direct sources	location/documentURI/URL Sources	document.URL
		document.documentURI
		document.URLUnencoded (IE 5.5+)
		document.baseURI
		location
		location.href
		location.search
		location.hash
		location.pathname
	Cookies Sources	document.cookie
	Referrer Source	document.referrer
	Window Name Source	window.name
	History Sources	history.pushState()
		history.replaceState()
Indirect sources	Storage Object	localStorage
		sessionStorage
		IndexedDB
		Database (Safari Only)
	Server Response Sources	
Other object sources	opener (IE <= 7 Only)	
	parent/top/frames[i].obj	
	event.data of onmessage event for postMessage method	

Table 1: Classification of sources[39]

Table 1 as well as Table 2 are derivations of the Stefano Di Paola's domxsswiki [L7], a DOMXSS Test Cases Wiki Cheat Sheet Project. It is intended to be a central place, knowledge base, related to the DOM-based XSS attack subclass. This project is still in development stage and not all Sinks and

39 For more information, please visit http://code.google.com/p/domxsswiki/wiki/Sources

Sources are represented there, e.g. see Amit Klein's list. Furthermore, domxsswiki gives a great opportunity for future work and Di Paola states, that contributions are welcome.

Main types (Sink)	Sub-types	Representatives
Direct Execution Sinks	Browser JavaScript execution sinks	eval
		Function
		setTimeout
		setInterval
		execScript
		crypto.generateCRMFRequest
	Collection	document.forms[0].action
		document.forms[0].submit
		document.forms[0].reset ...
Set Object Sinks		
HTMLElement Sinks	“document” object	document.body
		document.title ...
Style Sinks	cssText Sink	*
XMLHttpRequest Sink		XMLHttpRequest.open ...
Set Cookie Sink	“document” object	document.cookie
Set Location Sink		location
		location.href
		location.pathname
		location.search
		location.protocol
		location.hostname
		location.assign
		location.replace
Control Flow Sink		
Use of Equality And Strict Equality		
Math.random Sink		
JSON Sink		
XML Sink		

Table 2: Classification of sinks [L7]

Concerning the Storage in the S3 Meta-Model, we should point out, that this component is given in Table 1 and is specified as an indirect object Source. Though, in our proposed S3MM, it is treated as a fundamental criterion for separation of NP-DOMXSS from P-DOMXSS (please, see Figure 2). An example of stored DOM-based XSS is demonstrated in Subsection 3.2.5 Attack scenarios.

Why is so important such separation?

If a DOMXSS is represented in the S3MM and its Storage component matters for the proper attack execution i.e. that the Browser stores persistent the malicious payload, for example in the localStorage object of the HTML5 Browser API plugin. This payload can be shared between different DOM-based XSS attack techniques, which represents a cross DOM access, as stated in [L23], illustrated in Figure 44.

That's why, we propose in S3MM a hard separation between the components Source and Storage. By this way, we give a sufficient description of the S3 Meta-Model and would like to outline, that we consider further refinement of the model, regarding its properties, represented in Table 1 and Table 2, as an interesting objective for future work.

As the workflow description of DOM-based XSS, on behalf of our simplified Meta-Model is completed, we proceed with the next valuable topic for the technical evaluation of the DOMXSS on tactical STO layer in the following subsection.

3.2.4 URL obfuscation – Encoding/Decoding

In JavaScript, there are three fundamental methods, which can be used to encode (Table 3) or decode (Table 5) String values:

- escape()/unescape()
- encodeURI()/decodeURI()
- encodeURIComponent/decodeURIComponent

We shall introduce them, respecting their enumeration order.

The escape()-method returns a String in Unicode format, where all non-ASCII characters, spaces, punctuation[40] etc. are replaced with “%XX”.[41] Quite the contrary, the unescape()-method inverts a specific hexadecimal value into an ASCII String.

The encodeURI() returns an encoded String representation of an URI. In comparison to the escape()-method, it does not encode all characters[42]. An example for such characters can be seen in Table 3. In order to encode these specific characters the encodeURIComponent should be used.

Please note, that the classification is not presented in the domxsswiki project [L7], which is another topic for future work.
In the same manner, as Table 3 – Table 5 represents the general aspects, categorizing the URL decoding differences. Note, that the table is incomplete and demonstrates only a few examples of the important special character list.

40 With the exception of the following characters: * @ - _ + . /
41 The XX is equivalent to the hexadecimal value in the ASCII table.
42 The following characters are not escaped: , / ? : @ & = + $ #

Char	ASCII Number	escape	encodeURI	encodeURIComponent
!	33	%21	!	!
#	35	%23	#	%23
$	36	%24	$	%24
&	38	%26	&	%26
'	39	%27	'	'
…	...	…	…	...

Table 3: Encoding Differences, derived from [L13][43]

How URL encoding and URL decoding are related to the tactical utilization of DOMXSS?

Via the URL encoding, an intruder can obfuscate successfully the payload of the DOM-based XSS and thus pass through implemented Web-Application input filters. This increases the impact factor of the attack and also impedes the client-side Forensics, if such is applicable. For further information, please see Section 4.4 . A more advanced technique for payload obfuscation demonstrates the double URL encoding[44]. To illustrate this, we want to represent the following case study: a double encoding on non-WebKit-based Web-Browsers, see Figure 45 and Figure 46.

In [L24] is explained the principle of double encoding and a standard payload is presented in the context:

```
%253Cscript%253Ealert('XSS')%253C%252Fscript%253E
```

We applied the payload directly into the address bar of the current version of the top Browsers, which caused a NP-XSS in the following ones, see Table 4.

Browser	Tested Browser version	Browser engine	Attack successful?
Mozilla Firefox	v.10.0[45]	Gecko	Yes, produces at least 2 Popups.
IE	v.9.0.8112.16421	Trident	Yes, produces 4 Popups.
Chrome	v.16.0.912.77 m	WebKit	No, sanitized.
Opera	v.11.61	Presto	No, sanitized.
Safari	v.5.1(7534.50)	WebKit	No, sanitized.

Table 4: Address bar double encoded XSS

This explains, that NP-XSS is feasible in recent non-WebKit-based Browsers, because of the double URL encoding and automated URL completion. If such payload is applied to DOMXSS, as a value

43 We present an incomplete list.
44 https://www.owasp.org/index.php/Double_Encoding
45 In the most current version of Mozilla Firefox – v.11.0, this matter is sanitized.

of the XHR-object, most probably a large group of Web clients,specified as default for the client OS, will reproduce the demonstrated in Figures 45 and 46 attack.

It inevitably leads to the conclusion, that this issue should be discussed in detail in Chapter 4, where filtering mechanisms on behalf of URL decoding and Regexes will be proposed. At last, we encourage the testing of this exploit on mobile Web-Browsers as a future work.

Thus, we completed the discussion, concerning the tactical STO layer of DOMXSS. We would like to proceed further with the operational layer of its technical description, which is considerably more demonstrative for the reader, as some interesting examples and case studies of DOM-based XSS are explained.

Char	ASCII Number	decodeURI	decodeURIComponent
#	35	%23	#
$	36	%24	$
&	38	%26	&
,	44	%2C	,
…	...	…	...

Table 5: Decoding differences, derived from [L13][46]

3.2.5 Attack scenarios

Before we proceed further with the detailed exposition of the attack scenarios, we want to give a brief overview of the treated case studies and examples, shown in Table 6, in a reasonably arranged format.

Scenario name	Scenario type	Scenario description
Web-Goat	case study	Testing environment and educational scenario
AOL	case study	Real time scenario
Stored DOMXSS	example	Exploit on HTML5 localStorage object
JavaScript private functions	example	Ignoring the JavaScript conventions

Table 6: Scenario types

3.2.5.1 Case study: Testing environment and educational scenario

The OWASP WebGoat[47] Project framework provides a testing environment (under Windows/

46 We present an incomplete list.
47 https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

Linux/UNIX) in which different types of attacks can be executed. It is developed and can be used for educational purposes. We will describe more detailed this tool in Subsection 4.5.2 Tools, where all utilities relevant for this paper are presented. Let's advance to the DOMXSS educational scenario on behalf of the paradigm: AJAX Security::Lab:DOM-based cross-site scripting.

This scenario is presented in 5 simple steps, which are meant to be fundamental and should be considered as crucial for the security of every Web-Application, that uses/implements JavaScript.

In the first step, the goal is to use the input field, which is unsanitized, in order to embed an image element into the DOM of the Browser. In the input box, concerning the user name, is inserted the code as shown in Figure 47, which enforce the Web-Application to show a picture from the address– “http://localhost:8080/WebGoat/images/logos/owasp.jpg”. Here, we should mention, that an intruder can manipulate, specify arbitrarily the image URL. Among the billions of pictures on the Web, there are many with different inappropriate content, which can ruin the reputation of a Web-App (E-Commerce, Educational etc.). Furthermore, the attacker can even alter the “src” attribute value, to redirect the request to a Phishing Web site.

```
<img src="http://localhost:8080/WebGoat/images/logos/owasp.jpg" />
```

The modified DOM can be seen in Figure 47.

Going further with the “adjustment” of the DOM Construction, the second stage is about the execution of an alert screen (Pop-up). This is accomplished, as we keep on exploiting the input field of the vulnerable Web-App and modifying the <img> tag (shown in the code box below). Therefore, in the “onerror” attribute is implemented the JavaScript function “alert()”, e.g. “alert('P0Wnd')”. Please, note that the “src” attribute is not given in W3L-comform style in order to impel the Browser engine to produce an error and consequentially to execute the “onerror” attribute.

```
<img src=P0Wnd onerror="alert('P0Wnd')"/>
```

The result after stage 2 is represented in Figure 48.

In the third stage, the goal is to exploit the <iframe> tag (see next code box) and execute the “alert()” function. Please, note that, it is not even necessary to close the <iframe> tag in order to trigger the malicious code (Figure 49).

```
<iframe src=javascript:alert("P0Wnd")>
```

Stage 4 denotes, how an unsanitized input field can be concealed as a fake one via JavaScript, and used by an attacker to steal passwords. By inserting the code, given in the following code box, the DOM is changed and a common Web end user could easily walk straight into the trap (Figure 50).

```
Please enter your password:<BR><input type = "password" name="pass"/><button
onClick="javascript:alert('I have your password: ' +
pass.value);">Submit</button><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR>
<BR><BR><BR><BR><BR>
```

As the user enters His password, the “alert()” function causes the opening of a pop-up window, showing the concrete value of the “pass” variable, in this example it is "P0Wnd" (Figure 7). Please, note that in this case, a cracker can modify the “onClick” attribute so, that the password can be sent to a malicious server. Moreover, He can also induce other improper behavior of user's Browser.

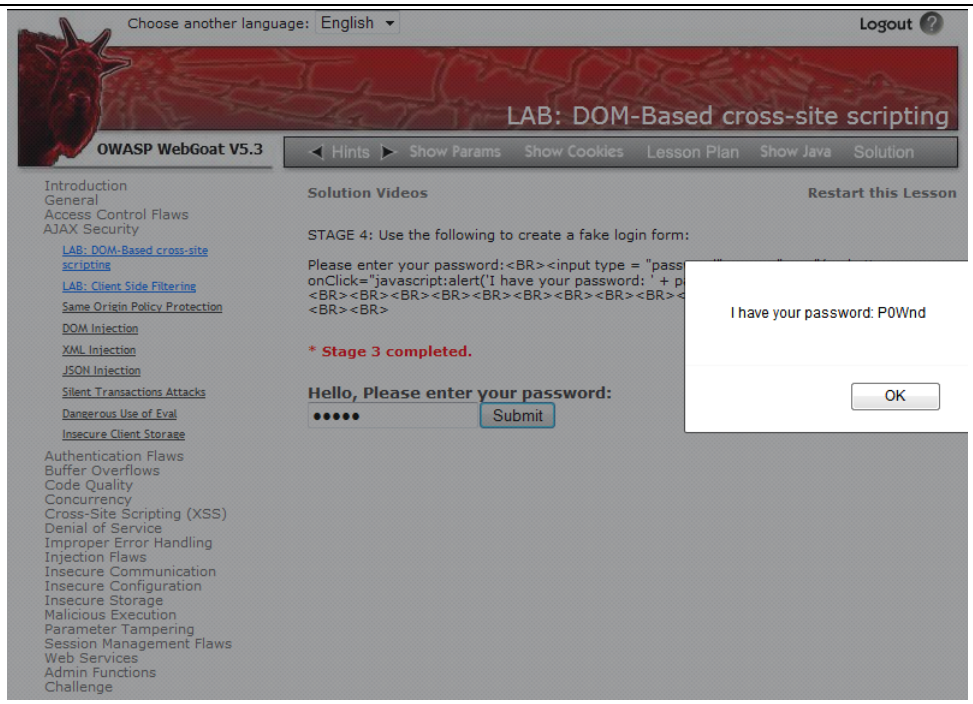


Figure 7: DOM-based cross-site scripting after stage 4

In the fifth stage, the sanitization of the input field, used in the previous four phases of this scenario, has to be initiated. For this purpose, the code of the JavaScript-file (see the code box below), which is responsible for the sanitization, has to be changed. As next, let's represent the unsanitized code.

```
function displayGreeting(name) {
        if (name != ''){
                document.getElementById("greeting").innerHTML="Hello, " + name+ "!";
        }
}
```

After short observation of the source code, we notice that the variable “name” is not properly sanitized and can be used for the malicious execution of JavaScript. To prevent this, we should use

the “escapeHTML()” function[48], as shown in the next code fragment.

```
function displayGreeting(name) {
   if (name != ''){
      document.getElementById("greeting").innerHTML="Hello, " + escapeHTML(name) + "!";
   }
}
```

After completing stage 5, we try to execute one of the attacks, that we applied in the previous phases, but as shown in Figure 51, it doesn't work anymore.

Let's denote the S3 Meta-Model for this particular case study:

S3 Meta-Model: WebGoat case study		
Prerequirements:		none
Core:	Source:	HTML DOM Text Object – textObject.name, in this case study – person.value
	Sink:	document.getElementById("greeting").innerHTML
	Storage:	none
	Payload:	Please enter your password: <input type = "password" name="pass"/><button onClick="javascript:alert('I have your password: ' + pass.value);">Submit</button>
Impact:		educational attack example

What is valuable in this case study?

First of all, the transition from the classic NP-XSS to DOMXSS is demonstrated and a security concerned user should be able to notice the difference – by injecting the malicious code, the DOM Construction dynamically changes and triggers the execution of the attack via the DOM rendering engine in the Browser. Secondly, an example of DOM-based XSS attack technique is presented with visual feedback on the testing environment. At last, a sanitization technique is proposed, which designates the WebGoat tool as an educational, respecting the Web Application Security.

As next, we will pay attention to the second scenario – real time attack scenario-AOL.

3.2.5.2 Case study: Real time scenario – AOL

What is crucial in this example?

The Aol. B2C portal still utilizes an outdated AJAX framework to auto complete the user's search

48 Please note, that the “escapeHTML()“ function is not predefined JavaScript function, but a method in JavaScript Framework prototype.

activities. This framework is jQuery v.1.3.2 from 2009. It loads the Business logic on the client-side. Thus as the user starts typing, it traces the stack for most recent similar search queries. In distinction to NP-XSS, it is not necessary to submit the query in order to activate by a server response the attack payload. We will demonstrate this in the following exposition. To obtain a sufficient feedback at the blind scanning stage, in the Aol. case study is applied the tool DOMinator (introduced in Subsection 4.5.2). This is very suitable for illustrating a proper pen-testing activity. The application flow is divided into four stages (given in Figure 8):

1. finding a vulnerability, using DOMinator,
2. build up an attack strategy,
3. exploiting the vulnerability,
4. enhancing this vulnerability.

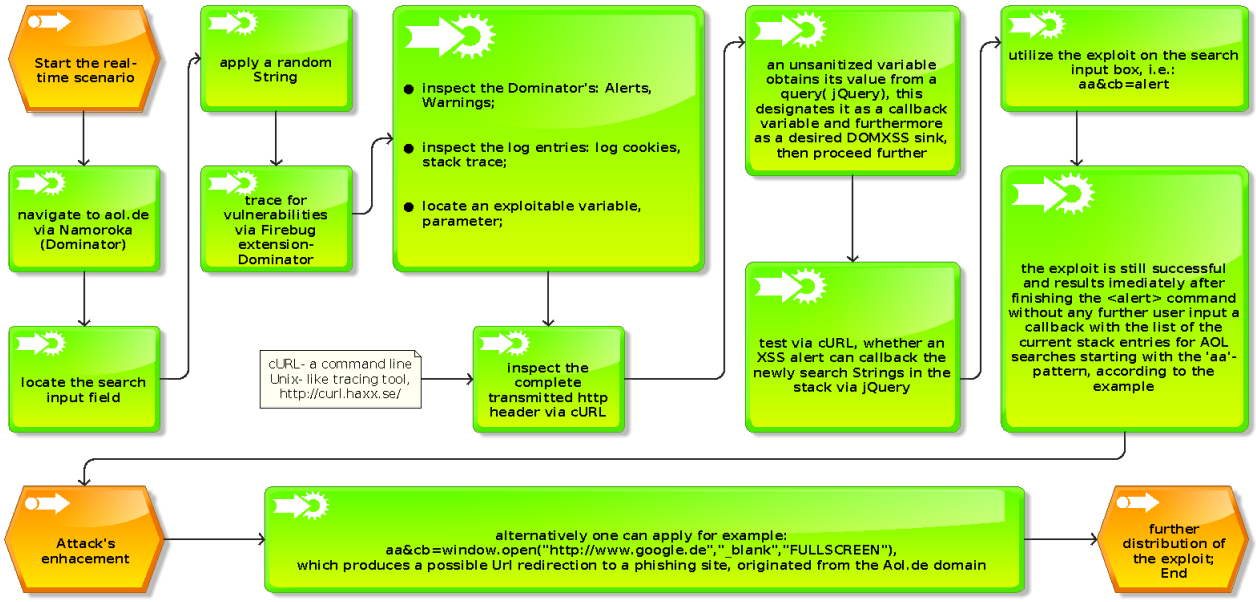


Figure 8: Application Flow walk-trough of real time scenario-AOL

The exposition of the Aol. attack test case experiences a similar approach as it was already proposed in 2.2.1 Non-persistent XSS Attacks. At the beginning, we try to find a vulnerable site – in this scenario it is the German Aol. Web site. As “aol.de” is loaded into the Browser (Namoroka with integrated DOMinator plug-in), DOMinator shows some warnings, that the page can be vulnerable (Figure 52).

In the second stage, we allocate the search input field and fill it with a simple String (in this case “aa”). In the “Alerts” field of DOMinator, which scans dynamically the alternations of the active DOM, loaded in the Browser, appears another warning, that should be inspected to refine the feedback about the Web-Application weaknesses.

We notice that, there is an exploitable variable ”cb”, which is colored in black in the “Alert” tab of the right pane of the DOMinator UI. In order to determine, how this variable obtains its value, we examine the complete HTTP-header via cURL[49] (Figure 9).

49 cURL – command line Unix-line tracing tool. For further information, please see http://curl.haxx.se.

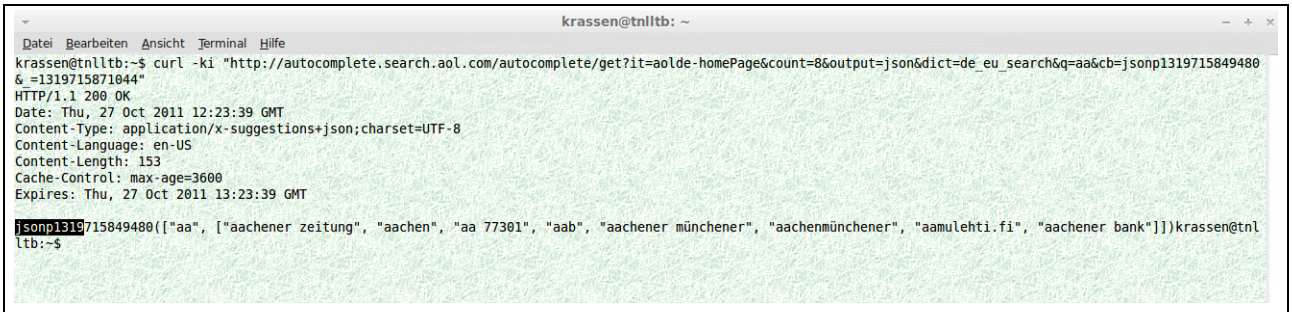


Figure 9: AOL scenario stage 2 – using cURL (step 1)

So, let's demonstrate, how we do that[50]. We copy the absolute URL, provided by the "Alerts" field and marked from DOMinator as vulnerable. Then, we utilize the command line tool cURL and activate the following options:

- "***-i/--include***", which applies the complete HTTP-header in the output with parameters like server-name, data of the document, HTTP-version etc.;
- "***-k/--insecure***", which enables the cURL explicitly to perform insecure connections and transfers, for more information, please consider the cURL man page.

We discover, that beside the common header information, the output includes also the JSON-stack content (see Figure 9). Furthermore, we should manipulate the original absolute URL, as replacing the prefix in the JSON-token with a standard XSS proof of concept function – "alert()", and recheck once again weather the output should provide the current content of the JSON-stack, considering the latest AOL searches (Figure 10).

Figure 10: AOL scenario stage 2 – using cURL (step 2)

Obviously, as the screenshot presents the implementation of the "alert()" function is successful, which approves the fact, that the exploited "cb" variable provides a callback indeed. This encourages to proceed with the application of the implicit exploit in stage 3 (Figure 11) on the AOL search input field.

As Figure 11 shows, the attack is successful, which illustrates, that an intruder can reproduce the

50 This attack is originally developed by Stefano Di Paola, http://www.youtube.com/watch?v=f_It469LUFM

JSON-stack content in a different output – Popup-window, than the regular one – drop-down list, according to the AOL Web-Application Business logic design. Furthermore, this could be extended by an intruder to a more severe attack scenario, as shown in Figure 53.

In stage 4, the “alert()” function can be replaced by another function, such as “window.open”, which enforces the Browser DOM to produce a new Browser window with an URL redirection to an arbitrary chosen domain by the attacker, e.g. phishing site. The particular code fragment is presented in the code box below, which specifies a redirection to the German domain of Google Inc.

```
aa&&cb=window.open("http://www.google.de","_blank","FULLSCREEN")
```

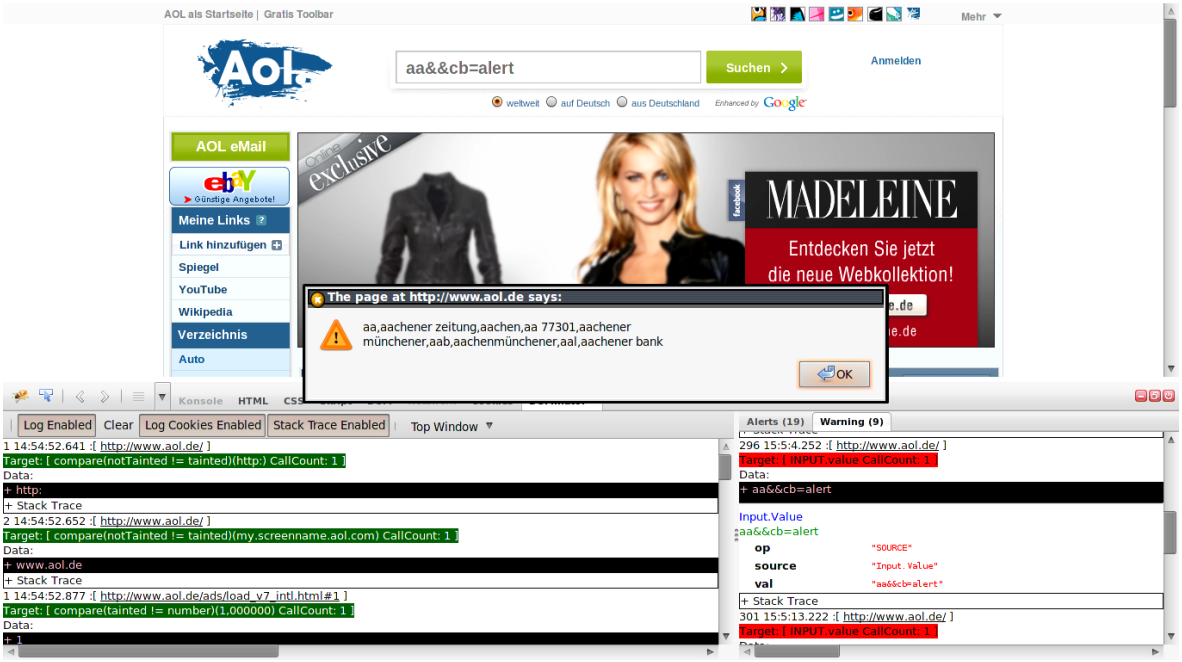


Figure 11: AOL scenario stage 3

Notice: in the original attack, described by Stefano Di Paola, the “cb” variable is returned by jQuery function, which in the scenario, presented in this paper, is done by the JSONP function from the Global search autocomlete v.1.3 module, in jQuery v.1.3.2.

Let's represent the S3 Meta-Model for this particular case study:

S3 Meta-Model: aol.de		
Prerequirements:		none
	Source:	Input.Value
	Sink:	jQuery v.1.3.2 (19.02.2009, out-of-date)

Core:	Storage:	none
	Payload:	aa&cb=alert and aa&cb=window.open("http://www.google.de","_blank","FULLSCREEN")
Impact:		high, this case study concerns a B2C E-Commerce portal DOM-based XSS injection and shown above the attack can be easily extended to a more severe one.

We propose another DOMXSS scenario (Globo), similar to this of Aol., which can be seen in Figures 54, 55, 56 in Appendix. Furthermore, the reader can compare the S3 Meta-Models of both scenarios, in order to gain an idea for the similarities and differences in their concept.

3.2.5.3 Example: Stored DOMXSS

We illustrate an example, demonstrating the persistent DOMXSS. In distinction to the previous case studies, for the successful attack completion, the Storage component of S3MM is required. This attack represents an exploit on the HTML5 Browser API “localStorage” attribute. To utilize iterative operations on the client-side, HTML5 grants a temporal container, a Web Storage[51]: “Storage” interface, “localStorage” attribute and “sessionStorage” attribute. Note, that its properties are meant to be with a global scope. This explains the previously mentioned fact, that such objects with shared properties should be considered as highly alluring for Intelligent intruders.

Let's denote the S3 Meta-Model for this particular example:

S3 Meta-Model: Example – Stored DOMXSS		
Prerequirements:		none
Core:	Source:	document.URL
	Sink:	elem.innerHTML
	Storage:	window.localStorage.name
	Payload:	<value_of_name_parameter>
Impact:		high

The related coding snippet to this attack example is illustrated in Table 7.

51 http://dev.w3.org/html5/webstorage/

```
<script>
//…...
var pos = document.URL.indexOf("name=") + 5;
var yourName = document.URL.substring(pos, document.URL.length);
decodeURI(yourName);
window.localStorage.name = yourName;
}
//.....
</script>
<!--...-->
<div id="header"></div>
<script>
   var elem = document.getElementById("header");
   var name = window.localStorage.name;
   elem.innerHTML = "Hello," + name;
</script>
```

Table 7: Code fragment of vulnerable to P-DOMXSS Web-Application [L10]

3.2.5.4 Example: JavaScript private functions

This is a very important example of permanent misuse and disrespect of the JavaScript conventions in recent applications. Mario Heiderich deliberates in 2011, [L25], a DOMXSS exploit on a private JavaScript function, which is still in usage: "__defineGetter__"[52]. This function is applied on an existent JavaScript DOM object and defines: if the object is called, which function or object method should respond to this call. In this particular example, "__defineGetter__" is applied to the "document.cookie" object. The goal is to protect against any alternation of "document.cookie". By reason that the "delete()" function is predefined in JavaScript, its call bypasses this object protection and resets the ephemeral state of the cookie.

In JavaScript, functions with leading and/or ending double underscores are designated as private functions, which are best candidates to be classified as deprecated and not applicable in newer version of the particular application. Therefore, it is highly recommended, that their utilization should be avoided, respecting the JavaScript conventions. Nevertheless, such private and protected JavaScript functions are largely used in modern Applications/Web-Applications. This represents a security risk, because the support and maintenance of deprecated functions is minimal to none, i.e. they just become neglected.

For more information on JavaScript conventions, please consider further reading of 4.3 JavaScript/AJAX patterns – false usage of patterns and conventions and [S10].

We want to represent the related coding sample, illustrating the attack in Table 8.

52 https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/DefineGetter

```
<script>
   document.__defineGetter__('cookie', function(){
      alert('no cookie access!');
      return false;
   });
</script><!--…--><script>
   delete document.cookie;
   alert(document.cookie)
</script>
```

Table 8: DOMXSS on private JavaScript function [L25]

Let's represent the S3 Meta-Model for this particular case study:

S3 Meta-Model: JavaScript private functions		
Prerequirements:		none
Core:	Source:	document.cookie
	Sink:	delete();
	Storage:	none
	Payload:	delete document.cookie;
Impact:		high, unauthorized access on predefined property

With the demonstration of these important and interesting DOMXSS scenarios and examples, we concluded the technical description of the DOM-based XSS on operational level. Furthermore, we achieved the evaluation on the technical aspects of this XSS subclass on behalf of the STO Model.

In addition to this, we should outline the importance of the proposed S3 Meta-Model as follows.

This simplified scheme gives a normed overview on the essential parameters, concerning a DOMXSS attack scenario. The {key,value} pairs in S3MM encourage the representation of the Model as a database (DB) in 3NF.

The key values in the DB should be: a_id; prerequirements; Source, Sink, Storage and payload; and impact. Based on this pattern, security specialists can build up relevant statistics of the DOM-based XSS scenarios, considering similarities respecting a single parameter (key value) or a combination of parameters. As an example, we should suggest a selection on the “prerequirements” key, regarding DOMXSS scenarios with a minimal support (MIN)[53] of other attack techniques for successful scenario completion and a selection on the “impact” key, designated as high, or severe (MAX). Consequently, the security researcher can utilize a projection on these keys, representing

53 A more precise approach should be to substitute the key prerequirements with the following: Browser version, Browser engine version, other techniques; respectively as: b_ver, be_ver and o_tech. These should help for the more precise selection of Sources, Sinks etc., related to the specific new keys.

the core of the S3 Meta-Model separately, or in combination, as given in Table 9.

DOMXSS database: DB{a_id[54], prerequirements, source, sink, storage, payload, impact}					
Selections:	$\sigma_{MIN(DB[2]) \wedge MAX(DB[7])}(DB) = DB_s$				
Projections:	Only Sources:	Only Sinks:	Only payload:		Sources&Sinks:
	$\pi_{DB_s[3]}(DB_s)$	$\pi_{DB_s[4]}(DB_s)$	$\pi_{DB_s[6]}(DB_s)$	...	$\pi_{DB_s[3] \wedge DB_s[4]}(DB_s)$

Table 9: DOMXSS S3MM database proposal

Therefore, in the full disclosure of DOM-based XSS attacks we encourage the implementation of the S3MM. Gathering the different S3MMs from DOMXSS submitted reports, will help to extend the proposed database, see Table 9. The finally projected data from DB_s, could be easily imported in statistics tools, e.g. in ".csv" format, and stochastically evaluated as required. Furthermore, establishing such classifications, will help for the evaluation of congruent statistics, representing in a more illustrative way the trends in DOMXSS Sources/Sinks. Respecting the different Browser/Browser engine versions (releases), the results can be outlined as circle graphs (pie charts) or even OLAP cube[55]. This will ease the sanitization process of vulnerable Web-Apps.

As next, we would like to represent and discuss appropriate defensive techniques, considering the previous statements.

54 In this database, the "attack Id" is represented as the primary key "a_id", which is the first key of the database; "prerequirements" is the second key and "impact" is the 7.th key, with the name of the database – DB, and DB is in 3NF

55 http://msdn.microsoft.com/en-us/library/aa140038(v=office.10).aspx

4 Prevention and defense

We should consider two general defensive approaches against DOM-based XSS:

I. In this approach we should address the topic, covering the proper design and development of modern Web-Applications, hardened against DOMXSS.

II. In this approach we should assume axiomatically, that Web-Apps are vulnerable to DOM-based XSS and we should describe proposals, concerning client-side detection and prevention mechanisms against this attack subclass, see further 4.4 False usage of filters.

Let's start with the representation of Approach I. At first, it is necessary to outline the fundamental questions, which concern a security system in general [Enum3]:

1. Is a Security Model technically and conceptually (logically) feasible?
2. Is the Security Model applicable in a real-life existing environment (Web-Application)?
3. Does the solution solve the problem and what are the additional requirements to achieve it?

In the same manner, as we gave a sufficient description of the DOMXSS technical background (Section 3.2), we should propose a security system, related to the defense against DOM-based XSS attack, taking into count the aspects of Enum3 and the completeness of the system rules.

Let' designate the essential parameters, describing the security system [Enum4]:

1. awareness on the Web-Application environment, which should be defended, hardened, sanitized and protected – requirements, parameters and application workflow, see Figures 5, 40, 67;
2. awareness on the applicable Security Model and its definition, requirements and parameters;
3. awareness on the approaches, derived from the Security Model, applicable on the Web-Application environment – definition, requirements and parameters;
4. ability to evaluate the feedback, derived from the system deployment and application, with the objective to improve it.

In order to cover Enum4, we deploy our exposition in this chapter on behalf of the STO Model, as in Chapter 3 . Let's represent a defensive STO Model (DSTO) in the next Table 10.

<table>
<tr><th colspan="5">Defensive STO Model</th></tr>
<tr><th>Level type</th><th>Strategical layer</th><th>Tactical layer</th><th colspan="2">Operational layer</th></tr>
<tr><td rowspan="7">Realization (Level of Security Model)</td><td rowspan="7">Basic Security Model</td><td rowspan="7">Related tactics: utilization of all basic tactical aspects, because they are required for the proper Security Model</td><td colspan="2">Security Teams tasks</td></tr>
<tr><td colspan="2">Admin tasks</td></tr>
<tr><td rowspan="3">Docs</td><td>Manuals</td></tr>
<tr><td>Best practices, Cheat Sheets</td></tr>
<tr><td>Other references</td></tr>
<tr><td rowspan="2">Policies</td><td>Security Policy</td></tr>
<tr><td>Same-Origin-Policy</td></tr>
</table>

<table>
<tr><td rowspan="11"></td><td rowspan="11">Advanced Security Model on behalf of Security Strategies</td><td rowspan="11">Related tactics: find a minimal set of sufficient tactical aspects, which offer the ability, that the Advanced Security Model can improve and sustain efficient</td><td colspan="2">S3 Meta-Model</td></tr>
<tr><td colspan="2">Browser engine Model</td></tr>
<tr><td rowspan="2">AFA</td><td>DFA</td></tr>
<tr><td>FFA</td></tr>
<tr><td colspan="2">Filtering</td></tr>
<tr><td rowspan="4">Patterns</td><td>Design Patterns</td></tr>
<tr><td>Coding Patterns</td></tr>
<tr><td>Security Patterns</td></tr>
<tr><td>Anti-Patterns</td></tr>
<tr><td rowspan="2">3. party Software</td><td>Security tools</td></tr>
<tr><td>APIs/Frameworks/ Libraries</td></tr>
<tr><td colspan="2">Coaching of the Teams</td></tr>
<tr><td rowspan="3">Execution (Manual vs. Automated)</td><td rowspan="3">Related questions in term of constantly improving the DSTO Model are:
1. Which stage of the DSTO should be executed manually?
2. Which part of the DSTO can be automatized?
3. Which processes can be executed concurrently and definition of the critical sections in DSTO?</td><td>Manual</td><td colspan="2">Proper, continuous and detailed version controlling of the Web-Application/ documentation</td></tr>
<tr><td>Automated</td><td colspan="2">Proper and continuous utilization of Code/Model Checkers, respecting the decisions of the Realization level, e.g. scheduled scans of Netsparker, Acunetix WVS etc.</td></tr>
<tr><td>Semi-automated</td><td colspan="2">Forensics and Analysis of the pen-testing results; utilization of patches and updates (e.g. 3. party APIs, filters etc.)</td></tr>
<tr><td>Deployment stage</td><td>Improving and following the determined Security Development Lifecycle (SSDLC)</td><td>Continuously compare with other (released) successful SSDLCs</td><td colspan="2">Applying the appropriate decisions on every stage of the SSDLC</td></tr>
</table>

Table 10: Defensive STO Model

Let's give an example of such defensive STO Model, shown in Figure 57.

As next, we would like explicitly to outline one of the most successful SSDLCs, shown in Figure 58. Furthermore, we should like to propose a simplified representation of a SSDLC, respecting the

scope of our paper – DOMXSS attacks. The description of this model is based on a DFA*, considering the proper utilization of the DSTO Model, see Figure 12.

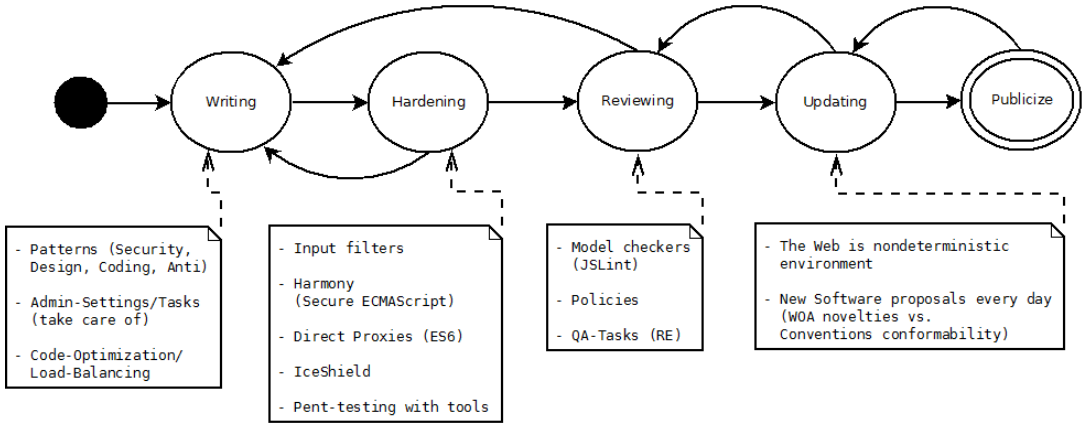


Figure 12: Security-concerned Software Development deterministic state machine (DFA)*

We should represent the DSTO more detailed in the following sections of this chapter, considering some of its crucial aspects.

4.1 Lack of sufficient collaboration between the teams[56], involved in the Defensive STO Model

Let's introduce a compact schema to illustrate the defensive workflow within an organization, or a firm, which develops, maintains, secures and gains monetary and/or knowledge-base revenue from Web-App(s), in Figure 13.

This schema does not imply detailed information about the teams, involved in the Defensive Model. If we neglect this fact, then we should consider, that even a strong DSTO Model should fail with greatly likelihood. Let's discuss this more detailed in the following paragraph.

A well-known phenomenon is, that most of the Software Applications fail to reach the stage of their implementation in Production environment, because the Requirements Engineering (RE) is not fulfilled. The responsible Team for the deployment, which controls the compliance of the Software requirements specifications, is the QA Team[57]. Furthermore, an apposite utilization of the SSDLC proposes, that the QA Team explains to the Application Developers Team the complete requirements list. Thus, a new Software Project should experience a proper start of the Development Stage. In many cases, an incorrect RE leads to logical and implementational drawbacks in Web-Application(s). Hence, it is relevant to outline this explicitly in our exposition, related to the Defense against DOMXSS, because as already stated – the main objective of this attack is to conquer the client-side implementation of the Web-App Business Model. The SSDLC also proposes, that a separate Team should evaluate the security level during every process within the Software Development Stages, until the Software Product reaches its degree of maturity to be

56 Please note, that we use the term “teams” predominantly describing Roles associated with the Software Development life cycle

57 http://www.qaap.net/blog/a-quality-assurance-team-can-save-your-company/

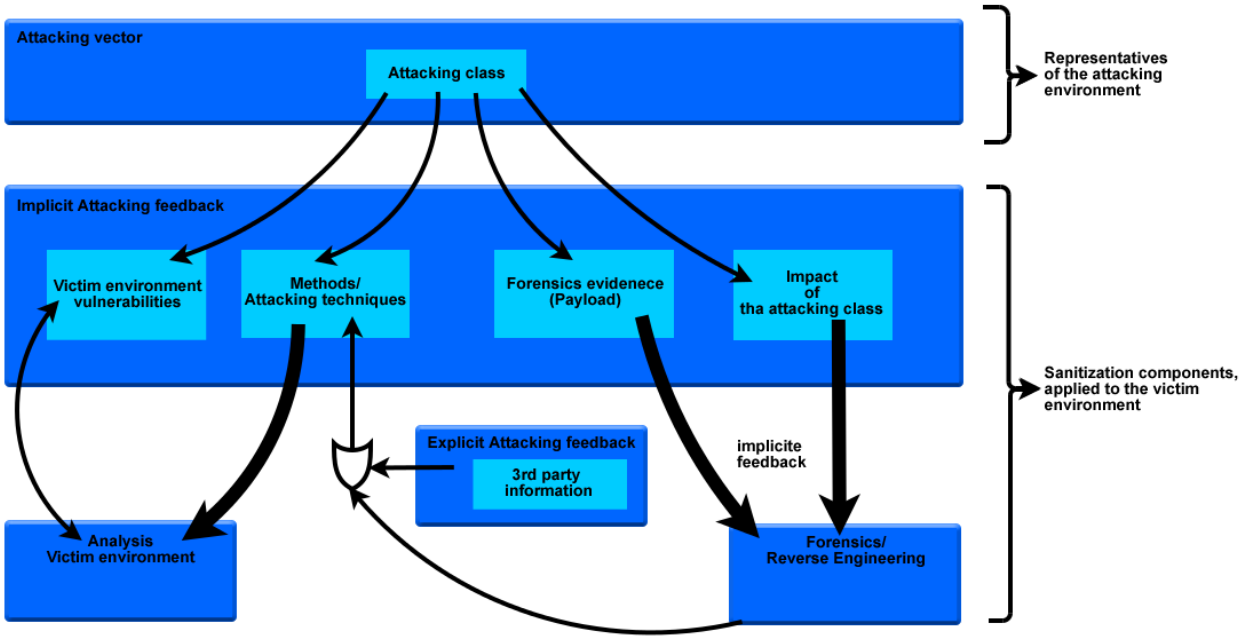


Figure 13: Sanitization components, applied to the victim environment

released in Production Environment. Consequently to an inappropriate developed Software, arises the next drawback in the SSDLC – the detection of a weak Web-Application source code in general. The responsible Team for this task is the Security Team. Not every Software Development Organization (Firm) has the opportunity to command a separate Sec-Team. We would like to stress out, that in such cases an organization should consider a revision on the implemented Source Code by an external Security Team(s) in reasonable time intervals, or to coach a sufficient amount of own associates (employees) to meet the requirements of a regular Sec-Team. Let's illustrate the involved Teams, within the Web-Application development, as designated above, in the next Figure 14.

This schema clearly states, that the Web-Application Development Environment should constantly gain feedback of the Production Environment. A reasonable representation of the Software Deployment, respecting a proper SDLC, should be specified as follows: Development Environment, Semi-Production/Testing Environment and Production Environment. Such 3-layer Deployment Environment implies a utilization of a Web-App version controlling. If the QA-Team, the Software Development Team and the Security Team represent and maintain a proper collaborative work, the already mentioned drawbacks in Web-Application(s) should be reduced considerably.

We should give detailed insight of the Sec-Team, according to the main objective of this section, see Figure 15. Please, note, that this is a general schema and could be further adjusted as appropriate.

An important question arises – how the Forensics Team could contribute to the DOMXSS related Defensive STO Model?

We are aware of the fact, that DOM-based XSS executes on a client-side. So:

How the Forensics Team should be able to collect evidence?

We should propose the following tactical approach and should call it – HoneyWebEnv. Let's describe briefly, what does this proposed technique mean.

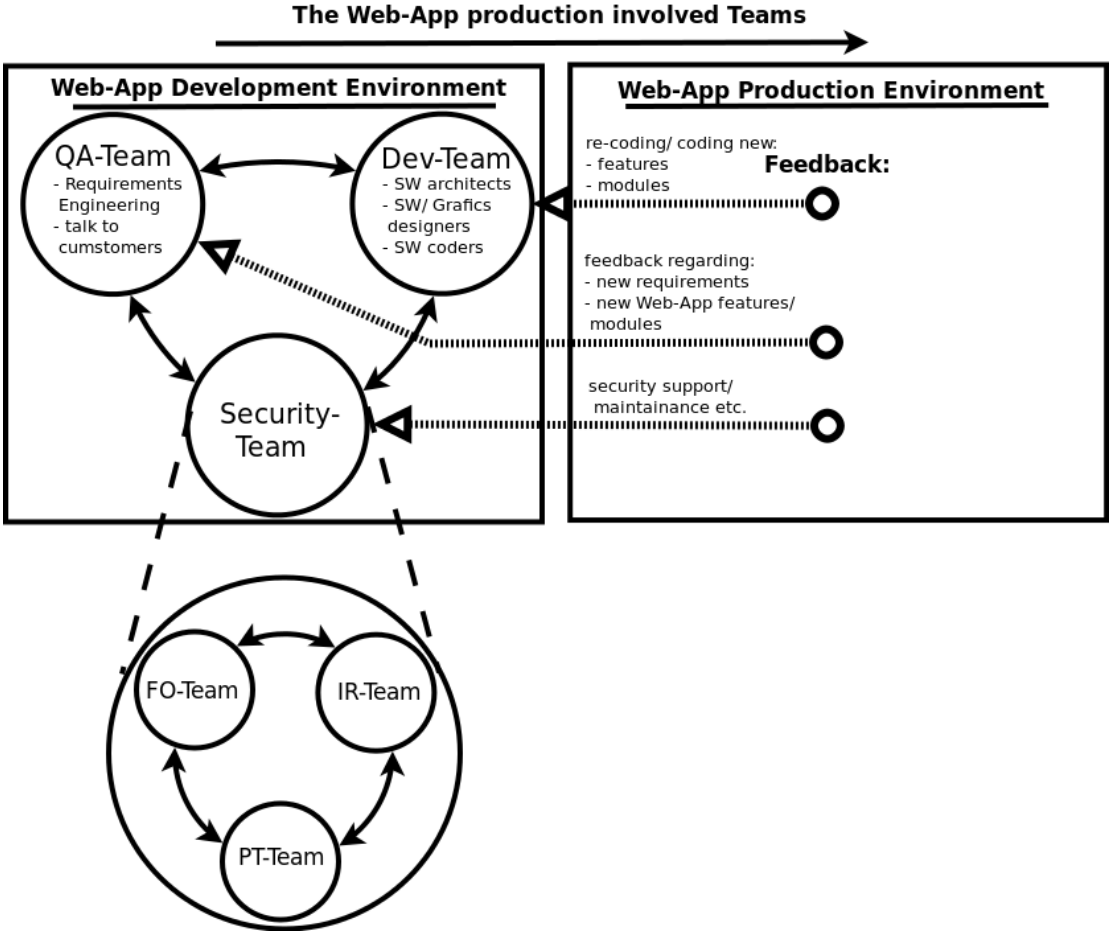


Figure 14: Involved teams in the Web-Application production

We explained in Chapter 3 the general attack strategy for Web-Applications and the "divide and conquer"-approach, considering the view of an attacker. These can be utilized also within the HoneyWebEnv, and give to the Pen-testing Team and Forensics Team the opportunity to work concurrent to the SSDLC. Therefore, the HoneyWebEnv should consist of the following modules:

- complete images of the Server[58] and Web-Application
- at least one HoneyHost(s)

Why should the complete images of the Server and Web-Application be integrated within the HoneyWebEnv?

These images should be used to gain feedback, considering the processes on the Server and/or within the Web-App. Furthermore, they should be automatically updated in proper periods of time in order to preserve the completeness and actuality, as up-to-date snapshots.

The images of the Sever and Web-Application could be utilized in background during the Development Phase and especially during the complete Semi-Production/Testing Phase, concurrently to the Web-Application development and SSDLC deployment processes, respectively.

58 A back-up image of the Server

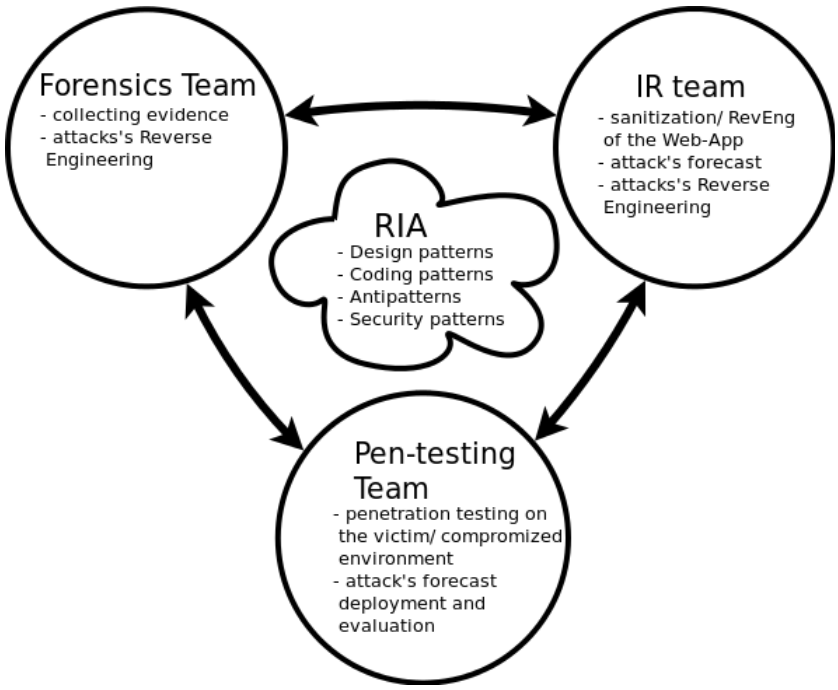


Figure 15: Structure of a security team

On one hand, the Pen-testing Team can try to break the Web-App without harming the proper SDLC workflow. On the other, the Forensics Team can gather evidence after the pen-testing activities and evaluate, whether they are related to the DOMXSS detection and sanitization or not. Special attention should be given to the source “location.hash”, as already stressed out in [MH12]. Main concern is the inability to log DOM-based XSS payloads, injected into the URL “location.hash”, like anchors on a Web site. Other important aspect concerns the privacy issues, evoked by the client-side Forensics investigation. The application of HoneyWebEnv would bypass these obstacles and thus the FO-Team can utilize properly its tasks without any restrictions.

What is a HoneyHost?

An explicitly specified local host (a PC, a mobile device etc.) should be adjusted, to collude with the Semi-Production/Testing Web-Application Environment and should simulate the default communication between a Client and the Web-App snapshot. This HoneyHost should be equipped with Top Browsers and related Browser plug-ins, regarding client-side Forensics. Furthermore, on the HoneyHost can be applied DOMXSS payloads, collected in the S3MM DB, proposed in Subsubsection 3.2.5.4 . Let's mention some of them. Considering, inspection of HTML5 localStorage property, we should outline:

- Foundstone HTML5 Local Storage Explorer v.1.1[59] (Mozilla Firefox),
- FireStorage v.1.0.2[60] (Mozilla Firefox),

Another useful plug-in for Mozilla Firefox, concerning URL Requests capturing is:

- Live HTTP Headers[61] v.0.17, with activated Request capturing, with “accurate”-level for

59 https://addons.mozilla.org/de/firefox/addon/foundstone-html5-local-storage/?src=search
60 https://addons.mozilla.org/de/firefox/addon/firestorage/?src=search
61 https://addons.mozilla.org/de/firefox/addon/live-http-headers/

POST HTTP-header requests

This HoneyHost could be also utilized in background in the same manner as the Web server image. The concurrent utilization of the HoneyWebEnv does not induce any critical sections with the deployment of the SSDLC.

At last, let's represent another proposal, a basic Sanitization workflow of the Security Team in the following Figure 16.

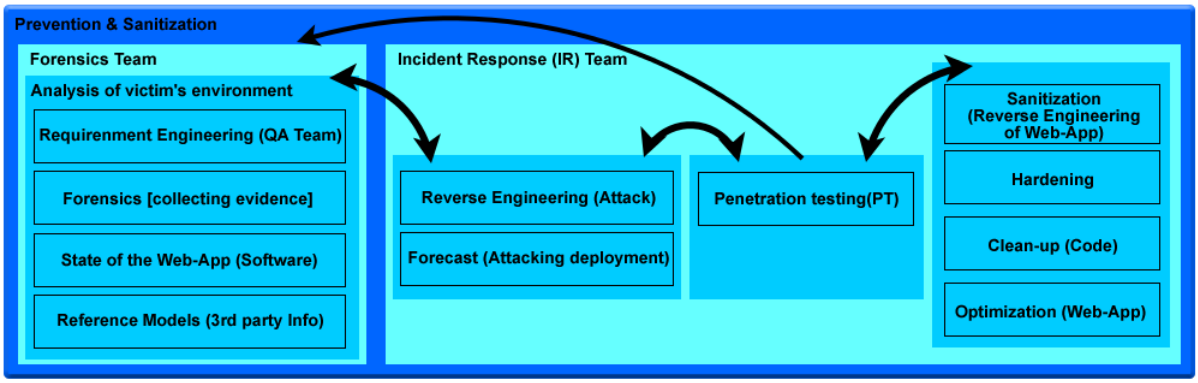


Figure 16: Sanitization workflow

As we demonstrated the complete organization between the different teams, responsible for proper and secured Web-Application development, we would like to proceed with the essential aspects, concerning the realization of Web-App(s), respecting the Implementation and Programming environment, as in Figure 5. Firstly, let's introduce a valuable approach for analysis of the Web-Application programming logic in the following Subsection 4.2 . Note, that this approach is still not fully established, as already mentioned in the Introduction chapter of the paper.

4.2 Function Flow and Data Flow Analysis

Another important aspect, respecting the prevention and sanitization of successfully deployed Web Application Attack Scenarios as a part of the Defensive STO Model, is illustrated by Rafal Los[62] in his presentation at OWASP AppSecDC in October 2010 [RL10]. Main topic of his research, concerns the Execution-Flow-based approach as a supportive technique to the Web-Application Security (pen-)testing. The utilization of Web-Application scanners (WAS)[63] should be determined as impressive, supporting the pen-testing job of the security professional/ethical hacker and not to forget the Intelligent intruder [LS09], [RL10]. Indeed, WAS can effectively map the attack surface of the Web-App, intended to be compromised. Still, open questions remain like – do WAS provide full Web-App function and data flow coverage, which reports greater feedback, concerning a complete security auditing of the Web construct in detail?

Most of the pen-testers/ethical hackers do not care what kind of functions, related to the Web-App, should be tested. If they do not know exactly the functional structure and the data flow of the Web

62 http://preachsecurity.blogspot.com/
63 As already mentioned: Netsparker, Qualys, Acunetix WVS etc.

Application, how should they consider appropriate and complete functional coverage during the pen-testing of the Web-App?

The job of the pen-tester is to reveal exploits and drawbacks in the realization of a Web-Application prior to the Intelligent intruder. Consequently to this, appears the next question – what are the objective parameters to designate the pen-testing job as completed and well-done?

Like Rafal Los states, nowadays the pen-testing of Web-Apps, utilizing WAS should be still digested as “point'n'scan web application security”. The security researcher suggests in his presentation, that a more reasonable Web-App exploit detection approach is the combination of the Application Function/Data Flow Analysis with the consequent security scanning of the observed Web implementation. A valuable comparison between the Rafal Los' indicated approach and the common security testing of Web-App(s), outlining the drawbacks of the second one, is given in Table 33, see Appendix.

Let's summarize these drawbacks, as follows.

The current Web-App pen-testing approaches via scanning tools do not deliver adequate functional coverage of modern and dynamic high sophisticated Web-Applications. Furthermore, the Business logic of the Web-App(s) is often underestimated as a requirement for the proper pen-testing utilization. A complete coverage of the functional mapping of the Web-Application could still not be approved. If the application execution flow is not explicitly conversant, the questions, regarding completeness and validity of the results from the tested data, should be denoted as open.

Therefore, Rafal Los suggests utilization of Application Flow Analysis (AFA) in the preparation part prior to the deployment of the specific Web Application scanning. This combination of the two approaches should deliver better results than those from the blind point'n'scan examinations. Explanation of this approach is illustrated in Figures 68, 69 and Tables 33, 34, 35, 36 given in the Appendix. For more information, please refer to [RL10], or consider studying the snapshot of the live presentation [L2].

Another reference work related to the problem is presented in [JMT09], [JMT11] based on [KU77], and [JMM11]. In the first paper [JMT09], the authors research the topic “Type Analysis for JavaScript” (TAJS). The ECMAScript 3rd edition, ECMA-262 Standard is covered, as the flow graphs for JavaScript are analyzed and represented in the notation, given by Ullman et al. in [KU77]. Furthermore Analysis Lattice and transfer functions are described in detail. Attention is given to the Recency Abstraction, the Interprocedural Analysis and the Termination of the Analysis. Jensen et al. outline explicitly the motivation of the project – to establish a static program analysis infrastructure, which can support the SDLC at its Development stage, because such model checkers are not successfully implemented for JavaScript. Main objectives of the paper are to deliver viable results to help programmers to detect common programming errors and even to prove the absence of such. Although, this project is not implicitly security seemingly near, the conclusions on the JavaScript code optimization should be designated as valuable, related to AFA. For the reader concerned, please consider further reading of [JMT09].

In the following project ”Interprocedural Analysis with Lazy Propagation” [JMT11], the proposed ideas in [JMT09] are extended, as the main goal is – by using the strong mathematical model of Monotone Frameworks [KU77] to prove the correctness of the Lazy Propagation technique for flow- and context-sensitive interprocedural analysis of JavaScript-based applications. As theoretical properties are outlined:

- Termination – the lazy propagation algorithm should always terminate, a proof is given in the Appendix part of the paper [JMT11];

- Precision – the algorithm sustains bottom-preserving, monotone and distributive; the proof is utilized on behalf of seven cases;
- Soundness – the modification of the transfer functions after the utilization of the Lazy Propagation Framework does not harm the semantics of the initial JavaScript code.

In the 3rd paper "Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications" [JMM11], the authors progress the research on their project to build a static analysis code checker for JavaScript, concerning the dynamic abilities of the language and the interaction between the properties of the Programming environment, Figure 5: HTML, JavaScript as programming languages; DOM-Abstract Model and the Browser-Engine.

In this project, main objectives are FFA and DFA of JavaScript applications, interacting with the HTML DOM and Browser API. Therefore, [JMM11] confirms the proposals and conclusions, given by Rafal Los in [RL10]. Moreover, the authors pay attention to the DFA, stressing out that Data Flow Analysis of JavaScript programming code cannot be analyzed separately from the HTML code-flow, which is explicitly explained on page 4, [JMM11]. The TAJS Framework and the modeling of the DOM and Browser API are described respectively in Chapter 3 and Chapter 4, where the HTML Objects, the JavaScript Events, the Special JavaScript Object Properties are discussed in detail. Note, that TAJS is developed as a plug-in for Eclipse[64].

The conclusions, concerning the implementation of FFA and DFA are clearly important, but the next valuable question appears: How to properly utilize AFA, while problems appear, even at the level, emerging the understanding of the Web-Application source code?

If we would like to distinguish proper and secure code from a weak one, we should be aware of those rules, which allow us to evaluate Software Design constructs and prove, whether the code is syntactically and grammatically correct or not.

4.3 JavaScript/AJAX patterns – false usage of patterns and conventions

At first, let's clarify the fact, that in our project we allow us to unify the AJAX Coding patterns to the JavaScript Coding patterns, because we only concentrate on JavaScript DOMXSS, as outlined in the limitation section of the paper. For more information, concerning AJAX patterns, please consider visiting [L28].

The main objectives, related to programming Software Application, arouse the following fundamental questions:

- Which Programming Languages and platforms fit best the project of concern?
- How to write properly programming code and how to review it?
- How to optimize properly programming code and consequentially to this – to complete the Web-Application as a working project?
- How to improve the modularity of the Web-Application as a Software Architecture?
- How to maintain a complex Software project to sustain its functionality, semantics and benefits?

64 http://www.eclipse.org/

- How to secure a Web-Application as a working online project in Production Environment?

The answers of these questions imply in general: sufficient knowledge on the Programming and Implementation environment, as illustrated in Figure 5. To utilize this, we feel obliged to give a clarification on the fundamental terms, concerning source code development, shown in Table 11.

Patterns	
Type	**Definition**
Coding patterns	Coding patterns "...are JavaScript-specific patterns and good practices related to the unique features of the language, such as the various uses of functions."[S10]
Design patterns	"A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context."[G94]
Anti-patterns	"An antipattern is not the same as a bug or a coding error; it's just a common approach that causes more problems than it solves."[S10]
Security patterns	"A security pattern describes a particular recurring security problem that arises in specific contexts, and presents a well-proven generic solution for it. The solution consists of a set of interacting roles that can be arranged into multiple concrete design structures, as well as a process to create one particular such structure."[S06]

Table 11: A general classification of patterns

Let's demonstrate the benefits of understanding the Coding Conventions. If we would like to write a better code, which is more understandable, we also imply the strive to write an easy maintainable code. Such includes – easy reviewing, convenient revisiting and comfortable tweaking, as explained in page 28, [S10]. Furthermore, let's explain the term maintainable code. It denotes the following features, that the code should be – readable and consistent, and predictable for understanding. The code should pretend to be written by a single person. At last, a maintainable code should be also well documented, as in [S10] and in Table 10, explaining the importance of the documentation at the Basic level of the DSTO. In many cases, code maintenance proposes Reverse Engineering of the programming code. It is sufficient to enumerate problems, escorting the RE process, as postulated in [S10]:

- time to learn and understand the problem,
- time to understand the code, that is supposed to fix the problem,
- in cases of huge and complex Web-Application projects – the person, who fixes the code (IR-Team, Dev-Team), is often not the person, who has written the code (Dev-Team), and furthermore in most of the cases not the same person, who has found the bug (PT-Team and in some cases – the FO-Team).

Taking these problems into count, we inevitably explain our proposed classifications for Web-Application, signifying the tight collaboration between the Web-Application involved Teams, as illustrated in Figures 14 and 15.

Moreover, there are other fundamental drawbacks, that assign a Software implementation on behalf of JavaScript/AJAX as tricky and hard to be sanitized, especially in our case, concerning DOM-based XSS. As mentioned in Chapter 1 , JavaScript is not fully compliant with ECMAScript. We also demonstrated in Subsection 3.2.5 , that there are still Top Applications, in which JavaScript Coding patterns are continuously misunderstood, e.g. usage of protected and private JavaScript functions and object methods.

As next, we should mention, that globals in JavaScript should be preferably avoided.

Other crucial drawbacks in JavaScript should be outlined as:

- JavaScript allows by default the usage of Anti-patterns;
- almost everything should be considered as an object.

This explains the existence of a large group of DOMXSS Sources and Sinks and as a consequence – the meaning of our proposed S3 Meta-Model. For the interested reader, we strongly recommend further reading of [S10].

Taking our last statements into count, we allow us to represent all of the JavaScript related patterns into separated tables, to demonstrate the mightiness and complexity of the JavaScript coding language.

We shall describe the JavaScript Coding patterns in Table 27, explicitly in the Appendix of our paper, because of its exhaustive representation.

Let's give a brief overview, concerning the JavaScript Design patterns in the next Figure 17, which help developers to implement proper JavaScript modules on the level of Software Architecture.

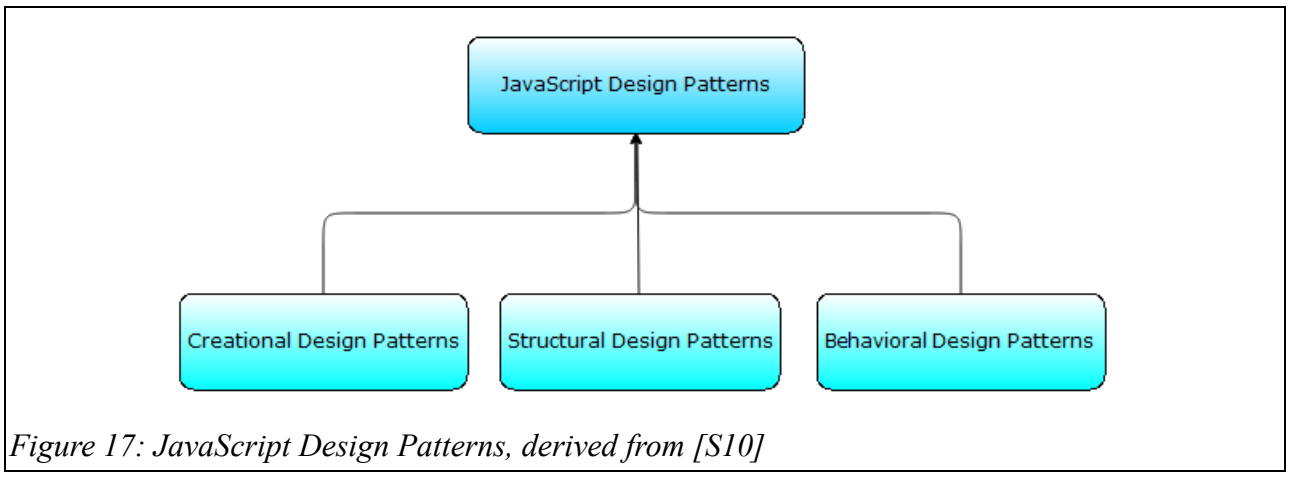


Figure 17: JavaScript Design Patterns, derived from [S10]

More precisely, the importance of Design patterns is represented by the fact, that this instrument helps developers to write maintainable code, considering its features enumerated above. Thus, a solution to the problem, described by the jargon “Spaghetti code”[65] is given. In Dev-Teams,

65 http://www.computerhope.com/jargon/s/spaghett.htm

especially when agile development[66] is considered as an acceptable SDLC approach, “Spaghetti code” seriously impedes the refractoring, the code revision and thus the understanding of the source code, and therefore the code maintenance. Furthermore, exploit detection and sanitization of 18k LOC[67] up to 450k LOC and more should be denoted as hard. On the contrary, these disadvantages are highly desirable for the Intelligent intruder. “Spaghetti code” is the most adorable environment for an Intelligent attacker to find exploits and inject DOMXSS. Therefore, we strongly recommend the understanding in depth and systematic application of JavaScript Design patterns.

For the readers concerned, we demonstrate the complete classification of JavaScript Design patterns, as in [S10], in Table 28.

Along with JavaScript Design patterns, we would like also to pay attention to the JavaScript Anti-patterns. We will illustrate two examples (please see Tables 12 and 13), which are considerably DOMXSS related. As already shown in Table 2, “eval()” is a representative of the “Direct Execution Sinks”. This function processes a String value (e.g. variable, statements etc.) and executes it as a JavaScript code.

Input	
var property = "name";	
Anti-pattern	**preferred**
alert(eval("obj." + property));	alert(obj[property]);

Table 12: Example 1 for an Anti-pattern, a well-known DOMXSS Sink [S10]

In the first case, by using the functional properties of the “eval()” function, an attacker can enforce the execution of malicious JavaScript code, by tainting the input value “property”. Such vulnerability would not be possible, if this Anti-pattern is avoided and the dynamic properties are accessed, using bracket notation, as shown in Table 12.

In the second case, the input variable is JSON text. The conversion of the data into an object can be achieved in the same manner as in the previous example – applying the “eval()” function. The problem is, that this method will turn even a non-conform JSON String into JavaScript code, causing malicious JavaScript execution. To resolve this issue, the JSON parser should be applied, as shown in Table 13, because the “JSON.parse()” function will recognize only JSON conform text and will deny all scripts within the String.

66 http://www.rakkhis.com/2011/06/agile-security.html
67 Lines of code

Input	
var jstr = '{"mykey": "my value"}';	
Anti-pattern	**preferred**
var data = eval('(' + jstr + ')');	var data = JSON.parse(jstr);
Output	
console.log(data.mykey); // "my value"	

Table 13: Example 2 for an Anti-pattern, a well-known DOMXSS Sink [S10]

As an addition to this, we illustrate a complete list of Software associated Security patterns, see Table 29 in Appendix. We shall conclude the representations of the JavaScript related patterns with the explicit deliberation of JavaScript DOM and Browser patterns in Table 14.

DOM and Browser patterns	
Class	**Instantiation**
DOM Scripting	DOM Access
	DOM Manipulation
Events	Event Handling
	Event Delegation
Long-Running Scripts	setTimeout()
	Web Workers
Remote Scripting	XMLHttpRequest
	JSONP
	Frames and Image Beacons
Deploying JavaScript	Combining Scripts
	Minifying and Compressing
	Expires Header
	Using a CDN
Loading Strategies	The Place of the <script> Element
	HTTP Chunking
	Dynamic <script> Element for Nonblocking Downloads
	Lazy-Loading
	Loading on Demand
	Preloading JavaScript

Table 14: DOM and Browser patterns [S10]

The reader concerned should notice, that in the DOM and Browser patterns, there is a specific pattern, concerning the "JSONP()" function, i.e. Remote Scripting → JSONP, see case study 2 in Subsubsection 3.2.5.2 . We strongly propose redesigning of the Aol. jQuery, according this pattern.

Enumerating the complete pattern lists is reasonable for the more deductive representation and their easier assimilation. Though, understanding the context of the patterns and their proper usage, cannot solve the problem – how effectively to utilize them in programming code and furthermore how to write proper and secure JavaScript-based Web-Applications. It is a matter of constant exercising and evaluation of the code, considering the deliberated aspects heretofore. Consequently, arises the next question – is it advisable to use 3rd party frameworks and APIs with already sanitized and secured functions and methods? A discussion on this matter will take place in the next section of the paper.

4.4 False usage of filters

One of the most crucial steps (phases) of the sanitization process is the design and proper configuration of Web-Application filters. These can be implemented, as follows:

- using common (predefined) ECMAScript methods;
- advanced methods, based purely on Regexes;
- mixed methods, a combination of the predefined and advanced methods;
- supportive methods (e.g visualizing).

Before we can discuss them in detail, it is necessary to clarify, where they can be applied. For this purpose, we need to take into count the S3 Meta-Model (shown in Figure 67). Depending on the S3MM, the sanitization process has to be applied on both – Implementation environment, as well as Programming environment (Figure 5), considering the following parameters:

- Source,
- Sink.
- Browser engine on the client-side,
- Programming logic of the Web-Application,
- user interaction with the Web-App (e.g. WOT[68]).

As illustrated in Figure 18, proper filters should be implemented on the client-side realization of the Web-Application Business logic. In particular, filters should be implemented for:

- the interpretation of the DOM graph in the Programming logic of the Web-App,
- the interpretation of the Web-Application Programming code by the Browser rendering engine, i.e. DOM rendering engine,
- last but not least, cognitive GUI visualization during the user interaction with the Browser is also advisable.

Nevertheless, it is highly desirable to integrate filters also on the server-side of the Web-App. These should prevent the successful execution of mixed attacks like:

68 https://addons.mozilla.org/de/firefox/addon/wot-safe-browsing-tool/

- P-XSS propagation on behalf of DOMXSS,
- NP-XSS injection via DOMXSS,
- DOMXSS propagation on behalf of stored XSS etc.

In order to apply proper sanitization filters on the Web-App, complete AFA has to be performed. Let's discuss the core elements of the S3MM in particular the Sinks, under the scope of Web-Application filtering.

As a most common and therefore dangerous type of sinks are denoted the execution sinks. They are JavaScript functions (object methods). If their functional arguments are not sanitized properly, an attacker can exploit this vulnerability to execute malicious (JavaScript-) code. Another possible type of sinks is the cssText Sink. It is Browser dependent, meaning that the exploiting code differs according to the specific Browser, (Table 15). This vulnerability can be caused, if the css-Style declaration is not sanitized and an unescaped input is allowed.

Browser (Version)	cssText Attack Vector	Impact
Opera (v. 10.63)	-o-link:'javascript:alert(1)';-o-link-source:current	Js Exec with user click
Mozilla Firefox (v. 3x.x/ 4.x)	-moz-binding:url(//vi.ct.im/page par=val#checkbox);	Js Exec
IE (v. 7/8)	a:expression(write(1))	Js Exec

Table 15: Browser dependent cssText Attack Vector [L7]

This explains the statement, that the client-side Web-Application filters should not only protect the Programming logic of the Web-App, but also the behavior of the different User Agents, i.e. DOM rendering engines, see Table 23.

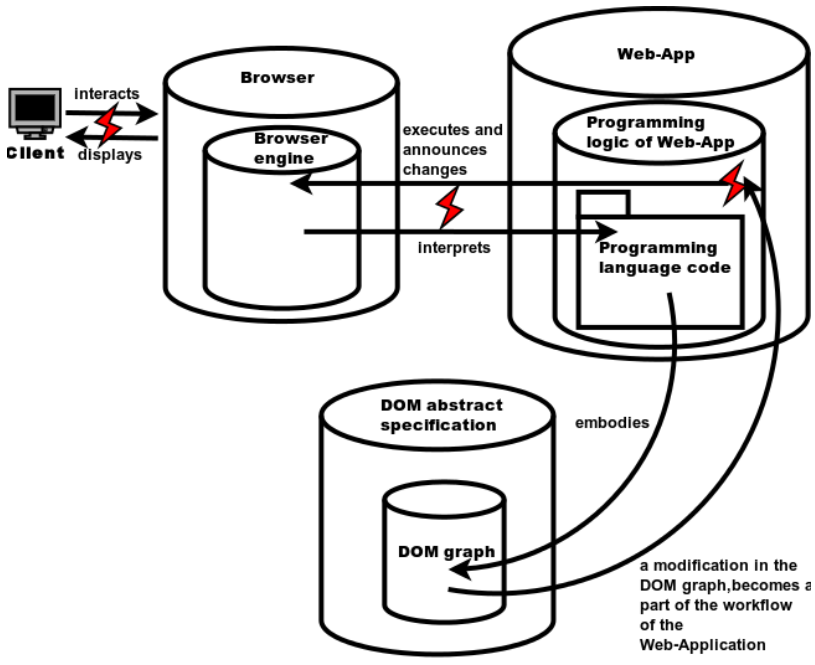


Figure 18: Client-side Web-Application filtering

Hence, more advanced JavaScript methods, which can be used for sanitization are:

- replace()
- match()
- test()

Their application is based on the deployment of regular expressions.

The “replace()” method searches for a match between a corresponding substring (or a Regex) and a string, and substitutes it. The “match()” method slightly differs from “replace()”, as it searches for a match between a Regex and a string, and returns the corresponding characters. In comparison to the first two methods, the “test()” returns not a matched string, but a Boolean value (true/ false).

Further options for Regexes available in JavaScript are Modifiers, Metacharacters, Quantifiers, RegExp Object Properties and RexExp Object Methods. It is important to outline the RegExp Object Properties, which provide modifiers for “global”, “ignoreCase”, “lastIndex”, “multiline” and “source”.

By improving the Regexes, the efficiency and the level of security in the filters for the input sanitization can be recuperated and tightened. Let's give an illustrative example, how an input for a user e-mail address can be filtered by an implementation of Regexes in JavaScript. The input form is given in the Appendix, Table 30. The user submits an e-mail in the HTML form and after the input button is activated the input String is filtered by the following JavaScript application, see Table 16.

The main advantage of such filters are, that they can be applied successfully in the case of a running Web server, as a library (e.g. JQuery) or in the case of a local HTML file, as an imported JavaScript file. This designates such filter as appropriate for input sanitization related to DOMXSS.
Let's explain the sanitization aspects of the proposed filters as follows:

1. We apply “use strict” as an ECMAScript 5 coding convention, which enables Strict Mode, concerning certain action preventions and throws more exceptions, suitable for debugging, in line 8.
2. The input String from the text area is assigned to the unfiltered variable “input”, line 10.
3. We also initialize the output variable “encodedHtml” and assign the value of the Regex to the variable “pattern”, lines 9 and 11 respectively.
4. We define the “Try...Catch” block, because of the “use strict” and to suppress Error responses in Production environment, lines 12-30.
5. As next, we apply the length restriction for the input String, in this case 20 characters, line 13.
6. We utilize the predefined JavaScript method “escape()”, which converts almost all Non-Alphanumeric characters into URL encoded form, thus JavaScript active elements are suppressed from execution, line 14.
7. The rest of the not suppressed Non-Alphanumeric characters are separately escaped via the “replace()” function, which is explained above, lines 15-19.
8. Only the “@” symbol is allowed in our filter, according to the filter specification.
9. The Regex function “test()”, see above, is applied to match the escaped input String with the “pattern” variable, our e-mail Regex, line 20-24.
10. In any case of a false result to the applied rules, see 4-9, the application responses with the minimum error response information, in this case – “Sorry”.

```
1.  /* Example for E-mail input sanitization JavaScript filter
2.  * <form> → encodeHtml, <textarea> → {htmlToEncode, htmlEncoded}
3.  * Derived from: http://www.yuki-onna.co.uk/html/encode.html
4.  * Regex, derived from: http://regexlib.com/REDetails.aspx?regexp_id=140
5.  * This code is JSLint conform!
6.  */
7.  function encodeMyHtml() {
8.     "use strict";
9.     var encodedHtml = "",
10.    input = encodeHtml.htmlToEncode.value,
11.    pattern=/^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*@[a-zA-Z0-9-]+(\.[a-zA-Z0-9-]+)*\.(([0-9]
{1,3})|([a-zA-Z]{2,3})|(aero|coop|info|mobi|museum|name))$/; //the Regex has to be improved
12.     try {
13.             if (input.length<=20) {
14.                     encodedHtml = escape(input);
15.                     encodedHtml = encodedHtml.replace(/\//g, "%2F");
16.                     encodedHtml = encodedHtml.replace(/\?/g, "%3F");
17.                     encodedHtml = encodedHtml.replace(/\=/g, "%3D");
18.                     encodedHtml = encodedHtml.replace(/\&/g, "%26");
19.                     //encodedHtml = encodedHtml.replace(/\@/g, "%40");
20.                     if(pattern.test(encodedHtml)){
21.                             encodeHtml.htmlEncoded.value = encodedHtml;
22.                     } else {
23.                             encodeHtml.htmlEncoded.value = "Sorry";
24.                     }
25.             } else {
26.                     encodeHtml.htmlEncoded.value = "Sorry";
27.             }
28.     } catch (e) {
29.       encodeHtml.htmlEncoded.value = "Sorry";
30.     }
31. }
32.
33. function testEncodedHtml() {
34.     "use strict";
35.      try {
36.         testEncodedHtmlArea.innerHTML = unescape(encodeHtml.htmlEncoded.value);
37.      } catch (e) {
38.         testEncodedHtmlArea.innerHTML = "Sorry";
39.      }
40. }
```

Table 16: Listing 2, example of JavaScript filtering code for e-mail

Note, that the proper order of the rules application is significant. If such order is not respected, especially in complex Web-Applications, which require heavy filtering and load balancing of concurrent user requests, can easily lead to a DoS. Please note, there are examples of Non-Alphanumeric XSS attacks described in [L29] and [L30]. Furthermore, output sanitization should

be also presented, as suggested in [JGETA07]. Note, that this filter does not provide any kind of log abilities. It is essential for the FO-Team to gather information about detected and sanitized malicious payloads, too. Such feature will unavoidably contribute to the filter improvement, designating a separation between escaped DOMXSS and those one, which compromise the filters. So the filtering process can be made more sophisticated and is subject of further development for the future. Based on such design, input filters can be utilized for various closed inputs like passwords, different integer inputs (monetary amount, amount of goods etc.), search on specific key words etc.
Let's mention cases, in which input filtering is not straight forward to be utilized and can be escaped by simple patterns. In Tables 31 and 32, we illustrate an input String obfuscation, applied to Web-based semantic search engines [L31]. This designates a dilemma – how to deploy proper input filtering but not to harm the semantic abilities and functionality of the Web-Application? Therefore, we should stress out the fact, that implementation of filters, based on Regexes is complicated and requires a mathematical knowledge on parsers in depth. Hereto, it should be mentioned that a huge drawback of such filtering represents the utilization of Black-listing, based on Regular Expressions. Inevitably comes the question – is there another solution as 3rd party prepared input validation filters (sanitization filters), based on strong Regexes for specific purposes?
Such filters should be constantly maintained and updated by well-trained security teams and professionals. An example of such filters is distributed by the PHPIDS[69] Project. A more detailed discussion on these filters should be outlined in the next subsection of our paper's exposition.
Heretofore, we observed filters for String input validation. Though, this should not be considered as sufficient and exhaustive. Another opportunity for applying filters in the Development stage of the SSDLC, represents the hardening of the DOM Construction. This means, that a strong Web-Application source code should not allow construction of unsanitized child nodes and tainting of the existing DOM nodes attributes. ECMAScript 5 provides a technique to control the access to JavaScript objects, called “Tamper proof objects” [L33]. The main advantages in this technique are:

- object attributes can be sealed, i.e. their further configuration is suppressed, but values can be changed,
- object attributes can be frozen, meaning that further attribute configuration and value modification are not allowed.

Another configuration option is the enumeration of the object attributes. These features contribute to the utilization of the Security pattern – Security by Obscurity (SbO). We should like to explain this more detailed. Concerning the DOMXSS, we should stress out that neither the Web-Application source code, nor the imported libraries can be obscured. This means, that the intruder is allowed to review the complete Web-App source code. So how can we deploy SbO as a Security pattern? This is possible only on the layer of the implementation of the Web-Application programming logic. In other words, if the input of a process in the application is sanitized, furthermore the DOM Construction, as proposed in ECMAScript 5, and finally the output are sanitized, there is less likelihood for possible DOM-based XSS injection. A good example of such DOM Construction filter is initiated by Mario Heiderich in the contest XSSMe² [L32]. This filter represents an integrated DOM Construction hardening approach. The contest is still active, which designates it as legitimate to our proposed DFA* Model, given in Figure 12, considering the aspects of constant improving and challenging pen-testing subject.
If such integrated DOM Construction filters represent an interesting topic for a contest, inevitably arises the question – are such filters hard to be constructed, assembled and applied in Production environment? As outlined in [L10], 14,5% of the nowadays most popular Web sites do not provide

69 https://phpids.org/

sufficient DOM Construction prevention and are vulnerable to DOM-based XSS.
So how can developers create secure Web-Applications utilizing client-side Business logic?
If the development of input validation and DOM Construction filters requires intensive Web-Application security knowledge and skills, are there any prepared 3rd party solutions, like PHPIDS for input validation, providing sufficient security of DOM Construction?
A promising project provides the ESAPI security framework[70]. Precisely, ESAPI4JS[71] concerns input validation[72] and DOM Construction hardening against XSS attacks, particularly against DOM-based XSS. Nevertheless, this project cannot be considered as completed, because of still existing drawbacks in the DOMXSS sanitization mechanisms, as outlined in [MN11]. We should consider this project as valuable opportunity for future work.

Another crucial aspect concerns the assembly of PHPIDS and ESAPI4JS with other development 3rd party libraries, helping programmers to create proper client-side JavaScript Web implementations. It is important to stress out this matter, because nowadays most of the Web-Applications utilize in the code production Web frameworks and JavaScript/AJAX development APIs. ESAPI4JS provides only support for jQuery and Prototype[73], securing the DOM Construction. Though, there are many implementations based on other JavaScript/AJAX libraries like Knockout.[74], Backbone.js[75], Dojo Toolkit[76] etc. If developers decide to utilize such libraries, they should either implement security filters manually, or find own solutions to assemble ESAP4IS in the DOM Construction. Consequentially, appear the next questions:

- Which 3rd party development library is hardened against DOMXSS, respecting the aspects of the particular Web-Application project, developers are working on?
- If a suitable for the developed project library does not provide sufficient level for client-side sanitization, which features should be manually hardened and improved?
- Is there a knowledge place, like domxsswiki, which delivers security related information and comparison between, concerning the different JavaScript/AJAX libraries?

As a related work, we should outline [JW11]. The authors propose an evaluation on an empirical analysis of the most commonly used Web frameworks, as CakePHP[77], Ruby on Rails[7879], Django CMS[80] etc., concerning the aspects of XSS filtering and especially DOM-based XSS filtering, outlined in Section 3.2 at [JW11]. The security researchers represent also a workflow Browser Model for the deployed evaluation on the frameworks. Furthermore, they deliver systematically the results, respecting the input validation and the DOM Construction aspects. We should consider this project as fundamental and valuable for a future empiric evaluation of the JavaScript/AJAX libraries, respecting the questions enumerated above.

Let's outline DOMXSS related Cheat Sheets, which provide basic security knowledge for client-side Web-Application developers, as follows:

70 https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API
71 http://code.google.com/p/owasp-esapi-js/
72 http://code.google.com/p/owasp-esapi-js/wiki/ExtendedValidationAPI
73 http://prototypejs.org/
74 http://knockoutjs.com/
75 http://documentcloud.github.com/backbone/
76 http://dojotoolkit.org/
77 http://cakephp.org/
78 http://rubyonrails.org/
79 Please not, that prototype.js JavaScript Framework is a part of Ruby on Rails.
80 https://www.djangoproject.com/

- DOM based XSS Prevention Cheat Sheet,
  https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet
- HTML5 Security Cheatsheet,
  http://html5sec.org/
  https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet
- XSS (Cross Site Scripting) Prevention Cheat Sheet,
  https://www.owasp.org/index.php/XSS_Prevention_Cheat_Sheet
- XSS (Cross Site Scripting) Cheat Sheet, Esp: for filter evasion,
  http://ha.ckers.org/xss.html

At last, we would like to enumerate information sources, which give more advanced information about XSS and DOM-based XSS in particular, as follows:

- full disclosures and PoCs:
  - security focus[81]
  - insecure.org[82]
- mostly NP-XSS, but also some DOMXSS PoCs:
  - xssed.com[83]
  - twitter.com #xss-channel[84]

At this point we should consider the evaluation of Approach I, which concerns the secure client-side Web-Application development against DOM-based XSS, as completed.

Let's process with the explanation of the second defensive approach. As postulated above, Approach II assumes, that the Web-Apps are with greatly likelihood vulnerable to DOMXSS and delivers a discussion over possible techniques and methods to secure the Web client. Let's enumerate the main methodologies, related to Approach II, as follows:

1. Description of the integrated XSS filters, concerning the different Browser user agents.
2. Discussion on Direct Proxies and a preventive implementation against DOMXSS, based upon them – IceShield.
3. A complete Browser implementation, Opus Paladium, implementing Same-Origin-Policy (SOP) against XSS.
4. Supportive 3rd party Browser plug-ins, in particular Web of Trust (WOT).

At first, we want to give an overview of the integrated XSS user agent filters in Table 17.

81 http://www.securityfocus.com/
82 http://insecure.org/
83 http://www.xssed.com/
84 https://twitter.com/#!/search/%23XSS

User agent	Filter description	
Trident, v8+	Name	IE 8 XSS Filter
	Reference link	http://blogs.technet.com/b/srd/archive/2008/08/18/ie-8-xss-filter-architecture-implementation.aspx
	Status	active
WebKit, v11+	Name	XSSAuditor
	Reference link	http://trac.webkit.org/browser/trunk/Source/WebCore/html/parser/XSSAuditor.cpp
	Status	active
Gecko	Name	XSS_Filter
	Reference link	https://wiki.mozilla.org/Security/Features/XSS_Filter
	Status	In progress

Table 17: Integrated XSS user agent filters

While the only Browser engines, Trident and WebKit, are capable of integrated sanitization against XSS, Presto and Gecko still do not provide build-in solutions. Furthermore, we evaluated MS IE XSS filter and XSSAuditor and achieved in many cases to escape those filters. Some of the test cases are provided in the Appendix part of the paper. Additionally, we should mention two 3rd party Browser extensions, which are relevant for the client-side DOMXSS sanitization, as follows:

- NoScript[85], concerning Gecko-based Browsers;
- NotScripts[86], related to WebKit- and Presto-based user agents.

NoScript provides a client-side detection and prevention against XSS, based on script-blocking mechanisms. The user can block the entire script(s) or rely on build-in regular expression rules against Cross-site scripting attempts. Further option is to manually extend the XSS white filters, which requires well-grounded user knowledge on the problem. A drawback in this filtering is the case – the client-side Web-Application logic represents an AJAX implementation. In such cases the user should allow the scripts execution completely and rely on the build-in XSS filters or manually adjusted Regexes. If the Regexes, cannot detect the DOM-based XSS, the user agent should be considered as compromised. This reflects against the dilemma – utilization of strong filters versus sustaining of the Web-Application semantics. Only well-educated and well-trained power users could utilize properly NoScript without experiencing mitigation of the desired Web-App functionality. That's why, we outline NoScript XSS Regexes as important topic for further improvement.

NotScripts should be clearly considered as a project in alpha development stage. The tool can either stop completely particular scripts on a Web site or allow such entirely. This designates the add-on as not capable of providing proper DOMXSS client-side protection. Because it is still the only tool for Presto user agents, we shall consider it as an valuable topic for future work.

As next, we should represent Direct Proxies as an instrument to access directly JavaScript host

85 https://addons.mozilla.org/de/firefox/addon/noscript/
86 https://chrome.google.com/webstore/detail/odjhifogjcknibkahlpidmdajjpkkcfn

objects. Main part of host objects are the DOM objects [S10]. If we can access the DOM Construction without triggering the Event Handler, we should be able to provide mechanisms for DOMXSS sanitisation before the DOM tree is loaded into the Browser engine. The implementation of Direct Proxies API of ECMAScript starts with the [[strawman:direct_proxies]] project[87]. Because of several drawbacks, mainly in the synchronization between the target and the proxy, this project should be considered as deprecated in the future and continued by the [[harmony:direct_proxies]] project[88] [L34].

An interesting project, based on Direct Proxies, Secure ECMAScript[89], is IceShield [MH11]. Main motivation of the project is to represent a proxy between the Web-Application and the user agent, which traps malicious DOMXSS payload on the client-side of the Web-App Logic. Moreover, it intercepts and prevents the execution of vicious JavaScript code. The tool can access the DOM Construction directly in a sandbox environment, before the parsing of the DOM tree by the Browser engine is conducted. Thus, IceShield can simulate object property tainting and also detect already exploited elements of the DOM Construction and sanitize them, by inline code analysis. Using the object methods from the Direct Proxies API “seal()” and “freeze()”, the tool can modify the DOM tree and reflects to the user agent a properly filtered snapshot of the original Web-Application. Hence, client-side DOMXSS input validation and DOM Construction hardening are achieved. There are three implementations of IceShield – a server-side filter, a network proxy and a Browser plug-in. Furthermore, the tool disposes of a DB with known DOMXSS representatives, which it can pivot during the detection process upon. Moreover, the security researcher(s) apply methods for detection of 0day DOM-based XSS culprit(s), based on machine learning and Fisher's linear discriminant analysis (LDA). As a result, the network proxy representation of the tool performs a high detection accuracy rate of 98% and furthermore detects three 0day attack attempts. The performance evaluation results sustain lower than the human response time. Though, as mentioned above these results do not represent the Browser plug-in performance abilities. Let's enumerate explicitly the limitations of IceShield, as follows:

- The tool should be further evaluated, concerning plugin XSS [L25], if the intruder does not use native DOM methods to inject the malicious payload, but exploits the client-side logic via Adobe PDF, Adobe Flash, MS Silverlight, MS ActiveX or Java applet etc., heuristics of the tool could not predict properly the outcome of the JavaScript code execution.
- The IceShield researchers outline, that the tool is lacking of tamper resistance support for older Browser engines.
- Further research should be utilized, concerning fingerprinting of timing attacks.
- It is important to estimate the performance of the tool implemented as a Browser plug-in.
- Another interesting topic for evaluation of IceShield is to test the network proxy realization against simulated DDoS DOMXSS attacks, to calculate the performance in a concurrent attack environment.
- The tool represents a rate of 9,6% partially hampered usability issues, as for example banner ads are incorrectly displayed. Considering the aspects of E-Commerce, this is a matter for further evaluation and optimization.

We should consider this tool as very promising and a great opportunity for future research.

87 http://wiki.ecmascript.org/doku.php?id=strawman:direct_proxies
88 http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies
89 http://wiki.ecmascript.org/doku.php?id=ses:ses

As next, let's represent a full Browser realization of client-side XSS filtering. The “Opus Paladium” Browser, based on the WebKit user agent, implements XSS sanitization on behalf of Same-Origin-Policy and the XSSAuditor filter [ST10]. The Browser is a Software implementation, derived from the project Alhambra. We decided to evaluate the tool against classic XSS as well as DOM-based XSS. At first, we want to describe the installation process and after that deliver the estimated results. Current version of “Opus Paladium” is OP2 and the source code can be found in [L35]. Installations routines[90] are provided for Mac OS X and Ubuntu Linux. We tested the installation on Ubuntu 11.04, Natty Narwhal. As the Browser uses the QtWebKit[91], the following packages should be additionally installed on the system: flex, gperf, qt4-qmake, libqt4-core, libqt4-gui, libqt4-dev, libsqlite3-dev. The current package version is “opbrowser-release-2010_06_19.tar.gz”. The root installation folder contains three scripts, which should be executed in the following order:

- “build-webapp.sh”, which downloads the most current version of QtWebKit and prepares the sources for compilation on the system.
- “build.sh”, which compiles the sources on the local host machine and prepares the project for execution.
- “run-op2”, which starts the application.

The installation process takes time, but after successful compilation the project should be considered as ready to use. We evaluated the tool with five different techniques on six Web sites, as represented in Table 18.

These results designate the project as not fully completed implementation against XSS, and more specific against DOM-based XSS. Nevertheless, we shall consider the OP2 secure Browser, as a challenging topic for future work.

At last, let's mention a supportive project, which provides a visual, cognitive feedback to the user, considering the trustability of a Web-Application. Web of Trust[92] (WOT) offers a safe browsing experience, respecting the following parameters: trustworthiness of the Web site, vendor reliability, privacy and child safety. Moreover, WOT utilizes an estimation of the results on behalf of the Web-Application linkage to other domains, adopting their results. Different users improve the WOT trustability DB dynamically. The plug-in is available for all user agents. The WOT user interface implements an impressive warning surface, which flags straight forward a Web site to the user as vulnerable or not. Though, this plug-in does not utilize the accessing technique like IceShield. Thus, if the DOM tree is vulnerable, it should be rendered by the Browser engine, despite of the warning.

We shall consider this plug-in as valuable for future development, because of its persuasive cognitive representation, which designates it definitely as a user-friendly UI.

Herewith, we conclude the exposition on Approach II.

90 http://code.google.com/p/op-web-browser/wiki/UbuntuInstall
91 http://trac.webkit.org/wiki/QtWebKit
92 http://www.mywot.com/

Case study	Domain name/Payload	Result
immediate(real-time) DOMXSS	aol.de	False negative
	aa&cb=alert	
	globo.com	False negative
	<img src=a onerror="alert(1)"	
	gsmarena.com	False negative
	<img """"><script>alert(1)</script>	
NP-XSS and access to host object property document.cookie	mit.edu	False negative
	http://student.mit.edu/catalog/search.cgi?search=<IMG """"><SCRIPT>alert(document.cookie)</SCRIPT>">	
NP-XSS and access to host object property escape the same-origin-policy	mit.edu	False negative
	http://student.mit.edu/catalog/search.cgi?search=	
Same-Origin redirection on an E-Commerce Web-Application	real.de	False positive
	After selecting a redirection to the online E-Commerce shop via the JavaScript implemented selection menu, the Browser refuses to render the content.	

Table 18: Evaluation of OP2 secure Browser

Thus far, we discussed aspects, related to the defensive SDLC, considering the Dev-Team, IR-Team and FO-Team. Let's represent tools, supporting the tasks of the PT-Team for discovering of vulnerabilities and hardening of Web-Applications against DOM-based XSS.

4.5 Related tools

In this section of the paper, is given a brief overview, concerning pen-testing tools, which are implicitly DOMXSS related. Moreover, we propose a collection of simple Tools' Requirement Rules, which should be considered as a fundamental and vital part for the further development and evaluation of PT-tools[93], applied to the DOM-based XSS attacks.

4.5.1 Tools' Requirements Rules (TRRs)

Before we represent in particular the different tools, we should like to propose a collection of Tools' Requirements Rules (TRRs). It should help for more appropriate categorization, outlining the following aspects:

93 Pen-testing tools

- giving a standardized estimation of the Tool, concerning its successful application on proper analysis, whether the Web-App is vulnerable to DOMXSS or not;
- giving an insight to the tool's developers, how to improve their testing utility;
- giving a requirements' list to developers, which plan to build new scanning tools;
- giving an insight to security researchers, to chose the most appropriate PT-tool for their purposes;
- giving an insight to security professionals, which tools in combination can give sufficient and more exact evaluation on behalf of the Web-Scanning;
- giving an insight to security researchers, what is valuable in general for Web-Scanning.

TRR Nr.:	**Explanation/ Definition**
1	Scanning all JavaScript and HTML files of a Web-Application (CSS and XML)[94]
2	Detecting Sources & Sinks & Storage (S3MM)
3	pivot against well-known DOMXSS-es & 0days attacks
4	detecting tainted input + output
5	Detecting the dynamically alternation of DOM
6	Reporting ability (visualizing)
7	Automatic Update of Regexes and Tool's DB
8	Support URL decoding/encoding (protection against double encoding)
9	Client-side Forensics ability
10	Detection and reporting of 3rd party Frameworks, APIs etc., applied to the Web-App
11	Browser compatibility
12	Automated execution/Tool scanning scheduling and reporting ability (logs, e-mails etc.)
13	Debugging feature and reporting ability of errors/warnings, produced during the tool's workflow
14	Tool's documentation
15	User Friendliness
/16/	Admirable, if the tool can utilize AFA: DFA and FFA

Table 19: TRRs

Hence, we propose this TRRs' list as a collection, we should explain its characteristic requirements as such:

1. As a collection, the rules should be weighted.

94 It is desirable to check also all CSS and XML files, but it is out of the scope of this paper (please, see the limitation section of the thesis)

2. All TRRs should be related to the subject.
3. No redundancies in this list, no tautologies should be presented.
4. The collection of the TRRs should be compact.
5. This collection should be approved for its completeness.

Table 20 represents the complete TRRs' matrix of the tools, discussed in this project thesis.

Legend	
Weight of feature	**Description**
x	has not this particular feature
?	not known or lack of documentation, if it has this particular feature
-	feature is not fully implemented and requires further development
+	has the feature

TRR	**Static**	**Dynamic**			**Mixed**		**Educational**
	DOM XSS Scanner	WebScarab	DOM tracer	DOMScan	domsnitch	DOMinator	WebGoat
1	-	x	-	-	+	+	x
2	+	x	x	x	x	x	x
3	x	x	x	x	x	x	x
4	x	x	?	+	+	+	x
5	x	x	+	+	+	+	x
6	+	+	+	+	+	+	+
7	+	x	x	?	?	+	x
8	x	x	x	x	x	+	x
9	x	x	x	x	x	x	x
10	x	x	x	x	x	x	x
11	+	+	-	-	-	-	+
12	x	x	x	x	x	x	x
13	x	x	x	x	?	-	x
14	-	+	-	-	-	+	+
15	-	-	-	-	-	+	+
/16/	x	x	?	?	x	x	x

Table 20: TRRs matrix according to the particular tools

We propose each TRR as related to a proper DOMXSS scanning and consider no tautologies in between. Although, at this stage of our research, we cannot assure against collisions, produced by concurrent application of the TRRs to the tools. We evaluate the Web-scanners as already developed products and we do not take part in the designing of a new PT-tool. Furthermore, we do not allow us to call this collection a complete TRRs' list. At last, we haven't explicitly outlined the related parameters for the evaluation of the TRRs' prioritization. Therefore, we highly consider this TRRs' collection for further development and approval, and a subject for future work.

As next, we proceed with the representation of DOMXSS scanning tools in particular.

4.5.2 Penetration Testing Tools

The PT-tools, which are used for the purposes of this paper can be seen in Table 21, as they are differed according to the following aspects:

- utilization objectives,
- current stage of development,
- Operating system requirements,
- implementation interface,
- specific implementation platform.

Before we depict the common properties, characteristics and functionality of the tools, it is necessary to introduce a simple classification for the better understanding of their applicability. Based on the way of operation mode, the DOMXSS tools can be divided into three main categories and a mixed category, shown in Figure 19, enumerated as follows:

1. Static Code Checkers (14,3%)[95]
2. Dynamic Web-App Scanners (42,8%)
3. Educational Tools (14,3%)
4. Mixed tools (28,6%)

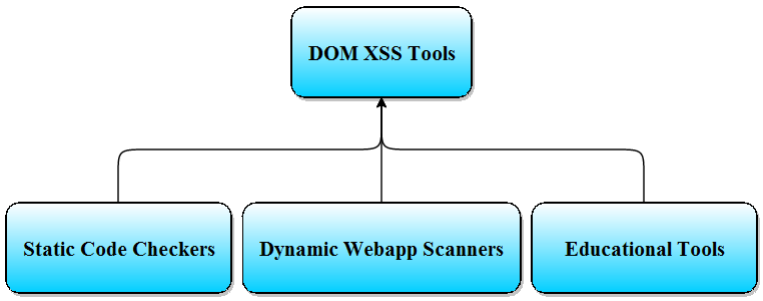


Figure 19: DOMXSS Tools

95 appearance in the tools' collection

In the next Table 21, every single representative of the DOMXSS PT-tools is specified precisely by the following criteria:

- tool name and current version,
- OS Support,
- Implementation interface (UI type) and specific implementation platform (e.g plug-in, binary etc.),
- utilization objectives and development team (company), maintaining the tool.

Tool name	Current version	OS Support	Implementation Interface	Specific Implementation Platform	Utilization' Objectives	Developer (Company)
DOMinator	v0.9.0.4	Windows/ Linux	Browser-based	Mozilla Firefox plug-in	Web-Application scanner for DOMXSS vulnerabilities	Minded Security
DOM XSS Scanner		Windows/ Linux/ Mac OS	Browser-based	Online Scanner (Browser independent)	Source code scanner, that points out all Sources and Sinks of a Web-Application	Gòmez, Ramiro
domsnitch	v0.717	Windows/ Linux/ Mac OS	Browser-based	Chromium extension	Dynamic Web-App scanner, an Add-on for Chrome	Vasilev, Radoslav Google Inc
DOMScan	v1.0	Windows	Browser-dependent	IE	Utility, which can be used for dynamic analysis	Blueinfy
DOM Tracer	v1.0	Windows/ Linux	Browser-based	Firefox extension	Dynamic Web-App Scanner, an Add-on for Firefox	Blueinfy
WebGoat	v5.2	Windows/ Linux	Browser-based	J2EE Web-Application	Web-App, developed with educational goal. On hand of visualized examples are shown the most critical security threads.	OWASP & Aspect Security
WebScarab	v1631	Windows/ Linux/ Mac OS	API	Java framework		OWASP

Table 21: DOMXSS Tools list

Representatives in the classes of the Static Code Checkers, Mixed tools and Educational Tools should be respectively denoted as follows:

- DOM XSS Scanner (static)
- DOM Snitch (mixed)
- Dominator (mixed)
- WebGoat (educational).

The Dynamic Web-App Scanners has the biggest ratio in the proposed categorization:

- DOMScan,
- DOMTracer and
- WebScarab.

4.5.2.1 Static Code Checkers: DOM XSS Scanner

In distinction to the other tools presented in this paper, the DOM XSS Scanner (Figure 59) is the only one, which is OS and Browser independent. This tool is an online scanner, that can be used during the development, testing and the sanitization process of a Web-App, concerning the detection of possible unsecured Sources and Sinks.

We should stress out, that this should be the standard procedure during the implementation and hardening phase of the code development, see Figure 12.

More detailed, DOM XSS Scanner parses through the source code of the Web-Application including all HTML, XML and (internal/ external[96]) JavaScript files and creates a full list of all detected Sources and Sinks. This simplifies considerably the hardening and sanitization in the development or maintenance phase. By visualizing the possible Source and Sinks within the Browser GUI, the IR-Team and the Dev-Team can inspect the related elements for malicious input/output data tainting, and consequentially apply AFA, precisely DFA.

Although, we should outline, that this tool has a severe drawback. We achieved a stored reproduction of reflected XSS to a compromised Web site, after utilization of scanning with the tool. We provide a full disclosure on our 1st and 2nd 0day in the Appendix of the paper. We suppose, that it does not implicitly impair its DOMXSS scanning abilities. Though, it is advisable, that the tool's maintainer/owner should reconsider inspection of the scanning filters and we highly recommend source code sanitization of this online scanning tool.

Another drawback of the tool represent the implemented Regexes for Sources and Sinks respectively.

Sources
/(location\s*[\[.])|([.\[]\s*['"]?\s*(arguments|dialogArguments|innerHTML|write(ln)?| open(Dialog)?|showModalDialog|cookie|URL|documentURI|baseURI|referrer|name|opener|parent| top|content|self|frames)\W)|(localStorage|sessionStorage|Database)/

Sinks
/((src|href|data|location|code|value|action)\s*['"\]]*\s*\+?\s*=)|((replace|assign|navigate| getResponseHeader|open(Dialog)?|showModalDialog|eval|evaluate|execCommand|execScript| setTimeout|setInterval)\s*['"\]]*\s*\(/

Let's revise the first Regex, concerning the detection of DOMXSS Sources. Inevitably, the reader concerned can notice the collision, while pivoting against the keyword “Database”. In the HTML5

96 This meets the aspect of Cross origin policy sanitization

“Storage” object attributes are “localStorage”, “sessionStorage” (W3C) and concerning the Safari Browser, the “Database” attribute. In domxsswiki, these attributes are classified as Indirect Sources. Though, as DOM XSS Scanner implements a static code checker, which traces the complete source code of the Web-Application, false positives to pure HTML-based text representatives are noticed, because the Regex cannot determine the difference between a plain text entry and the Indirect Source “Database”. The scanning of the following URL produces therefore 439 false positives:

http://en.wikipedia.org/wiki/Database

Finally, we should like to consider, that DOM XSS Scanner is not providing the appropriate level for pen-testing, especially for inexperienced Security researcher. Therefore, we recommend redesigning of the applied regular expressions and hardening of the DOM Construction of this tool as a Web-Application scanner. This is important project for future improvements.

4.5.2.2 Educational Tools: Web Goat (IDE/Framework)

WebGoat is an Open Source project, developed by OWASP[97], which main goal is to help not only people interested and working in the scope of Web-Application security, but also those, involved in the design, development, data and functional maintenance of a Web-App.

In its essence, this tool is a separate “fully” developed J2EE Web-Application, containing multiple security breaches, typical for the Top 10 most severe Web attack vectors. The WebGoat-user can debug those in a “step-by-step” manner. The treated security topics integrated within WebGoat platform include:

• Access Control Flaws, • AJAX Security, • Authentication Flaws, • Buffer Overflows, Code Quality, • Concurrency, Cross-Site Scripting, • Denial of Service, • Improper Error Handling, • Injection Flaws,	• Insecure Communication, • Insecure Configuration, • Insecure Storage, • Malicious Execution, • Parameter Tampering, • Session Management Flaws, • Web Services threats, which consider the WS Security[98].

Table 22: WebGoat topics

An illustration of the WebGoat GUI is given in Figure 60.

By this way, everyone involved in the creation and development, testing and maintenance of a Web-App can gain a view over the critical security aspects. Furthermore, it decreases the possibility of mistakes, driven by:

97 https://www.owasp.org

98 This is the most common abbreviation, denoting Web Services Security

- programmers' ignorance,
- disregarding the secure code programming Design patterns,
- misinformation or lack of knowledge.

Consequently, this increases the level of: stability, reliability and efficiency of the Web-Application.

We highly recommend the further development of WebGoat Framework and the implementation of more attack scenarios, concerning NP-XSS, P-XSS and especially DOMXSS. Therefore, we consider it as valuable topic for future work.

Another tool, which is not explicitly evaluated in the category of the educational tools, but it's worth to be mentioned is the HTML5 Security Cheatsheet project – "innerHTML"[99]. This tool helps security researcher to test, evaluate, and understand on-the-fly the user agent abilities to parse String concatenation, more precisely – how the Browser engine interprets the predefined HTML tags, violation to the proper HTML tags construction, parsing of undefined HTML tags etc. This inevitably helps for the process understanding, aiming the hardening of the DOM Construction, as explained in Approach I and Approach II. Let's demonstrate this on behalf of two examples:

Example	Browser version	Browser engine (user agent) interpretation	
Case 1	Google Chrome v18.0.1025.142 Gnu/Linux	**Payload**	href="><img src=a onerror='alert(x)'">works</
		Result	Results 2 x alerts of "58"
	Google Chrome v18.0.1025.142 m Windows 7	Result	Resilts 2 x alerts of "0" and 2 x Popups of "54"
	Opera v11.62 Build 1347 Gnu/Linux	Result	Resilts 2 x alerts of "<html5sec.org> 0" and 2 x Popups of "<html5sec.org> 74"
Case 2	IE v9.0.8112.16421 Windows 7	**Payload**	<iframe src="javascript:alert(document.cookie); window.open('http://www.google.de')">
		Result	Results 2 x empty alerts and opens a new Browser tab, redirecting to "www.google.de"
	Google Chrome v18.0.1025.142 Gnu/Linux	Result	Results 2 x alerts of "lang=en" and opens a new Browser tab, redirecting to "www.google.de"
	Google Chrome v18.0.1025.142 m Windows 7	Result	Results 2 x empty alerts and opens 2 new Browser tabs, redirecting to "www.google.de"
	Opera v11.62 Build 1347 Gnu/Linux	Result	Results 2 x empty alerts and supresses the redirection to "www.google.de" Popup
	Mozilla Firefox v11.0 Gnu/Linux	Result	Results 2 x empty alerts and supresses the redirection to "www.google.de" Popup

Table 23: innerHTML examples, html5.org

99 http://html5sec.org/innerhtml

4.5.2.3 Dynamic Web-App Scanners: DOMScan

DOMScan (please see Figure 61) is a dynamic Web-App Scanner. It is a separate binary representation, which detects the dynamic alternations in the DOM Construction within the Browser. DOMScan depends on the running Internet Explorer instance, which designates it as an OS dependent, i.e. MS Windows based. The DOM modifications of a specific Web-Application can be monitored in detail, e.g. collection of particular entry points and calls, which reflects the Sources/Sinks classifications. Furthermore, DOMScan can dynamically trace back JavaScript variables and their access methods during the code execution. In addition, the tool highlights the lines, where "eval()" calls and Document calls are detected.

Let's enumerate the DOMScan specific features as it follows:

- DOM-based XSS
- DOM-based vulnerable calls
- Source of abuse and external content loading methods
- Possible DOM logic and Business layer calls
- Same-Origin Bypass calls and usage
- Mashup usage inside DOM
- Widget Architecture review using the tool[100]

A serious drawback in the detection abilities of DOMScan is discovered during the tool evaluation. In distinction to DOMinator, presented in Subsubsection 4.5.2.7 , this PT-tool lacks of Sources/Sinks dynamic tainting and cannot distinguish between active DOM calls and such in comment blocks. The drawback is illustrated in Figure 62, Appendix. This is very confusing for an inexperienced security researcher. Therefore, we recommend the tool for further improvement.

4.5.2.4 Dynamic Web-App Scanners: DOM Tracer (Mozilla Firefox)

DOMTracer is another tool, that belongs to the group of the dynamic Web-App Scanners. It is an extension for Firefox Browser, and therewith OS independent. Via this plug-in can be traced the DOM and JavaScript calls of a Web-Application in real time.

This utility is a beta version, and some improvements has to be made, considering the compatibility with the new versions of Firefox (please see Figure 63). Nevertheless, we achieved via another Firefox plug-in "Disable Add-on Compatibility Checks 1.3"[101], to escape the version controlling and to assemble DOM Tracer successfully on the latest version of the Browser. Further drawbacks of DOM Tracer are produced by the multi-tab browsing, i.e. the tool parses all active tabs and window instances. All detected suspicious calls are displayed in a single DOM Tracer GUI viewer. This hampers the estimation on particular DOMXSS research, concerning specific Web-Application, because the DOM calls are produced concurrently by the different Web-Apps. Moreover, there is no cognitive Sources/Sinks highlighting as in DOM XSS Scanner, DOM Snitch and DOMinator. Especially for an inexperienced researcher, this tool provides a bad cognitive feedback. Therefore, we recommend a single instance – single tab utilization of DOM Tracer, to acquire more precise results. Nevertheless, we should denote the following achievement of the tool – the DOM Tracer GUI viewer represents the detected scripts and methods, considering the JavaScript Coding patterns

100http://www.blueinfy.com/tools.html
101https://addons.mozilla.org/de/firefox/addon/checkcompatibility/?src=search

and displays the source code in a proper formatting. This is very useful respecting the proper inspection of large AJAX/JavaScript 3rd party libraries as jQuery, Dojo Toolkit etc. As a good combination we recommend the application of DOM Tracer with DOMinator for the evaluation of a specific Web-Application.

4.5.2.5 Dynamic Web-App Scanners: Web Scarab and Web Scarab-NG

Web Scarab is another Open Source project, developed by OWASP[102]. This PT-tool has two distributions – Web Scarab (WS) (please see Figure 64) and Web Scarab-NG (WS-NG) (shown in Figure 65).
In comparison with WebGoat, which belongs to the group of Educational tools, as a Dynamic Web-App Scanner, it can analyze applications, working with the Application Layer protocols of ISO/OSI Model: http(s). Web Scarab is a Java-framework, thus it is OS independent.

The scope of application can be extended by adding several plugins (shown in Table 24) and deploy them in various modes of operation.[103] There are some differences between the two versions, which inevitably reflects on their functionality. The Web Scarab framework possesses more features than the NG version such as:

• Extensions, • XSS/CRLF, • SessionID Analysis,	• Scripting, • Fragments,	• Compare, • Search.

Though, the WS-NG is user-friendlier due to the better GUI interface, the WS is more appropriate for the purposes of this project thesis, thanks to the presence of the XSS/CRLF plug-in. It provides a passive analysis of user-driven data in order to point out potential XSS/CRLF vulnerabilities within specific Web-Application[104].

A reasonable utilization of WS in combination with the already mentioned Live HTTP Headers plug-in for Mozilla Firefox considers the client-side Forensics in the proposed HoneyWebEnv (Section 4.1). Live HTTP Headers detects Referer requests initiated from the local host. That's why as already proposed, it is suitable for the FO-Team tasks. Nevertheless, the utilization of a proxy between the client and the server-side implementation of the Web-Application, as the Web Scarab, should extend the Forensics Readiness (please see [JG05], p.7) of the HoneyWebEnv. Main idea is to detect and log possible calls, initiated by DOMXSS exploits, to the server, respecting the proposed strategy in Figure 3 (step 3) – i.e. as already proposed combination of DOM-based XSS with other attacks. Therefore, we should outline this subject as valuable for future work.

102 https://www.owasp.org
103 https://www.owasp.org/index.php/OWASP_WebScarab_Differences_(Classic_vs_NG)
104 https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

Web Scarab version	Plug-in	Plug-in option	
WS & WS-NG	Proxy	Proxy Control Bar	
		Intercept Request	
		Intercept Request	GET
			POST
			GET/POST
			All
		Exclude Request	
		Intercept Responses	
		Proxy Listeners	
	Spider		
	Manual Request		
	WebServices		
Web Scarab	Extensions		
	XSS/CRLF		
	SessionID Analysis		
	Scripting		
	Fragments		
	Compare		
	Search		

Table 24: Web Scarab-NG Plug-ins

4.5.2.6 Mixed tools: DOM Snitch

DOM Snitch is a tool, designed to help not only Web Administrators, Web-Devs or Pen-Testers, but also people, which can be described as normal users, newcomers or dilettantes, developing and hardening Web-Applications. In our classification of relevant for this paper tools, DOM Snitch belongs to the group of the mixed type tools. It is relatively new extension for the Chromium-based Browsers, that should be still considered as a beta-version, and yet possesses various capabilities such as (see source[105]):

- detection of the dynamically alternation of DOM,
- detection of miscellaneous security issues:
    - invalid HTML Tags,

105 http://code.google.com/p/domsnitch/wiki/QuickIntro

    - problematic HTML escaping,
    - invalid JSON[106],
    - falsified or unauthorized Cross-Origin,
    - tainted input and output data etc., accomplished via static code checking of the Web-Application
- report in form of activity logs

In order to sanitize concrete Web-App or just perform a simple scan, the user has to enter the specific URL and after the Web site is completely loaded by the Browser rendering engine, to start the DOM Snitch Chromium extension via the right-click menu. Here, we should mention, that in the current version the scanner supports exclusively http(s), as Layer-7 internet protocol. Furthermore, after the search is performed, all potential vulnerabilities are listed in a viewer log tab within the current Browser instance. Depending to their type and feasibility, they are visualized user-friendly in different colors. These (please see Figure 66) should be enumerated as follows:

- Invalid JSON
- Mixed content
- Reflected input
- Untrusted code
- Script inclusion

More over the security examiner can either execute a complete source tracing by activating all of the above mentioned settings, or perform a specific audition, considering the activation of one or more particular scanning criterion (criteria). According to their potential impact the errors are displayed as security bug (red), conceivable security bug (yellow), bug with minimal impact on security (green), multiple instances of the same alternation (gray). In addition, we should outline that an installed and activated DOM Snitch plug-in loads automatically in a separate tab after the first start of a Chromium-based Browser instance. Though, this tool experiences the same drawback as the DOM Tracer tool – the multi-tab browsing produces a mismatch in the result log. Therefore, we recommend a single instance – single tab utilization of DOM Snitch tool. This project should be designated as an active one and the author of the tool works on its further improvement.

4.5.2.7 Mixed tools: DOMinator Project (Mozilla Firefox)

The last tool, we should describe in this section, belongs to the group of mixed code analyzer, which on one hand traces the source code of the observed Web-App as a pure static code parser and on the other taints the code input strings to the Web application in real-time as a dynamic code checker.

DOMinator is developed by Stefano Di Paola[107] and the current version of the tool is v0.9.0.4, which is still in its alpha version. DOMinator is hosted on code.google.com[108], where users, researchers can find: the tool's synopsis, download area, source code SVN access[109], the tool's

106 Considering the established Security Design Patterns for JavaScript [S10]
107 http://www.mindedsecurity.com/
108 http://code.google.com/p/dominator/
109 http://code.google.com/p/dominator/source/checkout

Wiki[110] – as a support information source, and a support/discussion mailing list[111].

The installation is intuitive for experienced power users or security professionals/researchers and divided in three approaches:

- a complete virtual machine, which is set up with manuals and working DOMinator application,
- installation packages, considering Windows x86 and GNU/Linux x86,
- and compiled Firefox plug-in, which can be manually installed on working Windows/Linux Firefox Applications.

Limitations and prerequirements of the tool can be deliberated as follows:

- the DOMinator Project considers the GNU Lesser GPL;
- the tool is compiled at this stage only for the Mozilla Firefox dialect implementation – Namoroka, v3.6.13;
- there is no support for 64-bit OSes, which can be evaded by running the preinstalled virtual machine with DOMinator via VirtualBox[112] or VMware[113] Player for example;
- the Namoroka plug-in requires 3rd party plug-ins, additionally to be preinstalled: Firebug[114] v.1.7.2 and SpyderMonkey[115], which are included by default in the DOMinator plug-in package.

The operating mode of the tool should still be considered as unstable, which is denoted by abrupt crashes and application hangings of the running DOMinator under diverse Linux testing environments, based on 2.6 kernel, and current Windows based OSes: MS Windows XP Professional 32-bit and MS Windows 7 Home Premium 32-bit – all OSes installed on multicore machines. This explains the fact, that the tool is still in alpha stage, as already stated.

Furthermore, observing the preinstalled virtual machine with implemented DOMinator shouldn't be considered as efficient working environment. Though, after manually adjusted settings of the OS image – resizing adequately the swap and expanding the RAM size, better results are experienced.

Modifying the CPU support of the Virtual Machine, by applying multicore support and doubling the CPUs to a considerably efficient dual-core, brought more powerful operation of the virtual OS, though more frequent crashes of DOMinator.

Let's describe the application workflow and security related abilities of the tool more detailed.

As main operational patterns should be enumerated:

- the analyzing of the tainted data flow via SpyderMonkey, which designates the core of the DOMinator project,
- the visualizing of the feedback results and the parsed Web-App source code on behalf of FireBug.

110 http://code.google.com/p/domxsswiki/
111 https://groups.google.com/forum/#%21forum/dominator-ml
112 https://www.virtualbox.org/
113 http://www.vmware.com/de/
114 https://addons.mozilla.org/de/firefox/addon/firebug/
115 https://developer.mozilla.org/en/SpiderMonkey

The DOMinator appears in the FireBug frame as an extension tab. After it is selected, the tool's user experiences a dual panel feedback viewer. On its left side is presented the real time LogViewer, where one can select between multiple options: Log Enabled, Log Cookies Enabled[116] and Stack Trace Enabled. Furthermore, the security researcher can selectively choose the windows of the observed Web-Application and if required – to clear the current logs. On the right pane the security professional can observe the identified warnings and alerts, which are detected by the Exploit Checker, a controller of the DOMinator. The Sources and Sinks are highlighted via contrast coloring, which designates the tool as user-friendly for the security researcher, in term of pointing out the main possible security drawbacks of the currently observed Web-App, considering the main parameters for detection of DOM-based XSS injections. We illustrated a detection of DOMXSS in Subsubsection 3.2.5.2 of our exposition, please consider reading on this DOM-based XSS scenario.

We shall represent the workflow of DOMinator, derived from the original operating schema, provided by Stefano Di Paola [L12] in the following Figure 20.

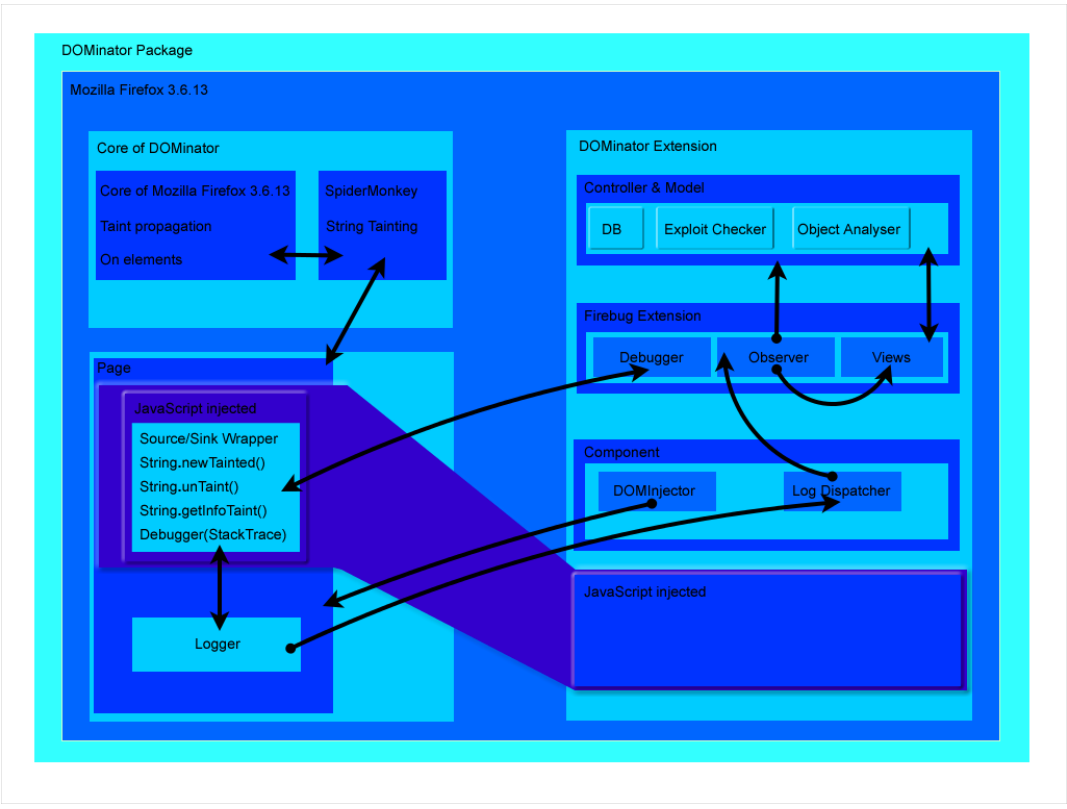


Figure 20: DOMinator Architecture, Stefano Di Paola, [L12]

We should stress out one more paradigm, which we intentionally avoided to include into the evaluation of the PT DOMXSS tools. In 2011, B. Kochher presents in [L36], [L37] another discussion on DOM-based XSS. The security researcher evaluates DOM Construction injections and the following DOMXSS PT-Tools: IBM Rational AppScan[117] and DOMinator, on behalf of

116 For enabling this feature, another 3rd party plug-in should be installed on Namoroka, Firecookie v1.3
117 http://www-01.ibm.com/software/uk/awdtools/appscan/ondemand/

seven illustrative examples. Kochher points out, that none of the tools detects properly the DOM-based XSS injections, he demonstrated in his presentation. We decided to extend his evaluation on the group of the tools, which we estimate in Subsection 4.5.2 , but WebGoat. Moreover, we examined the last paradigm presented in his slides, sample #7 [L36]. At first, we utilized a localhost scanning on the HTML content and consequentially we uploaded the HTML file online. In both cases, none of the tools detected the malicious Sink and only DOM Tracer and DOM XSS Scanner pointed out the malicious Source. We illustrate sample #7 in Table 25 and the related S3MM beneath.

```
   <!-- Derived from [L35] –>
1:  <html xmlns="http://www.w3.org/1999/xhtml">
2:    <head>
3:      <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
4:      <title>&lt;img src=0 onerror=alert(0)&gt;</title>
5:    </head>
6:    <body>
7:      <div id='demo'>Test</div><br /><br /><br />
8:      <script>
9:          function changeLink() {
10:             document.getElementById('demo').innerHTML = document.title;
11:         }
12:     </script>
13:     <input type="button" onclick="changeLink()" value="Change Test" />
14:   </body>
15: </html>
```

Table 25: B. Kochher, sample #7, [L36]

Let's represent the corresponding S3MM as follows:

S3 Meta-Model: B. Kochher, sample #7, [L36]		
Prerequirements:		The malicious code should be stored.
Core:	Source:	document.getElementById('').innerHTML
	Sink:	document.title
	Storage:	none
	Payload:	<img src=0 onerror=”alert(0)”>
Impact:		High, none of the evaluated tools successfully detects the Sink

Because of the aforementioned fact, that we achieved 100% false negatives and furthermore, that DOM Tracer and DOM XSS Scanner, which give at most false positives from the group of the

examined PT-Tools, we should not consider this last paradigm as essential for the tools' evaluation. Nevertheless, we should clearly stress out the fact, that all of the observed tools should be improved on their detection capabilities, considering DOM Templating (see [L36]).
We would like also to mention, the following additional tools:

- Ra.2[118] [L38],
- RatProxy[119] [L39],
- w3af[120], w3af/plugins/grep/domXss.py [L39],
- w3af with Selenium[121] [L39].

We briefly estimated these tools and therefore we decided not to include them in our evaluation of DOM-based XSS PT-tools. Let's deliberate our first impressions on these scanners. In [L38], page 22 a comparison between Ra.2 and DOMinator is provided. We designate the following drawbacks. We should not outline the installation of Ra.2 as straight forward. We set up Ra.2 under MS Windows 7 Home Premium and Ubuntu 11.04. For the Windows installation, we proceeded as described in the project's manual[122]. For the GNU/Linux set-up, we successfully assembled the tool on Mozilla Firefox v11.0. We should not designate the tool as lightweight and fast, and furthermore we experienced heavy loads on the Browser after scanning completions – multiple summary results tabs, were automatically opened, which drove the Browser to be unusable. Moreover, we detected only NP-XSS exploits, in distinction to those we detected with DOMinator. Scanning hangs were experienced on the following sites:

- globo.com, vimeo.com and real.de halted the scanning process at 52%,
- sample #7 [L36] halted the scanning process at 8%.

Ra.2 applies a vector list with 70 known XSS payloads, which are used for the dynamic tainting and this list can be extended manually.

The w3af PT-tool with its domXss.py extension was also briefly evaluated under MS Windows 7 Home Premium and Ubuntu 11.04. For improving the detection abilities, we activated another w3af plug-in – xssedDotCom. The main advantage of this extension is, that it searches on the xssed.com for submitted vectors, considering the URL of the particularly scanned Web-Application. As w3af is an Open Source project, it can be easily extended by another plug-in, namely our proposed S3MM DB. We experienced an irreversible crash of the tool under the Windows OS, which drove the tool unusable for further research. Despite of the experienced problems during the tools estimation, we consider them as a challenging topic for future work, concerning DOMXSS penetration testing.

Let' summarize the results of the defense chapter of our project thesis, as follows.

4.6 Synopsis

In Chapter 4, we observe in general two approaches, considering on hand the SSDLC, related to Approach I and on the other – methods for client-side protection against DOMXSS vulnerable Web-Applications, Approach II. A basic schema, illustrating a proper Web-App development is given in the next Figure 21.

118 http://code.google.com/p/ra2-dom-xss-scanner/wiki/Introduction
119 http://code.google.com/p/ratproxy/
120 http://w3af.sourceforge.net/
121 http://seleniumhq.org/projects/ide/
122 http://code.google.com/p/ra2-dom-xss-scanner/wiki/InstallationGuide

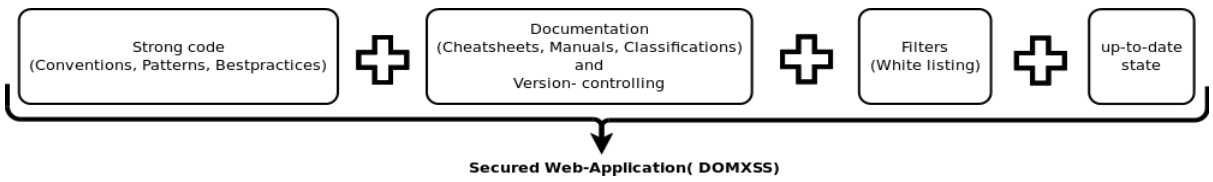


Figure 21: Secured Web-Application, considering DOMXSS

As a result, we should outline, that we encourage the constant improvement and concurrent utilization of the both approaches. We should keep in mind, that this redundancy can strengthen the client-side protection against DOMXSS, but reflect negatively to the semantics and functionality of the Web-Application. This dilemma was already digested in Chapter 4 and we would like to mention again its importance as a topic for future research.

Another important matter is the distribution of the DOMXSS sanitization filters. As aforementioned, they can be embedded straight in the HTML content, which requires a manual implementation, or they can be applied distributively to the Web-Application. Inevitably comes the question – what happens if the server(s), hosting the filtering libraries, becomes compromised and experiences DoS?

There are two huge drawbacks in this constellation, either the Web-App refuses to implement properly the Business logic, which reflects the “divide and conquer” Model given in Figure 3, or the input validation and DOM Construction filtering is hampered to utilize their proper job. Therefore, we designate this matter as an important subject for future research, too.

At this point we should consider the exposition of our project thesis as sufficient and completed. We would like to outline the most important aspects, considered for future work, in the next chapter of our project thesis.

5 Future work

We outlined in the exposition part of our project thesis aspects, related to DOMXSS, which we designate as considerably good candidates for further improvement. Let's summarize them in this chapter in a more taxonomic manner.

At first, we propose the classifications on Sources and Sinks, introduced by Stefano Di Paola, which specify the Core of our S3 Meta-Model, as significant for future work. Furthermore, the domxsswiki Project should be extended and improved as appropriate. This central place for gathering knowledge on the topic of DOM-based XSS attacks is crucial for security researches, as already explained. As a consequence to this, we should like to mention, that the proposed S3MM database and the empirical estimation on the JavaScript/AJAX 3rd party development libraries should be evaluated as interesting topics for future work.
Additionally, we propose HoneyWebEnv also as a valuable subject for further evaluation.
Moreover, the DOMXSS input validation and DOM Construction filters: PHPIDS, ESAPI4JS etc., should be improved, in particular the regular expressions. Definitely, one of the greatest challenges for future estimation represents the DOMXSS defensive filtering on semantic search.
As outlined, IceShield project is designated as promising and therefore this tool represents a great opportunity for future work. Furthermore, the Direct Proxies project should be also enhanced and extended.
Another important aspect for further improvement and approval, comprises the proposed TRRs collection, which we would like to standardize and evaluate DOMXSS security PT-tools upon.
Consequently to this, we shall encourage the collaborative work on the further development of the considerably best DOMXSS scanner, described in our paper – the Stefano Di Paola's DOMinator Project. The author and developer of this Web-Application scanner, reports that versions of DOMinator compatible with Mozilla Firefox 7+ should be released in the near future. This represents another great opportunity to improve our project as well.
Hence, we should stress out the fact, that the other related DOM-based XSS scanners, are also important to fall in the scope of future work, especially the DOM XSS Scanner. This tool is reasonable, because of its Operating Systems independent feature, though, as we demonstrated in the defense exposition and the Appendix part of our project thesis, DOM XSS Scanner comprises huge drawbacks. We hope that, the reported vulnerabilities of this interesting tool should be sanitized as soon as possible.
Moreover, we outlined an extension of the DOM-based XSS attack scenarios in the WebGoat IDE/Framework, as a promising and valuable topic for further improvement of this Educational tool.
We should also point out, that the cognitive perception of the PT-Tools is important too, as we mentioned this in the discussion on WOT.
Consequentially to this, we shall designate the dilemma – redundancy in the preventive approaches versus sustaining of the Web-App functionality and semantics, as a valuable vector to concentrate future improvements upon.
Also, we should mention the fact that, along with the additional PT-tools outlined at the end of Subsubsection 4.5.2.7 , there are other XSS-detection scanners like – MetaXSSSploit, XSSF, XSSer etc., which are not implicitly designated as DOMXSS-able. It is advisable, that these tools, should be evaluated, whether they could indeed provide contribution to the detection of DOM-based XSS attacks, and therefore should be associated to the group of the estimated PT-Tools, as related future work.
Last but not least, all proposed Models, Schemes and TRRs should be approved on strong

mathematical level and supported with sufficient argumentation, considering the aspects of topic relation, compactness and completeness.

6 Conclusion

We were motivated to develop our project, because of the fact, that there is still no well-structured documentation, considering the 3rd subclass of the Cross-site Scripting attack vector – DOMXSS. At first, we made an extensive research and gathered comprehensive amount of documentation related to the attack. As next, we decided to proceed with the practical part of our project thesis, namely to utilize various types of XSS attack techniques and to evaluate the corresponding penetration testing tools. We applied manual injections and scanning approaches, and consequentially automated point'n'scanning. Hence, we observed possible sanitization methods and proceeded with the estimation of more complex attack scenarios. We noticed, that the distinction of the DOM-based XSS attack paradigms to the classic Cross-site Scripting ones, was not straight forward. That's why, we decided not only to deliver a strong theoretical background to our thesis, but also to strive to provide simplified classifications and models. In addition to these, we selected the most illustrative and comprehensible attack examples, which build a sufficient bases to represent the mightiness of the DOMXSS.
More precisely, we start the exposition part of our project thesis with a comprehensive explanation of the classic Cross-site Scripting representatives – reflected and persistent XSS. Furthermore, we deliver a historical overview, pointing out the most significant stages in the research on the DOM-based XSS attacks and their development. We denote explicitly the current Standard for JavaScript, ECMAScript version 5. In this retrospective part, we outline valuable classifications, methods for sanitization and adopt interesting techniques from the PT-scanning.
As next fundamental part in the paper's exposition, we specify the comprehensive description of the DOMXSS technical background. Firstly, we introduce the Intruder Model and give definition of the attack environment. Moreover, we provide an illustrative model on the general attack approach, based on the “divide and conquer” method. Hence, we distinguish the main differences between the DOM-based XSS and classic XSS – DOMXSS concerns exploit on the migrated Web-Application Business logic on the client-side. We focus on the trends in the Web-App development and proceed with an exhaustive approach on the attack technical description. Therefore, we utilize the Strategic-Tactical-Operational Model, STO. On the strategic layer, we improve the Sources/Sinks classification, given by Stefano Di Paola, to a simplified Meta-Model. We should call this the Source/Sink/Storage Meta-Model, S3MM. Furthermore, we provide a proposal for building a database upon S3MM, which should considerably ease way for statistical evaluation of the existing DOM-based XSS attacks.
On the tactical layer, we discuss different techniques, representing obfuscation as improvement of the attack completion. We consider them either as prerequirements or variations of the attack payload, taking into count the S3 Meta-Model.
On the operational layer, we illustrate two case studies as attack scenarios and two valuable DOMXSS examples. Further paradigms, we demonstrate in the Appendix part of the paper.
The next and most capacious chapter (logical part) delivers knowledge on the defensive techniques against DOM-based XSS. We specify two general approaches as fundamental for the application of proper defense. Approach I concerns exclusively the proper development of Web-Applications, respecting a proposed Defensive STO Model. Thence, we suggest a Security Software Development Lifecycle, SSDLC. We illustrate further the SSDLC on behalf of a simplified Deterministic State Machine, respecting the prevention against DOMXSS. As next crucial aspect, we outline the involved teams, as Roles, participating in the SSDLC. In general, we denote a Development Team, a Quality Assurance Team and a Security Team. Further clusterization of the

Sec-Team, is specified as follows: an Incident Response Team, a Forensics Team and a Penetration-testing Team.
As next, we propose a supportive model, a HoneyWebEnv, suitable for the concurrent utilization of the Sec-Team's tasks during the utilization of the SDLC. We suggest an advanced technique for improving the SSDLC deployment, on behalf of the Application Flow Analysis. Two approaches – proposed by Rafal Los and S. Jensen et al., support this advanced technique and are described in detail within the related subsection of the defense part, Chapter 4. Next valuable matter in the DSTO represent the JavaScript/AJAX patterns. We observe and illustrate exhaustively the JavaScript Coding patterns, Design patterns, paradigms of Anti-patterns, DOM and Browser related patterns, and Security patterns.
Concerning the tactical layer of DSTO, we discuss the input validation and DOM Construction filters against DOMXSS. We compare different approaches for application of Web-App filtering: common implementations, based on ECMAScript predefined functions; advanced implementations, based on regular expressions, as 3rd party filters, PHPIDS supported Regexes; mixed implementations and supportive implementations – WOT. We propose a general schema, concerning the assembly of the DOMXSS sanitization filters on the client-side logic of the Web-Application. As an illustrative example of mixed filters, we suggest a derived and improved JavaScript based filter for e-mail input validation. At this point, we stress out that a challenging topic for future work represent the input validation filters on Semantic Search. Another challenging aspect designates the log abilities of such filters. Concerning the DOM Construction sanitization filters, we discuss the predefined methods in the ECMAScript 5 Standard. Furthermore, we illustrate an impressive manual implementation of integrated DOM Construction filtering, represented by XSSMe² contest project. At this point, we stress out the fact, that implementation of proper input validation and DOM Construction defensive filters should be considered as sophisticated. Thence, we deliver a description of 3rd party libraries, which provide Web-Application defensive filtering, as ESAPI4JS. We conclude Approach I, as we enumerate the related Cheat Sheets, concerning hardening of Web-Apps against DOMXSS.
As next, we explain the second general approach, Approach II. This approach axiomatically assumes, that Web-Applications are vulnerable to DOM-based XSS and considers the questions about client-side security. Valuable representatives in this approach are described as follows: Browser integrated XSS filters, 3rd party XSS filters, a complete Software implementation as a secure Browser and supportive Browser plug-ins. As the most important example in the group of the XSS filters, we designate IceShield, which is based namely on the Secure ECMAScript project – Direct Proxies. Furthermore, we compare the 3rd party filters application to JavaScript/AJAX 3rd party coding libraries and propose an empirical estimation, comparing the abilities of the different coding libraries, as a valuable opportunity for future work. As next, we evaluate the secure Browser “Opus Paladium” (OP2) against DOMXSS and XSS attacks. In our case study, we determine OP2 as a non-matured project, which comprises many drawbacks. We designate OP2 as a good candidate for future improvements, too. At last, we finish Approach II with the evaluation of the Browser plug-in Web of Trust (WOT). We conclude the defensive part of the paper with DOMXSS pen-testing tools, where we compare and estimate seven different PT-Tools. We propose a further classification, considering tools standardization and a set of fundamental aspects supporting the tools improvement, which we should call – Tools' Requirement Rules, TRRs. We shall pay attention especially to the definition of the TRRs as a collection. This designates them as another important topic for future work. At the end of the tools' discussion, we outline new DOMXSS injections, which determine the pen-testing tools for future refinement, according to their detection abilities. Thus, we consider the defensive part of our project thesis as completed.

At last, we systematically outline the various valuable aspects for future work on DOM-based XSS attacks, in a separate chapter.
Hence, we consider the deployment of our project as sufficient to the proposed objectives and able for further refinement.

Affirmation

Hereby we declare that we have written this thesis by ourselves without any assistance from third parties and have exclusively used the indicated literature and resources.

Bochum, 16. Apr. 2012 Z. Danailov, K. Deltchev

Eidesstattliche Erklärung

Hiermit versichern wir, dass wir das Masterstudienprojekt selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Bochum, 16. Apr. 2012 Z. Danailov, K. Deltchev

Bibliography

[AE11]	Arian Evans, 2011: Web Application Security Metrics Landscape	2011
[WHS10]	WhiteHat Security, WhiteHat Website Security Statistic Report, 9th Edition	2010
[WHW11]	WhiteHat Security, WhiteHat Website Security Statistic Report, 11th Edition	2011
[KD10]	Krassen Deltchev, New Web 2.0 Attacks	2010
[S10]	S. Stefanov, JavaScript Patterns	2010
[JGETA07]	J. Grossman et al., XSS Attacks - Cross Site Scripting Exploits & Defense	2007
[WASC10]	Web Application Security Consortium, WASC THREAT CLASSIFICATION	2010
[RL10]	Rafal Los, Into the Rabbit Hole: Execution Flow-based Web Application Testing	2010
[CW07]	Brian Chess, Jacob West, Secure Programming with Static Analysis	2007
[MH12]	M. Heiderich, Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM	2012
[LS09]	Larry Suto, Analyzing the Effectiveness and Coverage of Web Application Security Scanners	2009
[JMT09]	S. Jensen, A. Møller, P. Thiemann, Tape Analysis for JavaScript	2009
[JMT11]	S. Jensen, A. Møller, P. Thiemann, Interprocedural Analysis with Lazy Propagation	2011
[KU77]	J. Kam, J. Ullman, Monotone Data Flow Analysis Frameworks	1977
[JMM11]	S. Jensen, M. Madsen, A. Møller, Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications	2011

[G94]	E. Gamma et.al., Design Patterns: Elements of Reusable Object-Oriented Software	1994
[S06]	M. Schumacher et.al., Security Patterns	2006
[MN11]	M. Niemietz, JavaScript-based ESAPI: An In-Depth Overview	2011
[JW11]	J.Weinberger et.al., A Systematic Analysis of XSS Sanithization in Web Application Frameworks	2011
[MH11]	M. Heiderich, IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM	2011
[ST10]	S. Tang et al., Alhambra: A System for Creating, Enforcing, and Testing Browser Security Policies	2010
[JG05]	J. Garcia, Proactive & Reactive Forensics Forensics, Antiforensics & Automation	2005
[BS09]	Breach Security, Inc., The Web Hacking Incidents Database 2009, Bi-Annual Report	2009

List of Links

L1	*JavaScript and the DOM Series: Lesson1*; J. Padolsey; 2009. http://net.tutsplus.com/tutorials/javascript-ajax/javascript-and-the-dom-series-lesson-1/
L2	*Into the Rabbit Hole: Execution Flow-based Web Application Testing;* R.Los; 2010. http://www.securitytube.net/video/1102, http://www.youtube.com/watch?v=JJ_DdgRlmb4&feature=related
L3	*DOM Based Cross Site Scripting or XSS of the Third Kind*; A.Klein; 2005. http://www.webappsec.org/projects/articles/071105.shtml
L4	*Cross-site File Upload Attacks*; P.D.Petkov; 2008. http://www.gnucitizen.org/blog/cross-site-file-upload-attacks/
L5	*Remote File Inclusion*; R.Auger; 2005. http://projects.webappsec.org/w/page/13246955/Remote-File-Inclusion
L6	*OWASP Top 10 for .NET developers part 2: Cross-Site Scripting (XSS)*; T. Hunt; 2010 http://www.troyhunt.com/2010/05/owasp-top-10-for-net-developers-part-2.html derived from: http://www.slideshare.net/jeremiahgrossman/whitehat-security-website-security-statistics-report-q109
L7	*domxsswiki*: http://code.google.com/p/domxsswiki/wiki/Index
L8	*IIS allows universal CrossSiteScripting*; Thor Larholm; 2002. http://cd.textfiles.com/hmatrix/Tutorials/hTut_0165.html
L9	*25c3: Attacking Rich Internet Applications*; S. Di Paola, kuza55; 2008. http://www.youtube.com/watch?v=RNt_e0WR1sc
L10	*Client-side JavaScript Security vulnerabilities:* *The Twilight Zone of Web Application Security*; O. Segal; 2011. http://www.slideshare.net/orysegal/clientside-javascript-vulnerabilities
L11	*Next Generation Web Attacks –HTML 5, DOM(L3) and XHR(L2)*; S. Shah; 2011. http://www.slideshare.net/shreeraj/shreeraj-shah-html5owasp
L12	*DOMinator Control Flow*; Minded Security; 2011. http://dominator.googlecode.com/files/DOMinator_Control_Flow.pdf
L13	*Analysis and Identification of DOM Based XSS Issues*; S. Di Paola; 2012. http://www.nds.rub.de/media/attachments/files/2012/01/AnalyzingDOMXSS_StefanoDiPaola_Bochum.pdf
L14	*Hacking Web 2.0 - Defending Ajax and Web Services*; S. Shah; 2007. http://www.slideshare.net/shreeraj/hacking-web-20-defending-ajax-and-web-services-hitb-2007-dubai
L15	*Next Generation Web Attacks –HTML 5, DOM(L3) and XHR(L2)*; S. Shah; 2011. http://www.slideshare.net/shreeraj/shreeraj-shah-html5owasp
L16	Cardisoft eUniversity Solutions; http://www.cardisoft.eu/frontend/article.php?aid=87&cid=96

L17	TIBCO Developer Network; http://developer.tibco.com/
L18	*The story of Web 2.0 and SOA continues - Part 1*; D. Hinchcliffe; 2007. http://i.zdnet.com/blogs/soaweb20convergence_update2.png?tag=content;siu-container
L19	*Is Web 2.0 The Global SOA?*; D. Hinchcliffe; 2005. http://dionhinchcliffe.com/2005/10/28/is-web-2-0-the-global-soa/
L20	http://dbis-group.uni-muenster.de/projects/WOA/
L21	*A Web-Oriented Architecture (WOA) Un-Manifesto*; D. Hinchcliffe; 2009. http://hinchcliffe.org/default.aspx
L22	*Web 2.0 success stories driving WOA and informing SOA*; D. Hinchcliffe; 2008. http://www.zdnet.com/blog/hinchcliffe/web-20-success-stories-driving-woa-and-informing-soa/168
L23	*Hacking Browser's DOM Exploiting Ajax and RIA*; S.Shah; 2010. https://media.blackhat.com/bh-us-10/presentations/Shah/BlackHat-USA-2010-Shah-DOM-Hacks-Shreeraj-slides.pdf
L24	*Double Encoding*;OWASP; 2009. https://www.owasp.org/index.php/Double_Encoding
L25	*Locking the Throne Room* *How ES5+ will change XSS and Client Side Security*; M. Heiderich; 2011. http://www.slideshare.net/x00mario/locking-the-throneroom-20
L26	*SQL Injection, XSS, Cross-site Scripting,* *Defenses, Attacks*; M.Paul; 2007. http://securitymasala.files.wordpress.com/2007/11/mano_paul_sqlinjandxss_catalyst_eu.pdf
L27	*Strategie statt Zufall*; U.Ries; http://cyberpress.de/wiki/Security_Development_Lifecycle,_Teil_1
L28	*Patterns – AJAX Patterns*; http://ajaxpatterns.org/Patterns
L29	*Non-alphanumeric code with JavaScript & PHP* *Shazzer - Shared online fuzzing*; G. Heyes; 2011. http://www.rubcast.rub.de/index2.php?id=918
L30	*JavaScript Smallest NonAlnum Quine*; E. V. Nava (sirdarkcat); 2010. http://sla.ckers.org/forum/read.php?24,33201,33713
L31	*What is Semantic Search?*; T. John; 2012. http://www.techulator.com/resources/5933-What-Semantic-Search.aspx
L32	*XSSMe²*; M. Heiderich; 2011. http://xssme.html5sec.org/
L33	*Changes to JavaScript, Part 1: EcmaScript 5*; M. S. Miller et al.; 2009. http://www.youtube.com/watch?v=Kq4FpMe6cRs
L34	*Changes to ECMAScript, Part 2:* *Harmony Highlights - Proxies and Traits*; T. Van Cutsem; 2010. http://www.youtube.com/watch?v=A1R8KGKkDjU&feature=relmfu
L35	*“Opus Paladium” Web browser project Website*; 2010.

	http://code.google.com/p/op-web-browser/
L36	*DOMXSS: Encounters of the 3rd kind*; B. Kochher; 2011. http://www.slideshare.net/clubhack/domxss-club-hack2011clubhack2011
L37	*DOMXSS: Encounters of the 3rd kind*; ClubHack; 2011. http://www.clubhack.tv/event/2011/ http://www.youtube.com/watch?v=-CPZwORSBLE http://www.youtube.com/watch?v=nJY9-nJyX0s
L38	*Ra.2 DOM XSS Scanner* *A DOM-based XSS scanner, for the rest of us!*; N. Patnaik, S.Sahoo; 2012. http://www.slideshare.net/nishantdp/nullcon-2012-ra2-blackbox-dombased-xss-scanner
L39	*Автоматизированный поиск уязвимостей вида* *DOM/based XSS: проблемы и решения*; Тарас Иващенко; 2010. http://www.slideshare.net/risspa/risspa-domxss

Appendix

Chapter 1, related figures

Cross-Site Scripting statistics

Top Attack methods

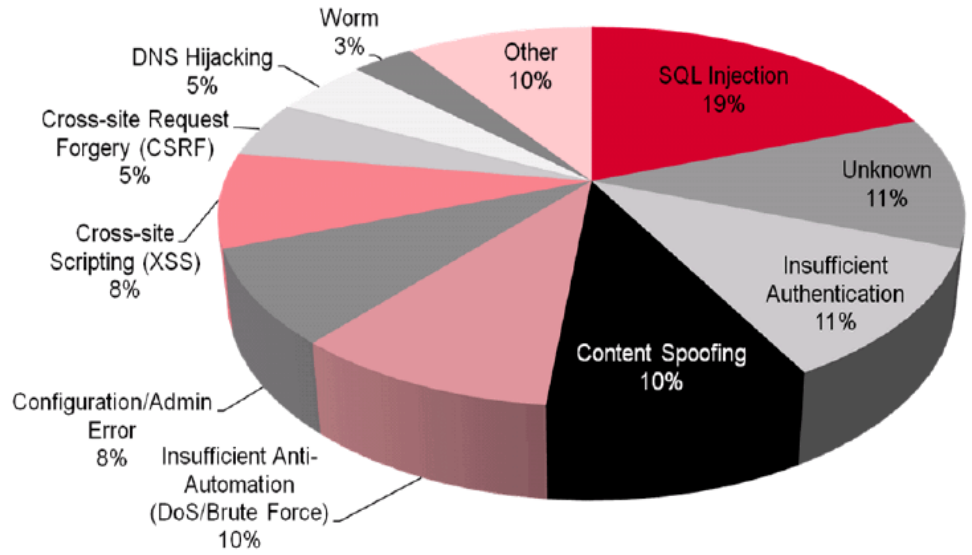


Figure 22: The Web Hacking Incidents DB '09 [BS09]

Attack pathways

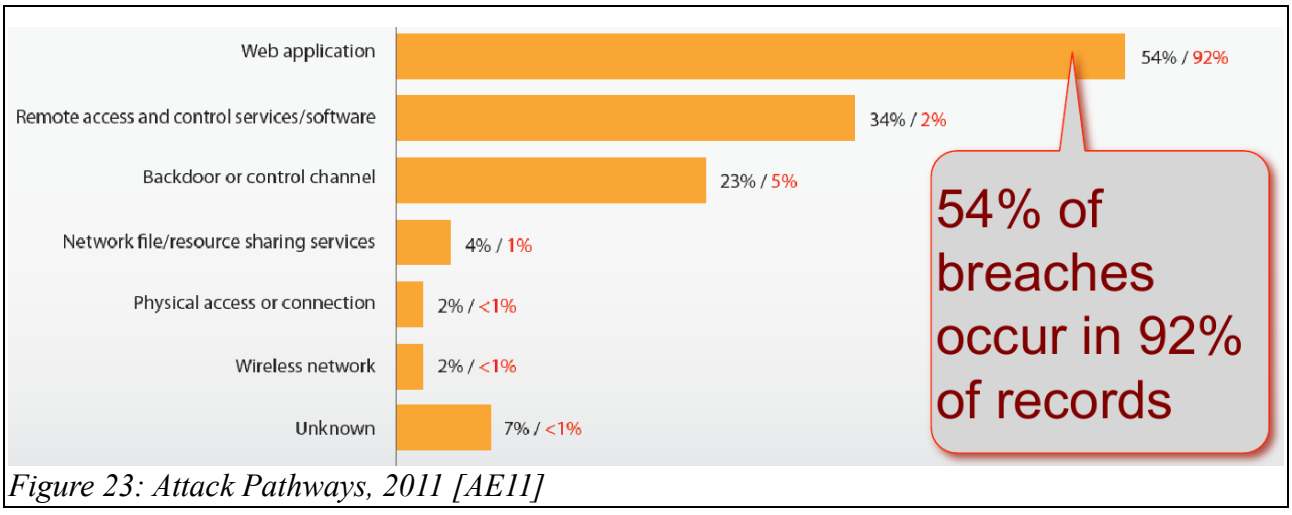


Figure 23: Attack Pathways, 2011 [AE11]

WhiteHat Security Top 10

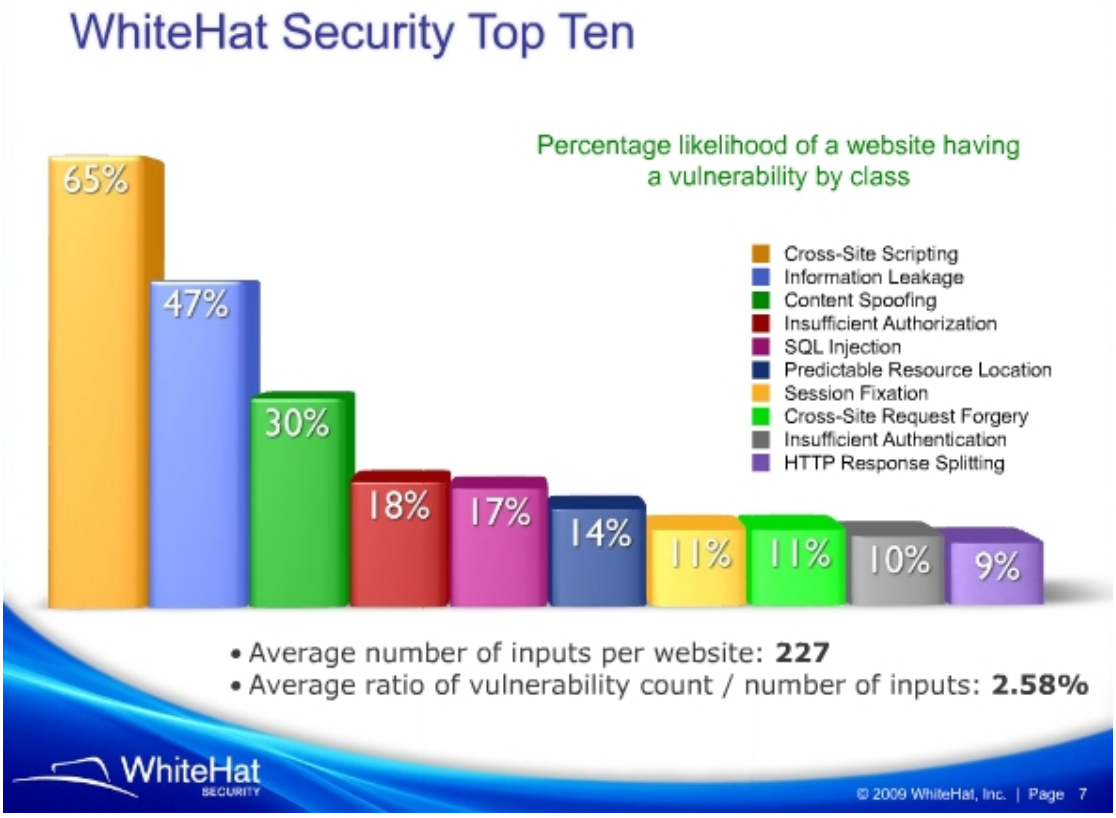


Figure 24: Whitehat Security Top 10 [L6]

Percentages of vulnerabilities resolved (compared by extensions)

Class of Attack	Severity	ASP	ASPX	CFM	DO	JSP	PHP	PL
SQL Injection	Urgent	70%	72%	66%	79%	58%	70%	71%
Insufficient Authorization	Urgent	21%	45%	46%	20%	25%	18%	10%
Directory Traversal	Urgent	43%	20%	67%	0%	33%	32%	16%
Cross Site Scripting	Urgent	100%	0%	100%	0%	0%	50%	0%
Cross-Site Scripting	Critical	51%	57%	50%	51%	52%	66%	54%
Cross-Site Request Forgery	Critical	18%	34%	17%	27%	39%	57%	27%
Session Fixation	Critical	19%	18%	0%	36%	50%	50%	100%
Abuse of Functionality	Critical	76%	23%	82%	38%	57%	59%	97%
Insufficient Authentication	Critical	55%	37%	0%	33%	71%	0%	100%
Information Leakage	High	32%	34%	57%	49%	45%	39%	29%
Content Spoofing	High	31%	30%	43%	37%	44%	46%	69%
Predictable Resource Loc.	High	29%	64%	85%	64%	53%	56%	29%
HTTP Response Splitting	High	28%	24%	33%	10%	36%	42%	35%
Directory Indexing	High	33%	56%	40%	25%	27%	33%	18%
TOTAL		65%	67%	75%	72%	63%	69%	74%

Figure 25: Percentages of vulnerabilities resolved (compared by extensions) [WHS10]

Top Ten Vulnerability Classes (compared by extension)

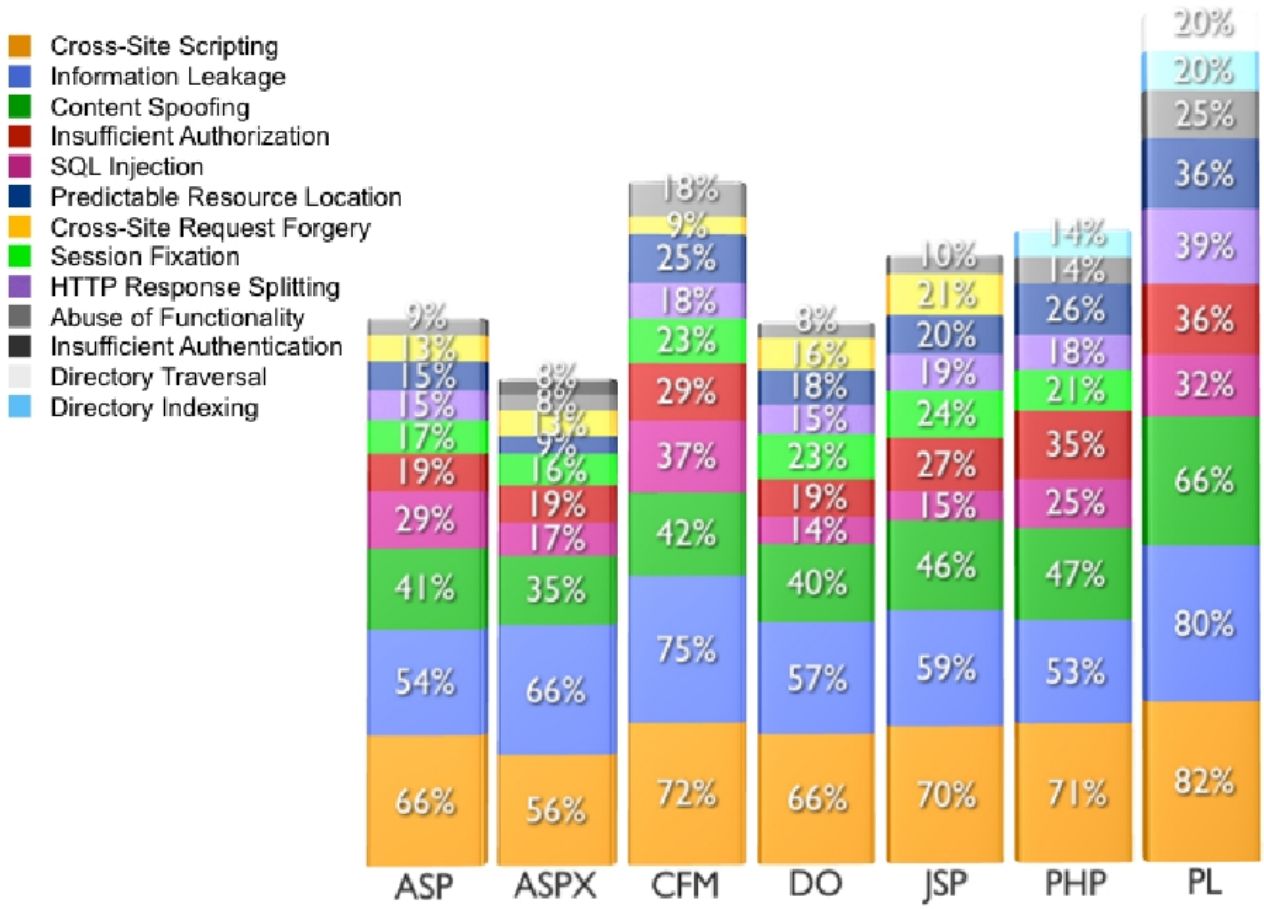


Figure 26: Top Ten Vulnerability Classes (compared by extension) [WHS10]

2010 at a glance – Sorted by industry sectors

Industry	Nr. of Vulns	Std. Dev	Remediation rate	Std. Dev	Window of Exposure(Days)
Overall	230	1652	53.00%	40.00%	233
Banking	30	54	71.00%	41.00%	74
Education	80	144	40.00%	36.00%	164
Financial Services	266	1935	41.00%	40.00%	184
Health care	33	87	48.00%	40.00%	133
Insurance	80	204	46.00%	37.00%	236
IT	111	313	50.00%	40.00%	221
Manufacturing	35	111	47.00%	40.00%	123
Retail	404	2275	66.00%	36.00%	328
Social Networking	71	116	47.00%	34.00%	159
Telecommunications	215	437	63.00%	40.00%	260

Table 26: 2010 at a glance – Sorted by industry sectors [WHW11]

Chapter 2, related figures

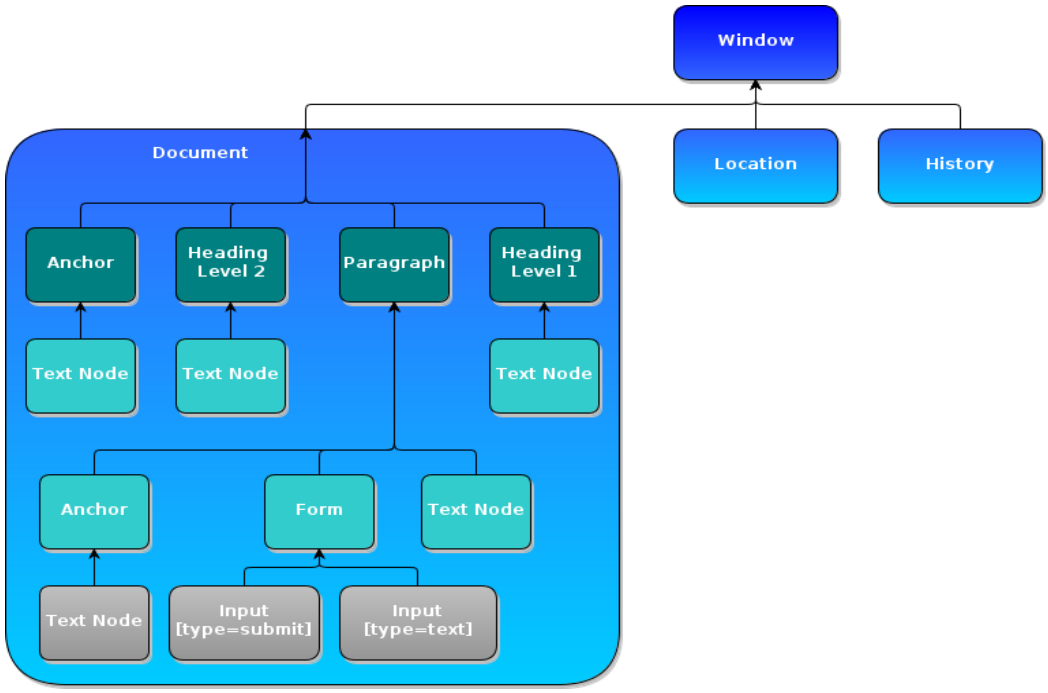


Figure 27: Basic architecture of DOM objects in HTML, derived from [L1]

ERM Models for XSS attack scenarios:

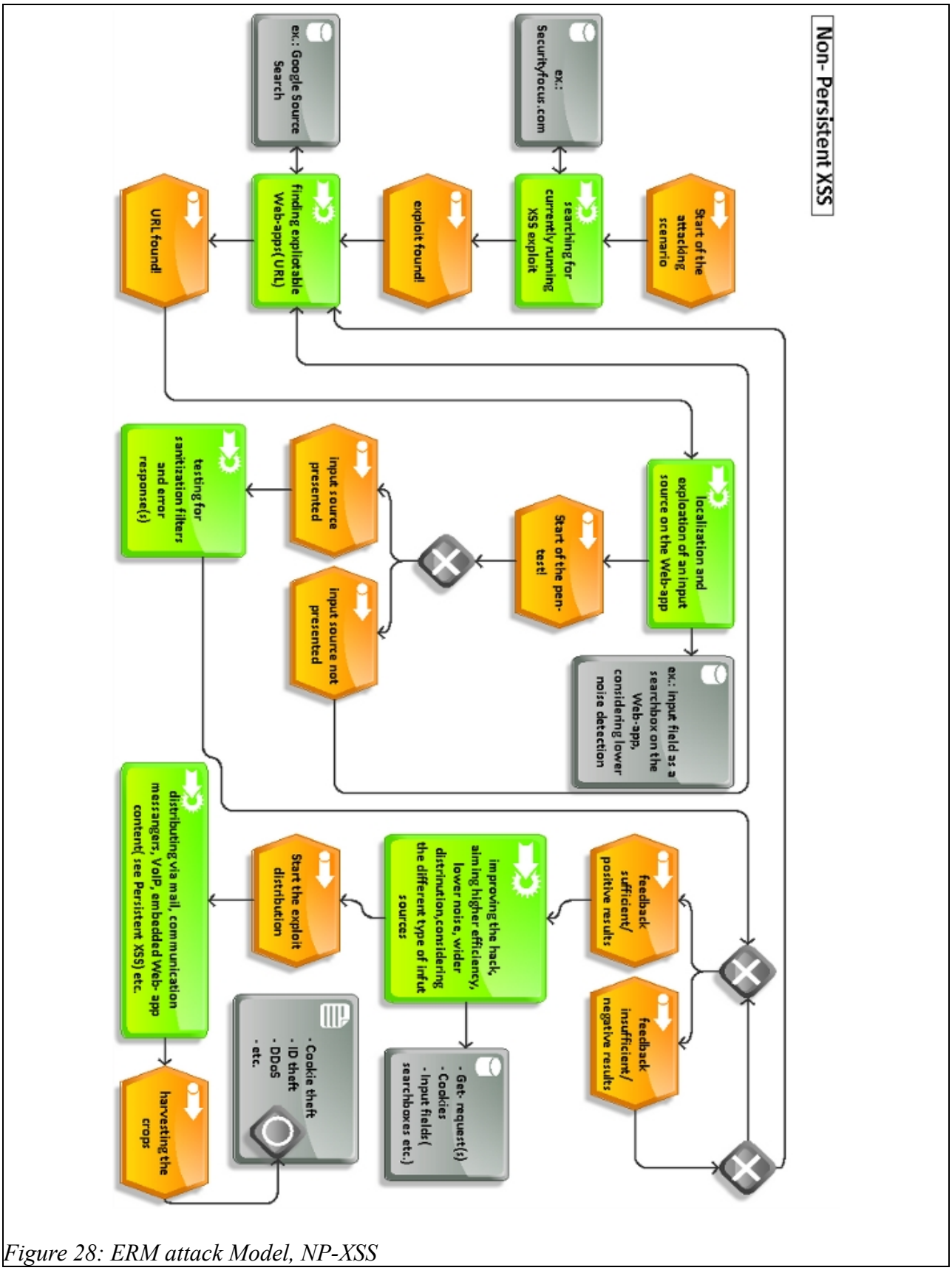


Figure 28: ERM attack Model, NP-XSS

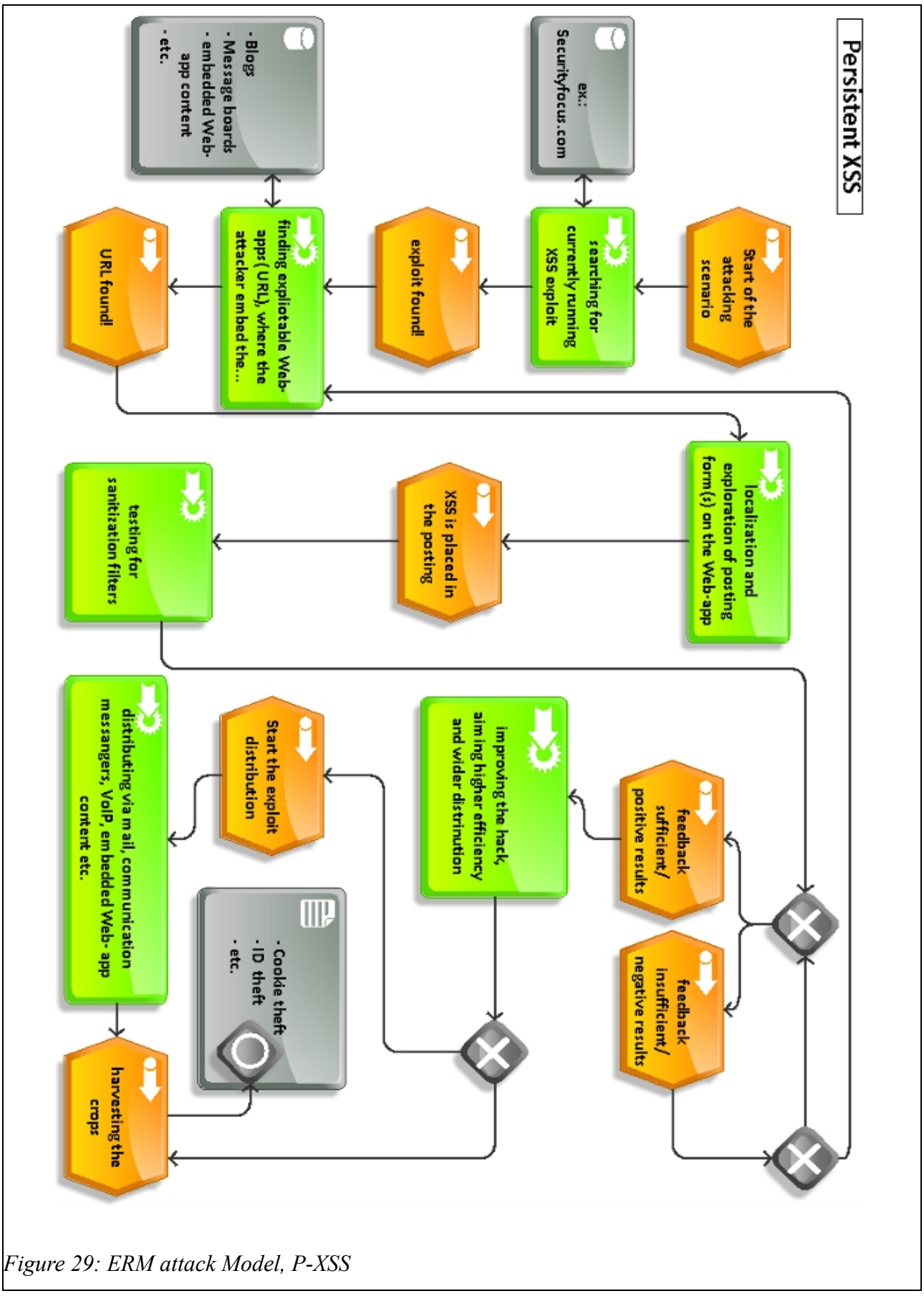


Figure 29: ERM attack Model, P-XSS

Chapter 3, related figures

Examples of HTML DOM Visualizer

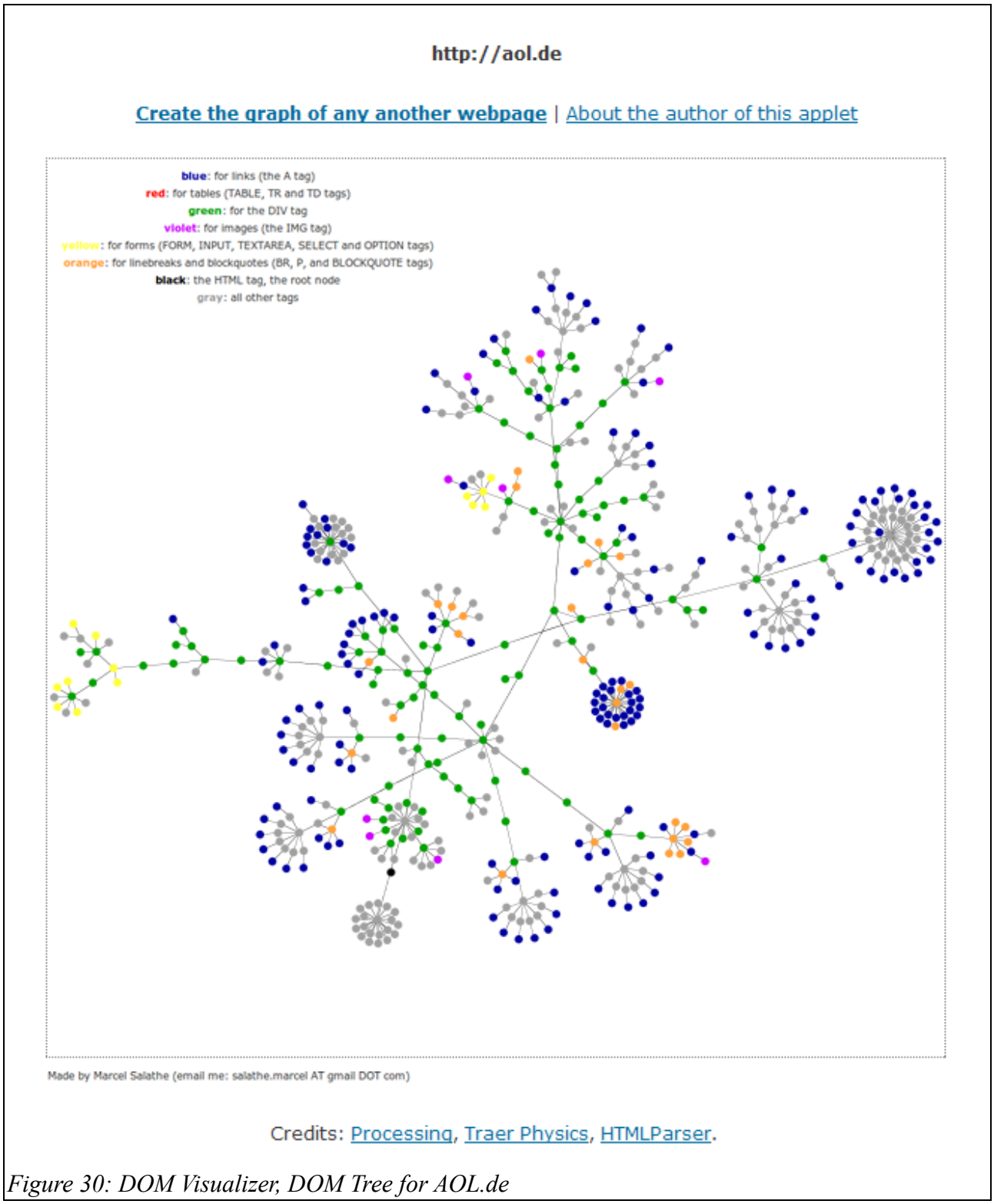


Figure 30: DOM Visualizer, DOM Tree for AOL.de

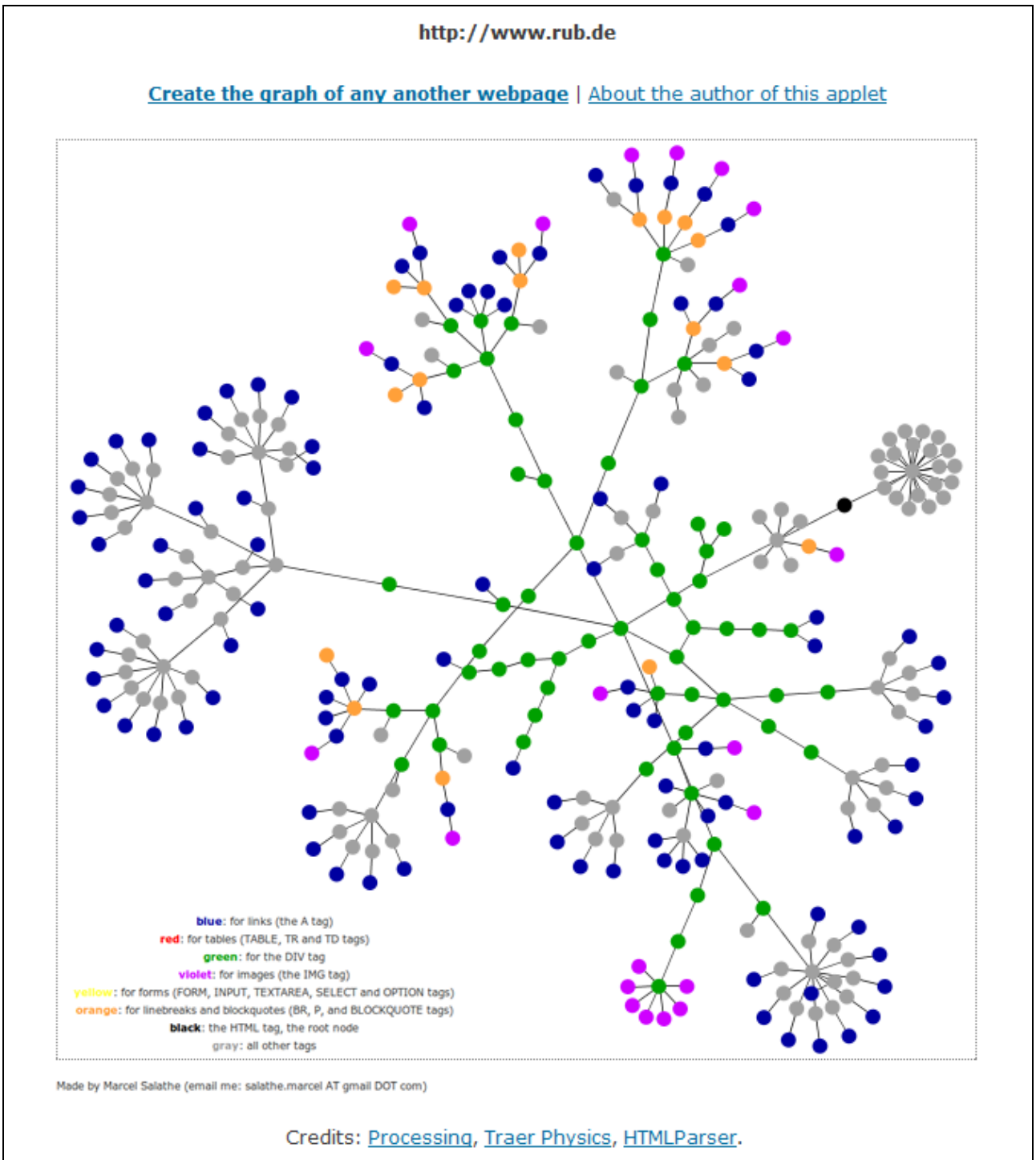


Figure 31: DOM Visualizer, DOM Tree for RUB.de

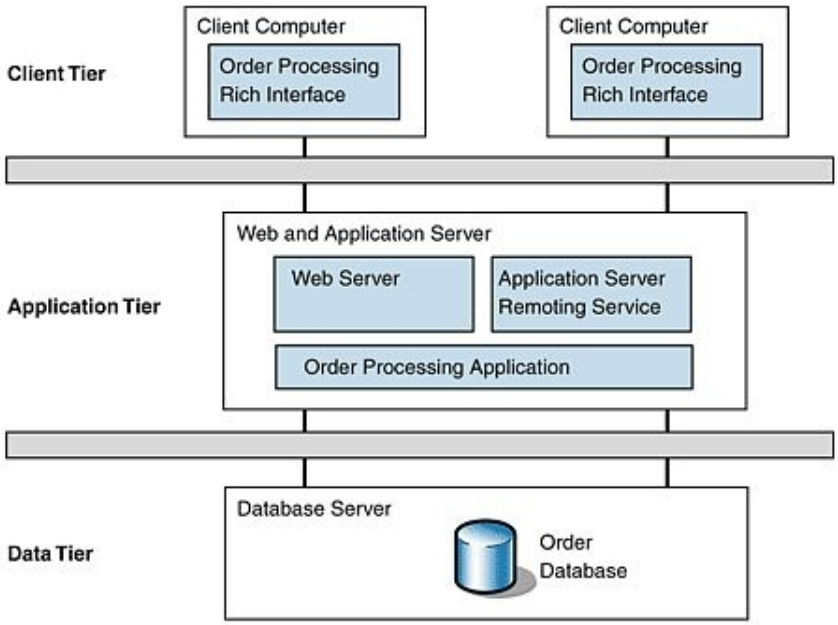


Figure 32: 3-Tier architecture [L16]

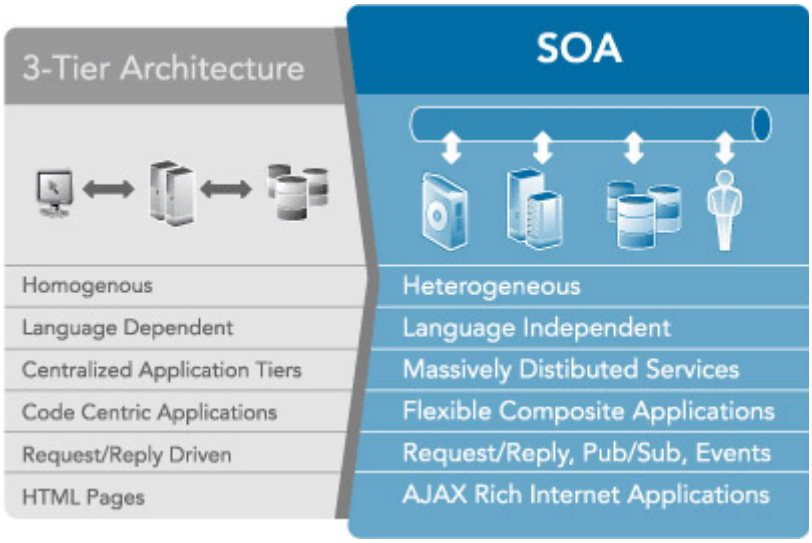


Figure 33: Application Architectures are Evolving from 3-Tier to SOA [L17]

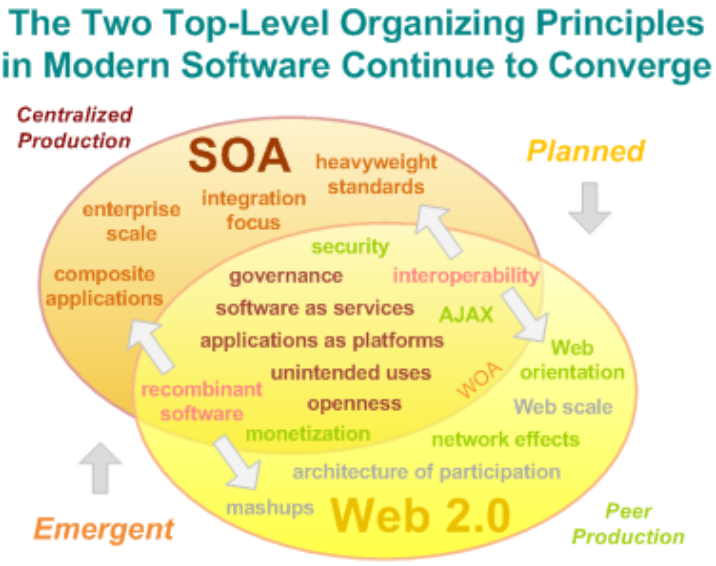


Figure 34: The 2 Top-level organizing principles in Modern Software continue to converge [L18]

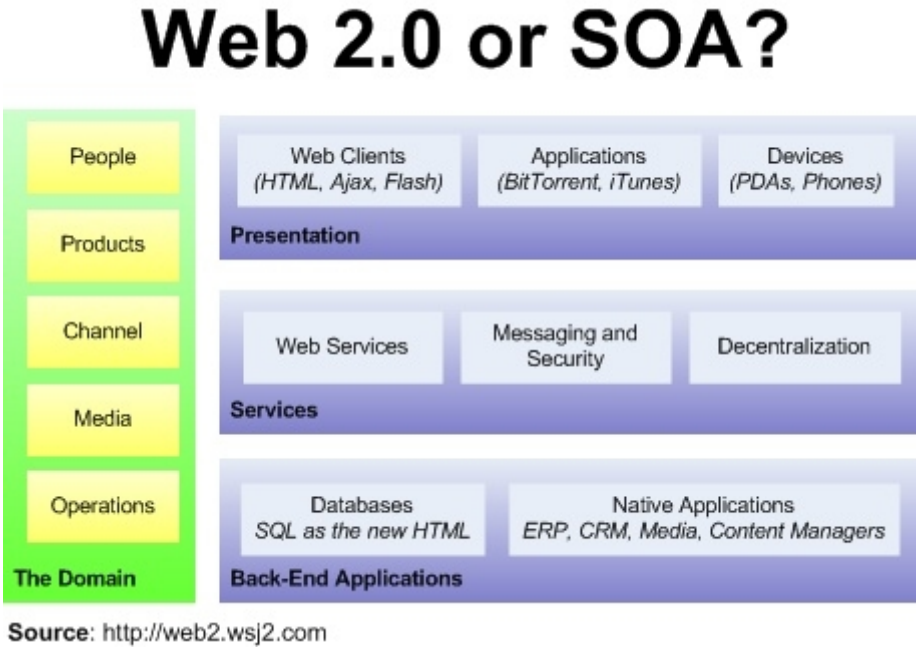


Figure 35: Web 2.0 or SOA? [L19]

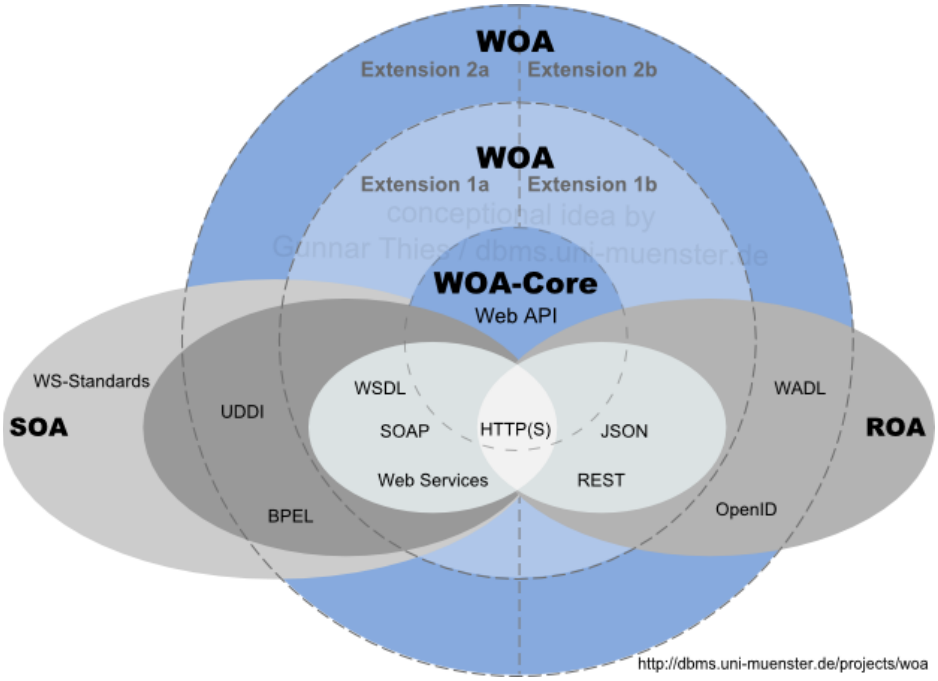


Figure 36: Technologies / standards used in a web oriented architecture [L20]

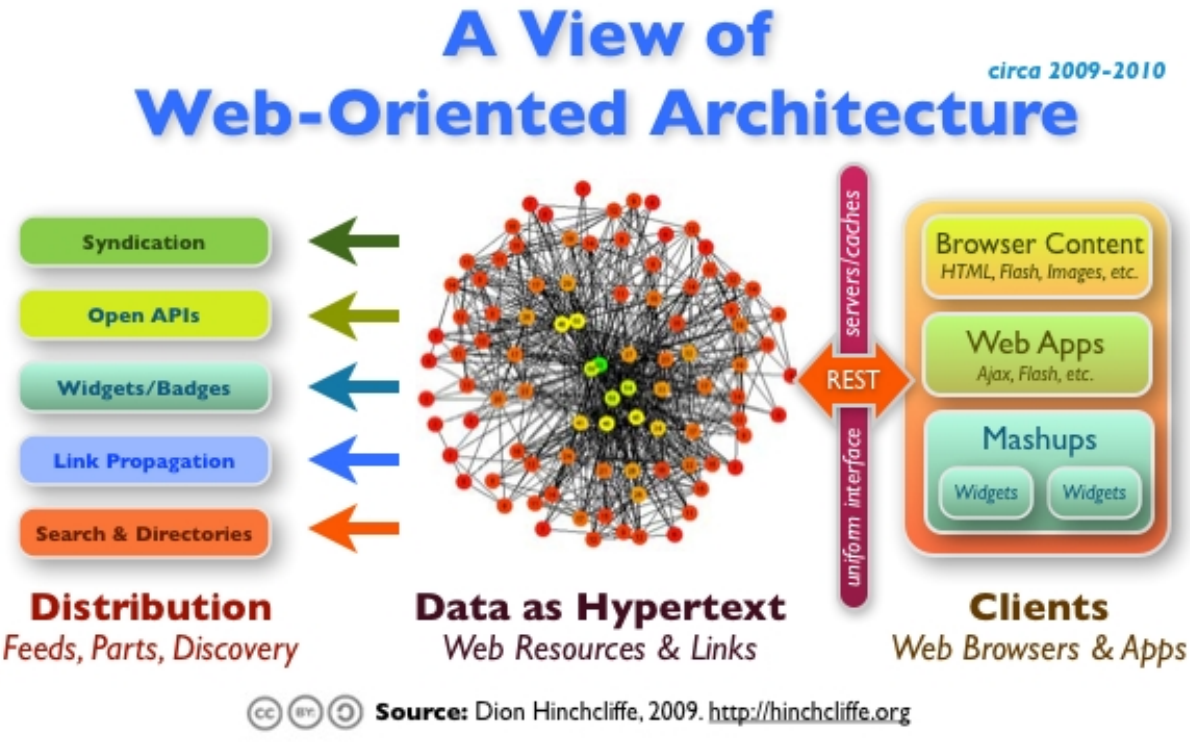


Figure 37: A view of WOA [L21]

Figure 38: The high levels of success of Web 2.0 Models for creating Software Ecosystems helped "discover" WOA and inform SOA [L22]

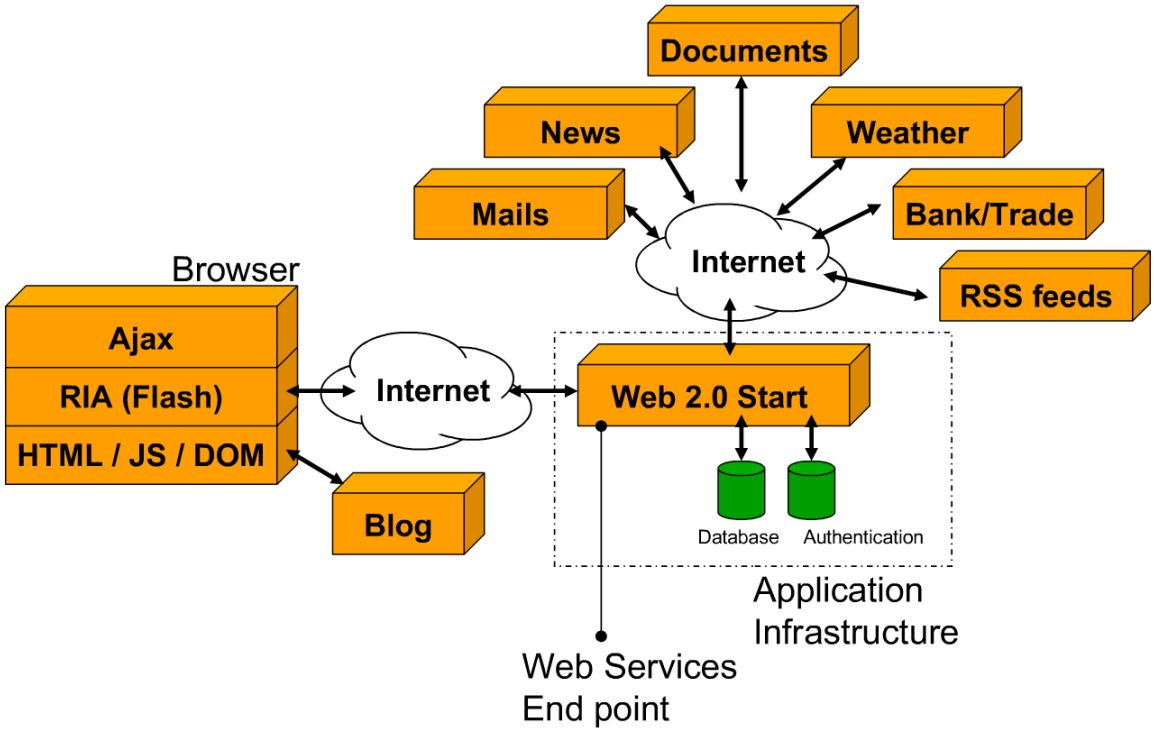


Figure 39: More technical representation of Web 2.0 Architecture [L14]

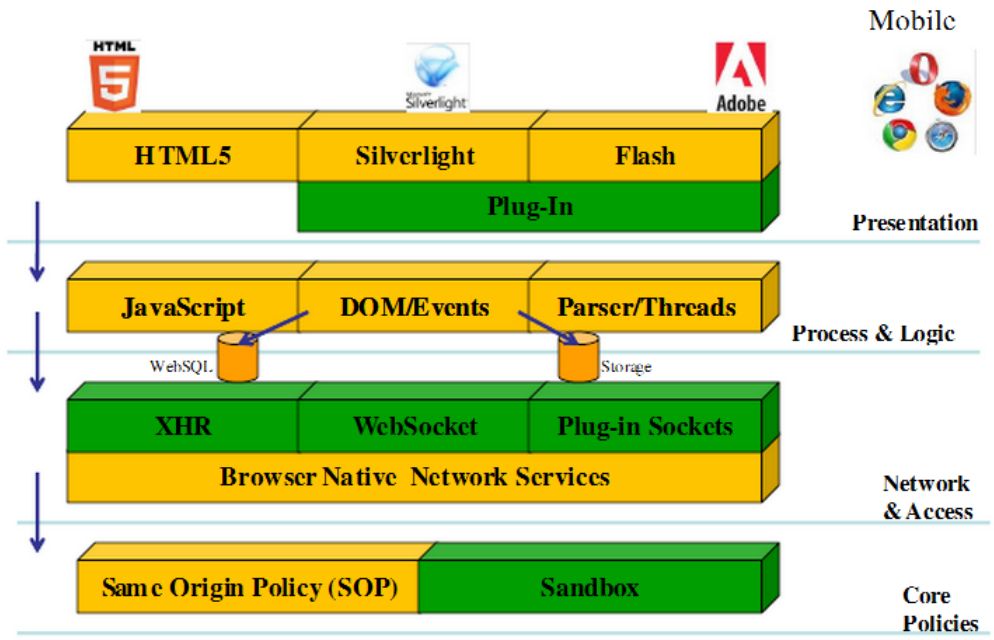


Figure 40: Browser [L15]

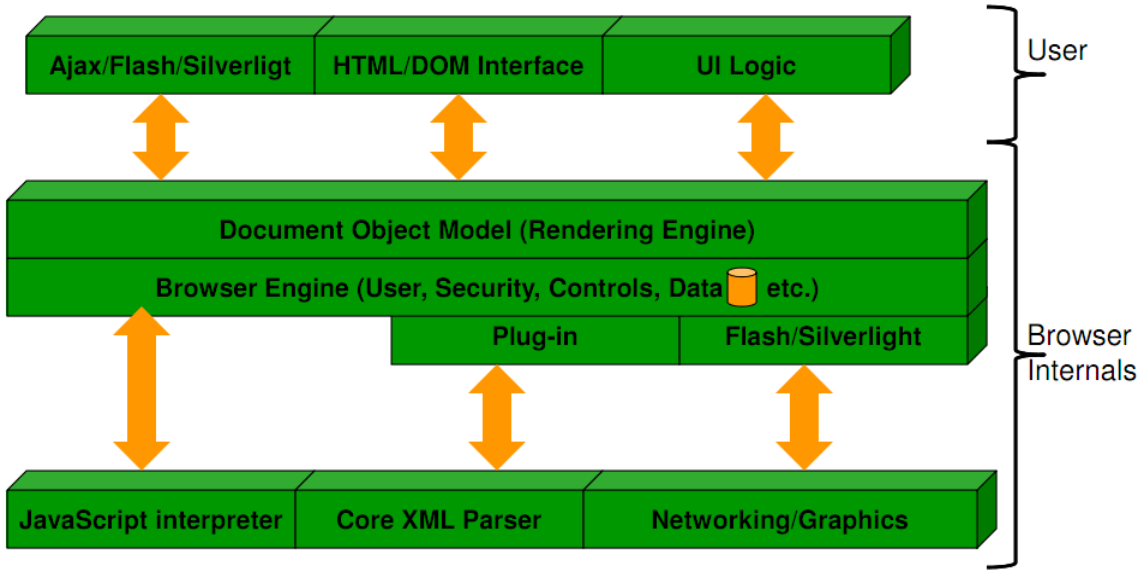


Figure 41: Browser/Application View [L23]

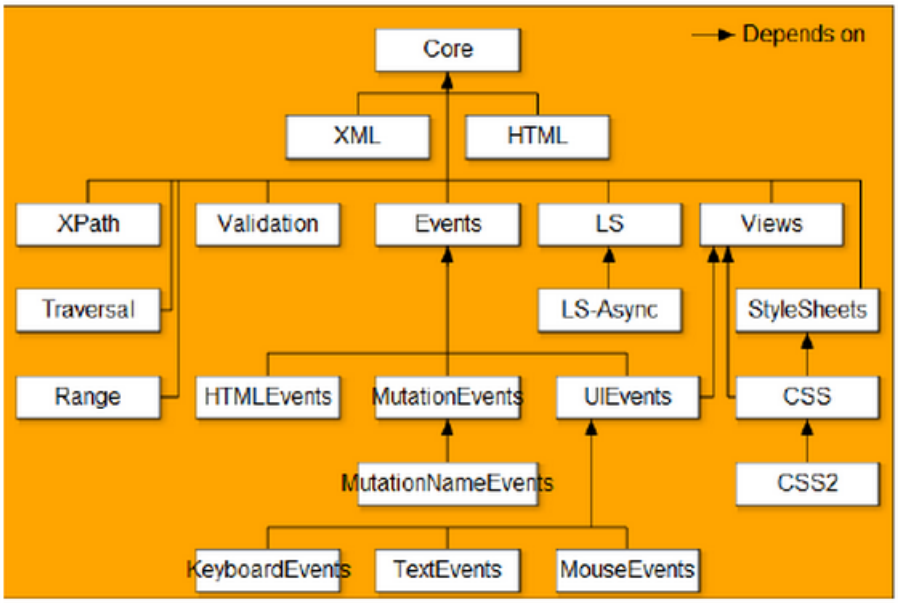


Figure 42: Browser DOM [L15]

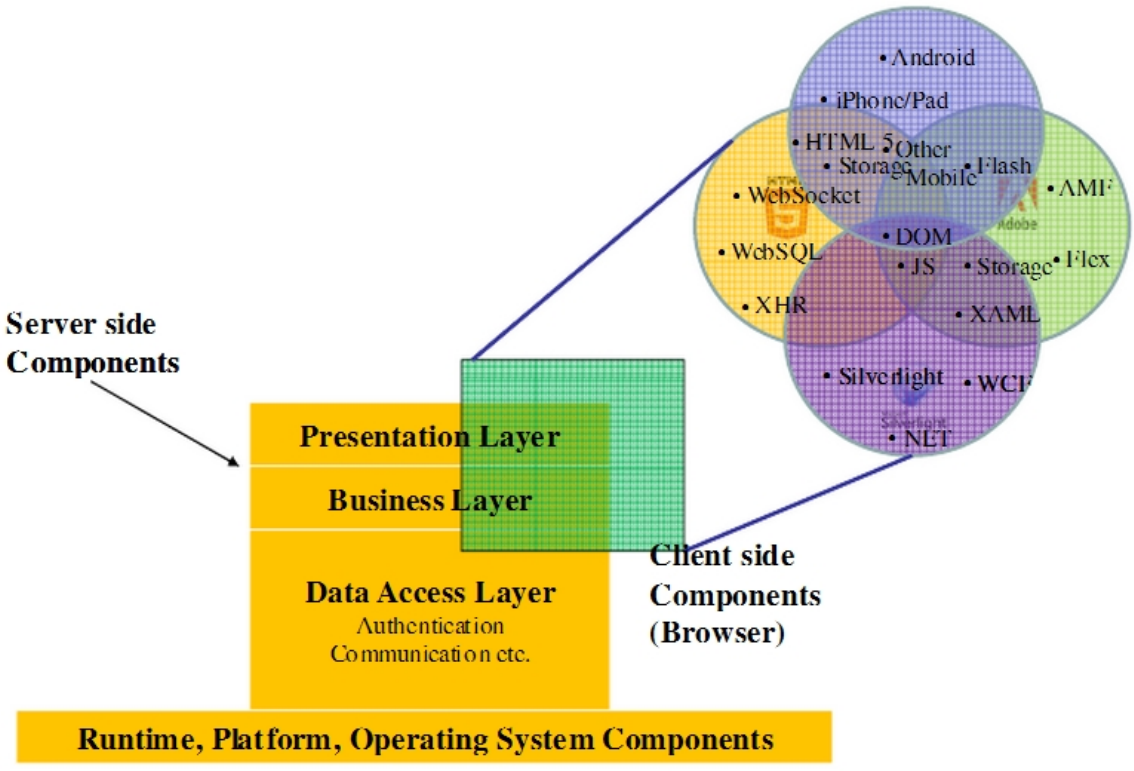


Figure 43: Technology Shift and Trend [L15]

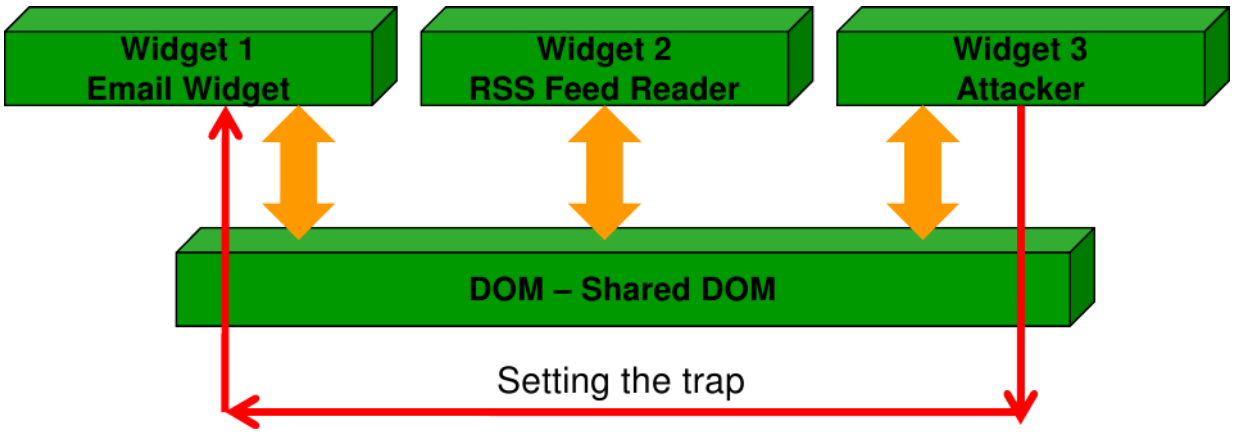


Figure 44: Cross DOM access [L23]

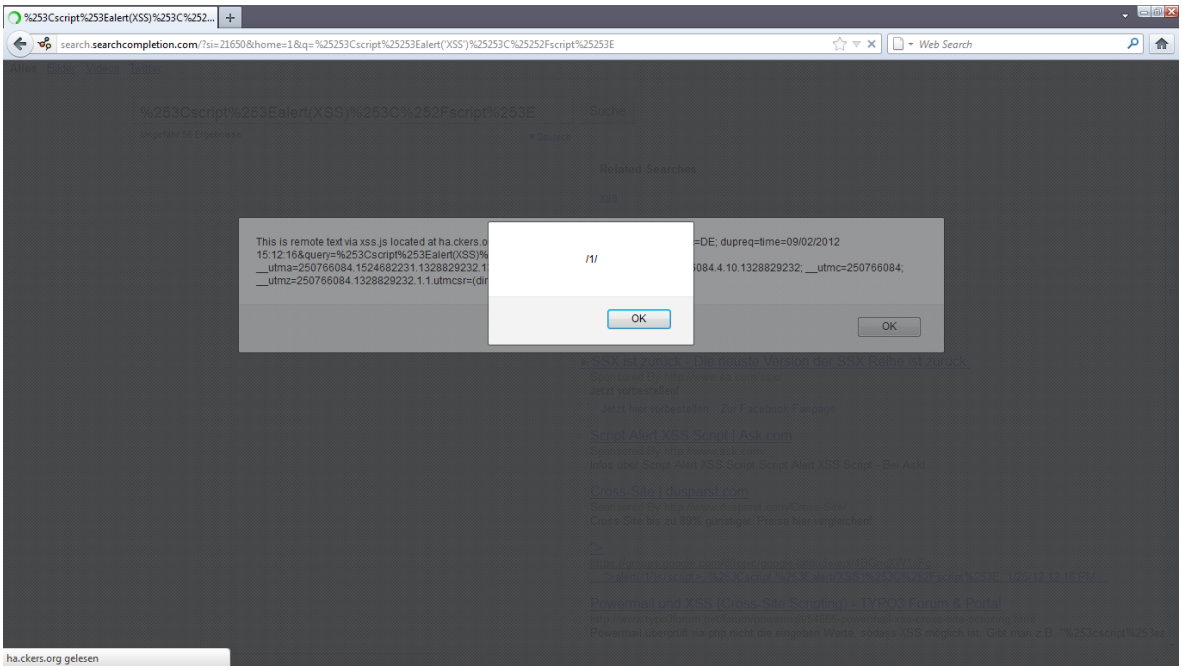


Figure 45: Double encoding result 1

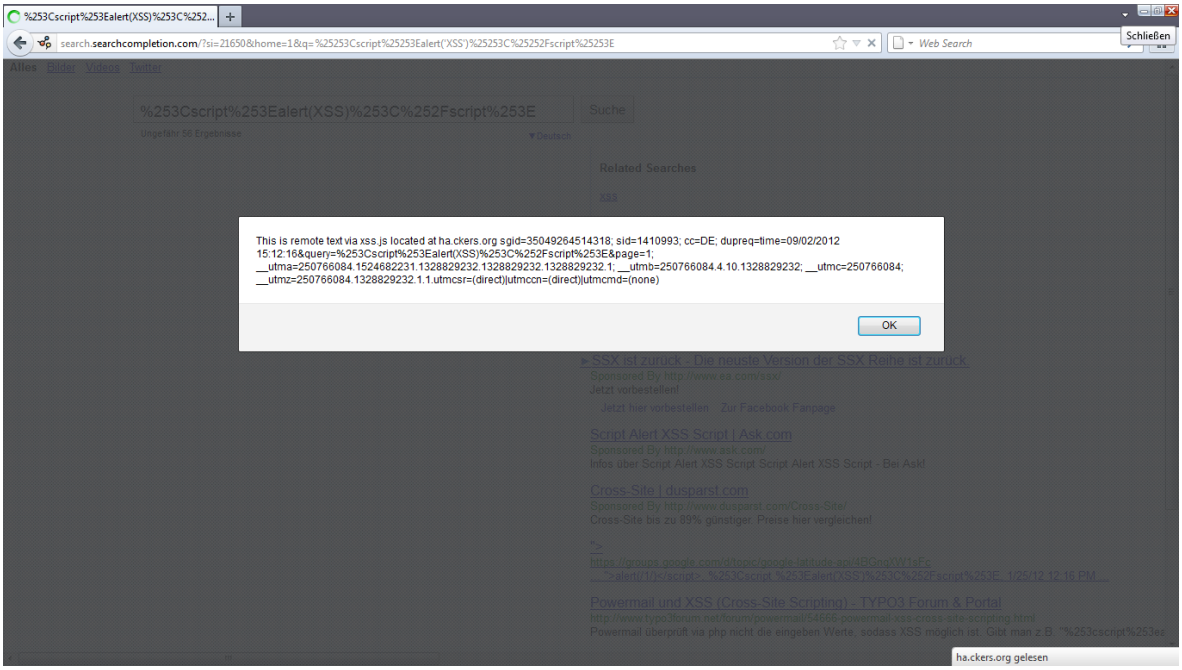


Figure 46: Double encoding result 2

Attack scenarios

Case study: Testing environment and educational scenario

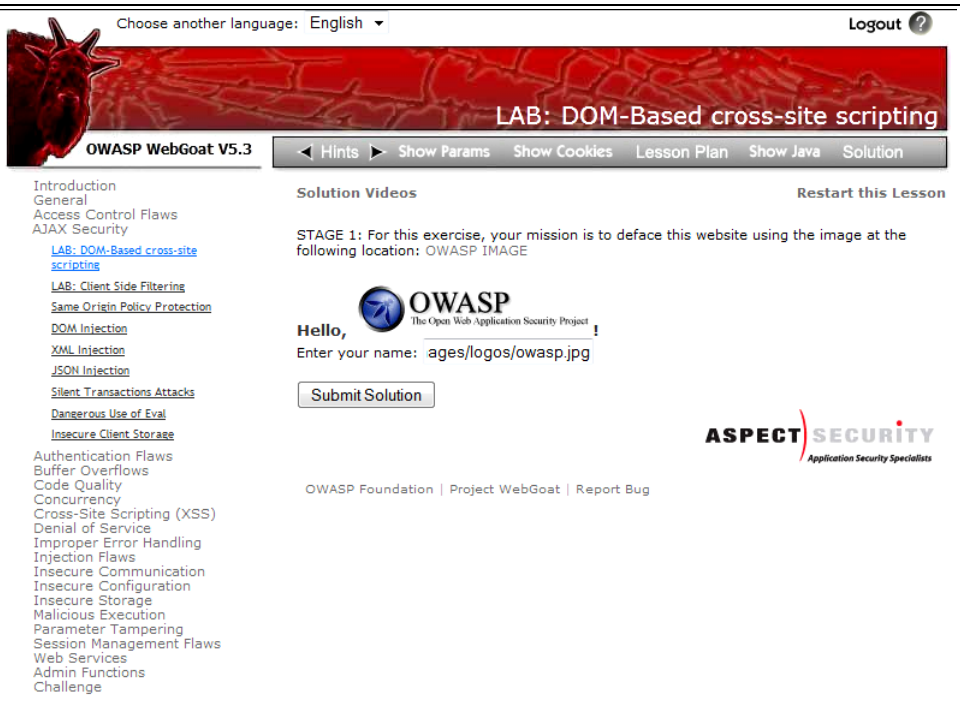


Figure 47: DOM-based cross-site scripting stage 1

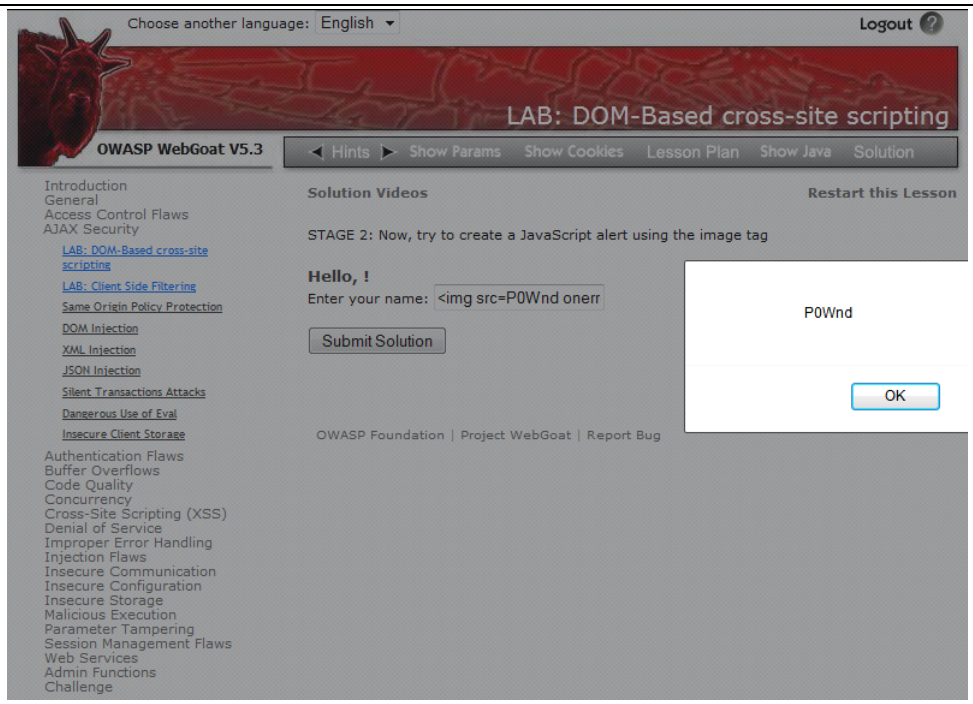


Figure 48: DOM-based cross-site scripting stage 2

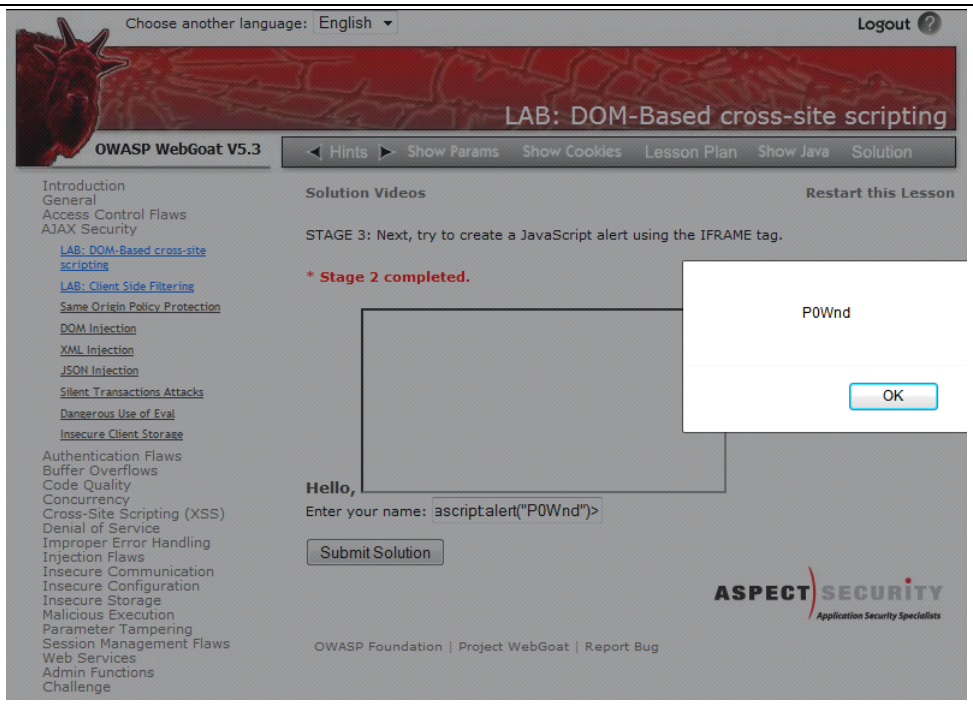


Figure 49: DOM-based cross-site scripting stage 3

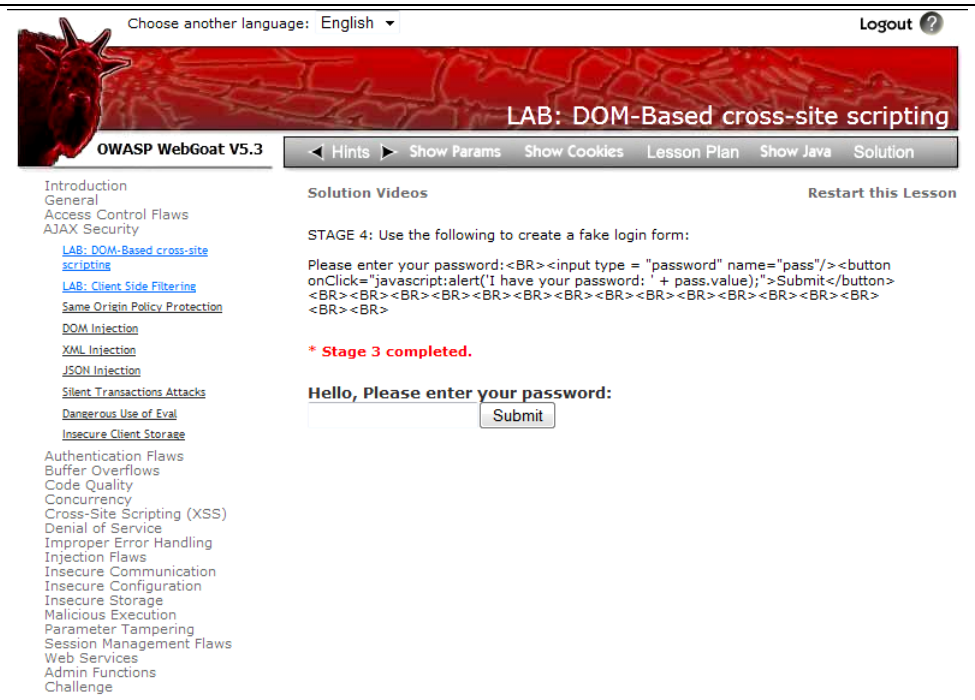


Figure 50: DOM-based cross-site scripting before stage 4

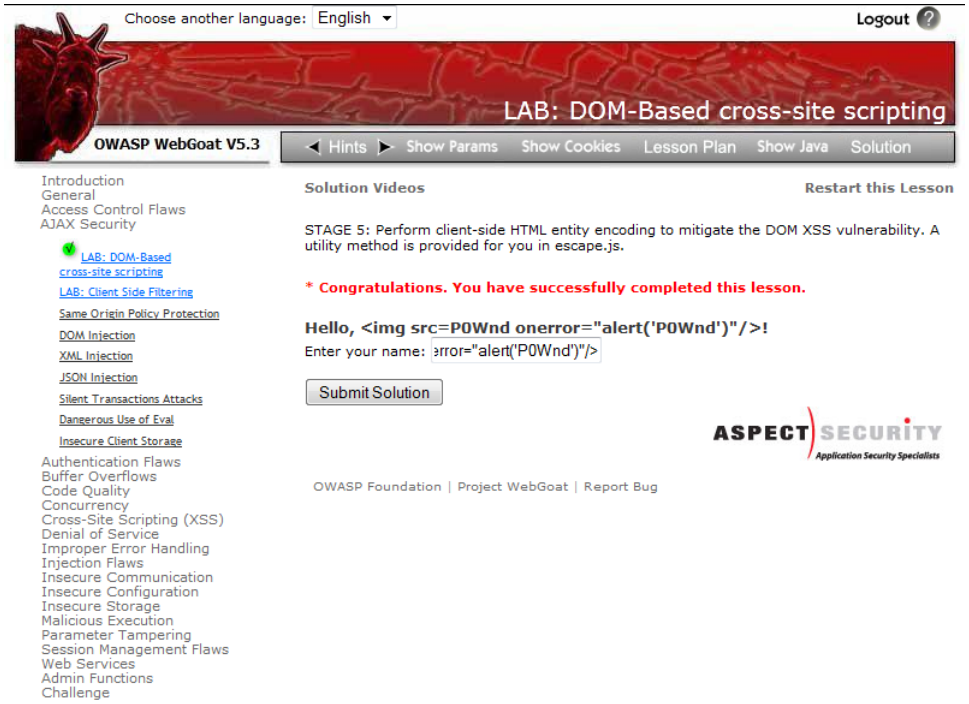


Figure 51: DOM-based cross-site scripting after stage 5

Case study: Real time scenario-AOL

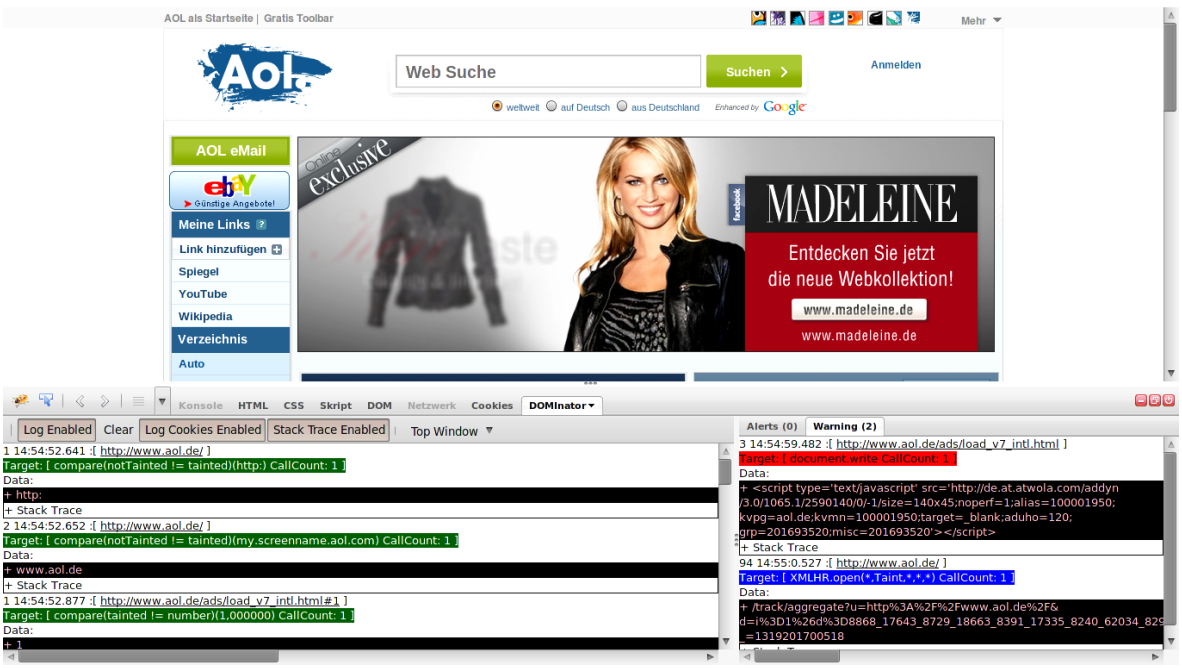


Figure 52: AOL scenario stage 1

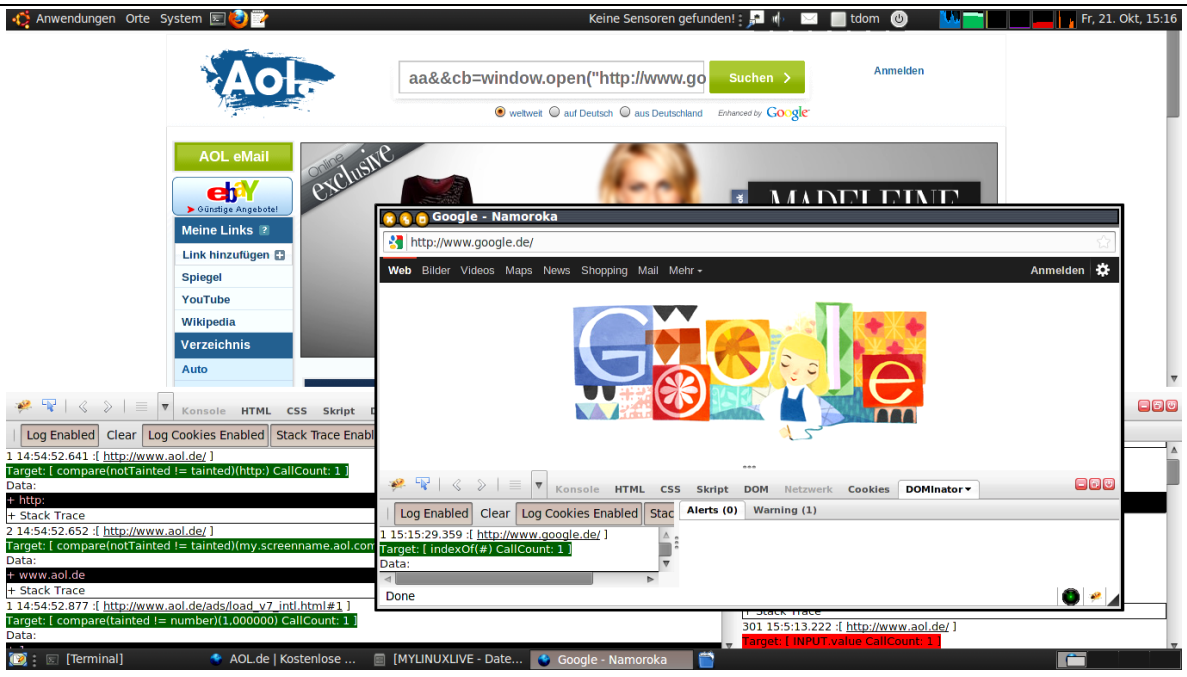


Figure 53: AOL scenario stage 3 enhancement

Scenario– Globo

S3 Meta-Model: Globo.com		
Prerequirements:		none
Core:	Source:	Input.Value
	Sink:	jquery.buscaPadrao.js
	Storage:	none
	Payload:	<img src=P0Wnd onerror=”alert('P0Wnd')”
Impact:		high

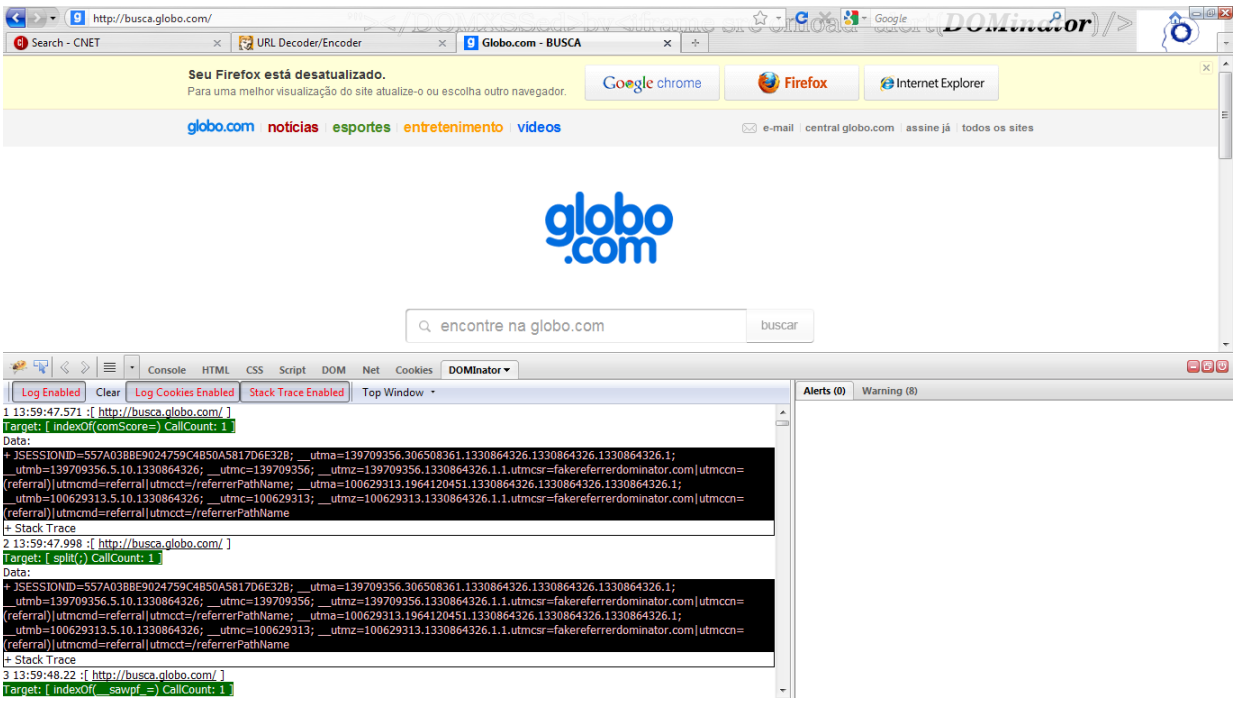


Figure 54: Globo scenario stage 1

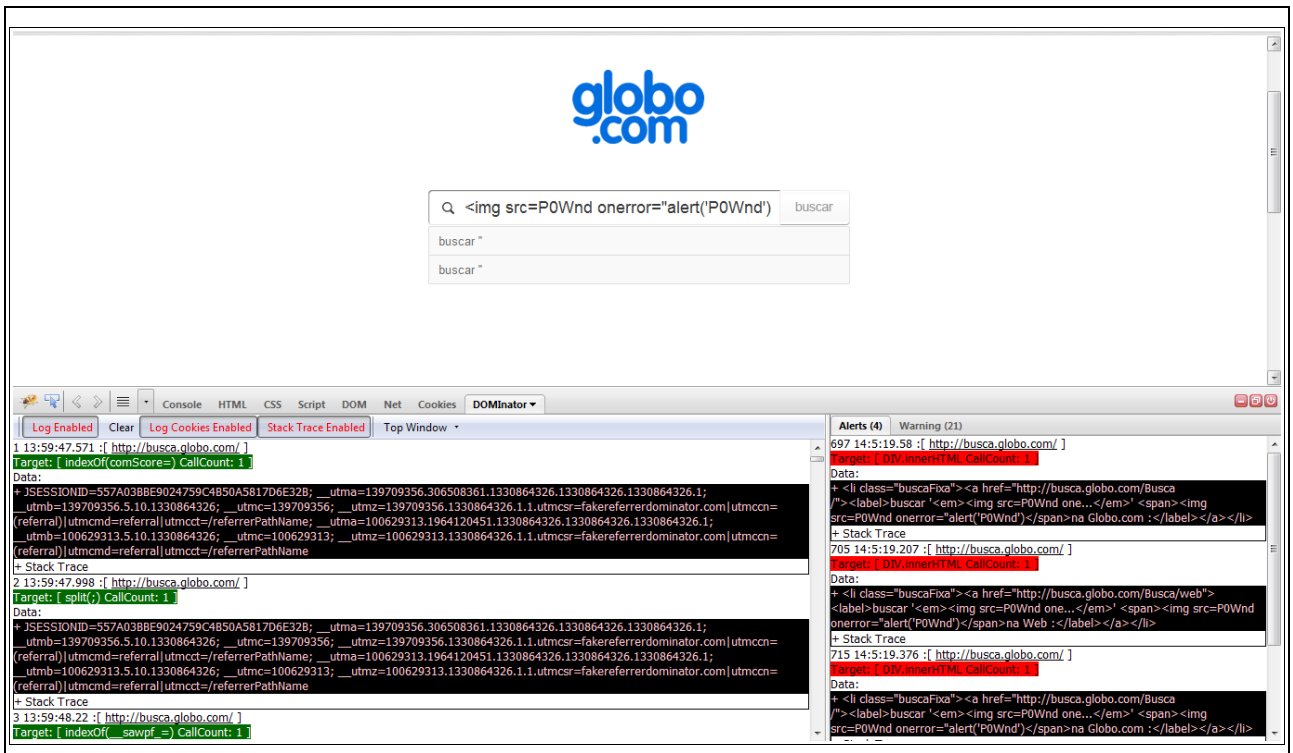


Figure 55: Globo scenario stage 2 – crafting the exploit (step1)

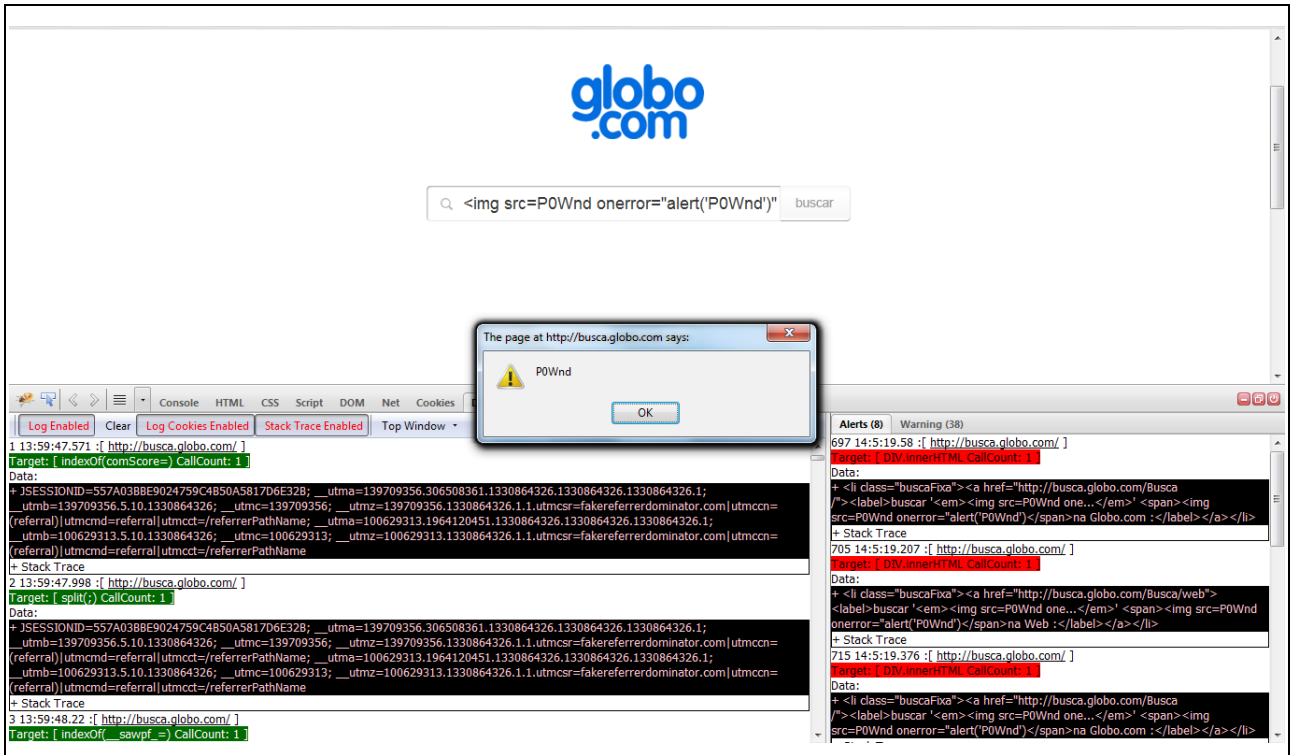


Figure 56: Globo scenario stage 2 – crafting the exploit (step2)

Chapter 4, related figures

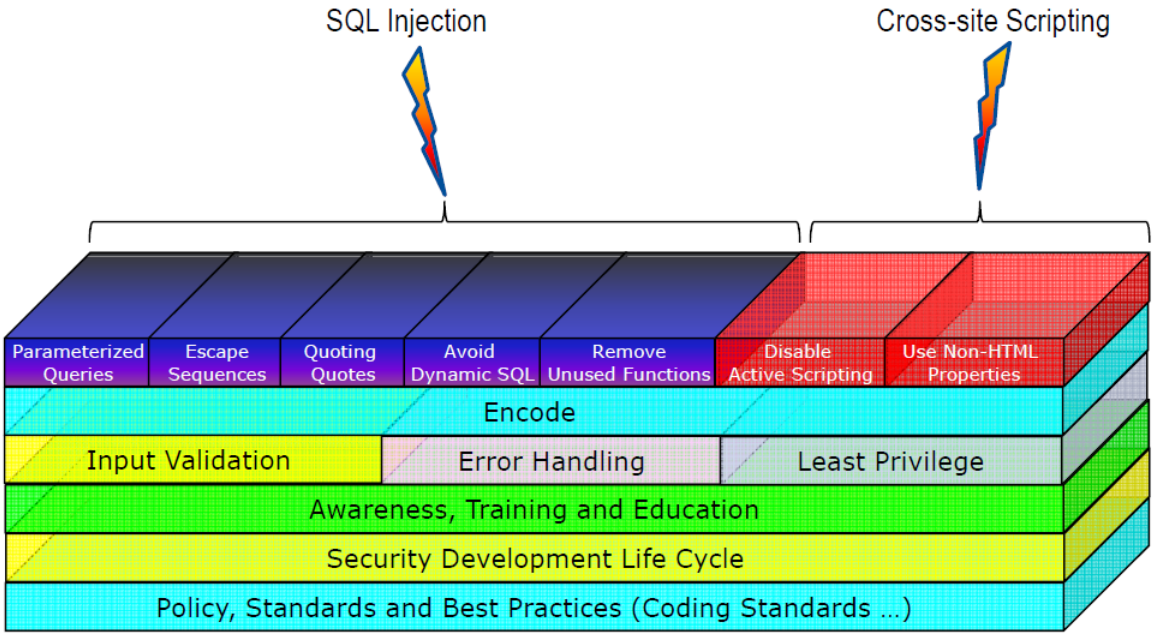


Figure 57: Web-Application complete Enterprise Security Model [L26]

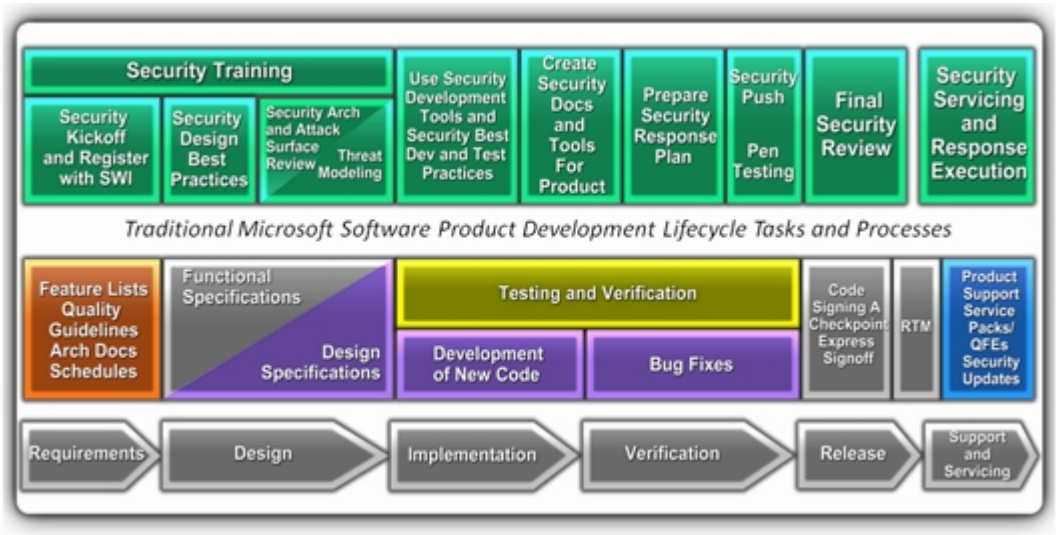


Figure 58: Microsoft Security Software Development Lifecycle (MS SSDLC) [L27]

JavaScript Coding Patterns

JavaScript Coding Patterns		
Type	**Class**	**Instance**
Essentials	Writing Maintainable Code	
	Minimizing Globals	Globals antipatterns
		Side Effects When Forgetting var
		Access to the Global Object
		Single var Pattern
		Hoisting: A Problem with Scattered vars
	for Loops	
	for-in Loops	
	(Not) Augmenting Built-in Prototypes	
	switch Pattern	
	Avoiding Implied Typecasting	Avoiding eval()
	Number Conversions with parseInt()	
	Coding Conventions	Indentation
		Curly Braces
		Opening Brace Location
		White Space
	Naming Conventions	Capitalizing Constructors
		Separating Words
		Other Naming Patterns
	Writing Comments	
	Writing API Docs	
	Writing to Be Read	
	Peer Reviews	
	Minify	
Literals and Constructors	Object Literal	The Object Literal Syntax
		Objects from a Constructor
		Object Constructor Catch
	Custom Constructor Functions	Constructor’s Return Values
	Patterns for Enforcing new	Naming Convention
		Using that

		Self-Invoking Constructor
	Array Literal	Array Literal Syntax
		Array Constructor Curiousness
		Check for Array-ness
	JSON	
	Regular Expression Literal	Regular Expression Literal Syntax
	Primitive Wrappers	
	Error Objects	
Functions	Background	Disambiguation of Terminology
		Declarations vs. Expressions:Names&Hoisting
		Function's name Property
		Function Hoisting
	Callback Pattern	Callback basic schema
		Callbacks and Scope
		Asynchronous Event Listeners
		Timeouts
		Callbacks in Libraries
	Returning Functions	
	Self-Defining Functions	
	Immediate Functions	Parameters of an Immediate Function
		Returned Values from Immediate Functions
		Benefits and Usage
	Immediate Object Initialization	
	Init-Time Branching	
	Function Properties- A Memoization Pattern	
	Configuration Objects	
	Curry	Function Application
		Partial Application
		Currying
		When to Use Currying
Object Creation Patterns Namespace Pattern	Namespace Pattern	General Purpose Namespace Function
	Declaring Dependencies	
	Private Properties and Methods	Private Members
		Privileged Methods

		Privacy Failures	
		Object Literals and Privacy	
		Prototypes and Privacy	
		Revealing Private Functions As Public Methods	
	Module Pattern	Revealing Module Pattern	
		Modules That Create Constructors	
		Importing Globals into a Module	
	Sandbox Pattern	A Global Constructor	
		Adding Modules	
		Implementing the Constructor	
	Static Members	Public Static Members	
		Private Static Members	
	Object Constants		
	Chaining Pattern		
	method() Method		
Code Reuse Patterns	Classical Inheritance Patterns	Expected Outcome	
		The Default Pattern	Following the Prototype Chain
			Drawbacks in Default Pattern
		Rent-a-Constructor	The Prototype Chain
			Multiple Inheritance by Borrowing Constructors
			the Borrowing Constructor Pattern
		Rent and Set Prototype	
		Share the Prototype	
		A Temporary Constructor	Storing the Superclass
			Resetting the Constructor Pointer
	Modern Inheritance Patterns		
	Klass		
	Prototypal Inheritance		
	Inheritance by Copying Properties		

	Mix-ins	
	Borrowing Methods	Borrow and Bind

Table 27: JavaScript Coding patterns [S10]

JavaScript Design patterns		
Type	**Class**	**Object**
Creational	Factory Method	Abstract Factory
		Builder
		Prototype
		Singleton
Structural	Adapter	Adapter
		Bridge
		Composite
		Decorator
		Facade
		Flyweight
		Proxy
Behavioral	Interpreter	Chain of Responsibility
		Command
		Iterator
		Mediator
	Template Method	Memento
		Observer
		State
		Strategy
		Visitor

Table 28: Representatives of JavaScript Design patterns [S10]

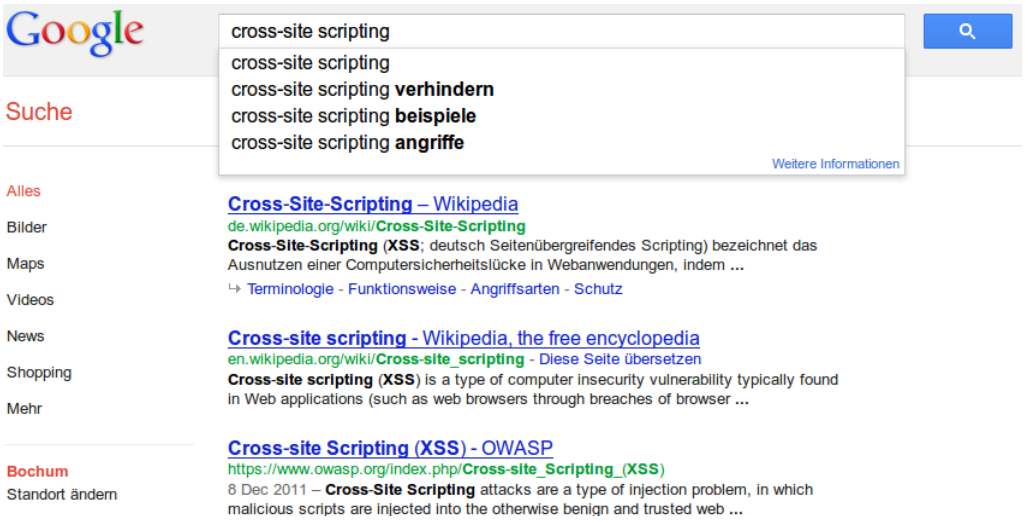
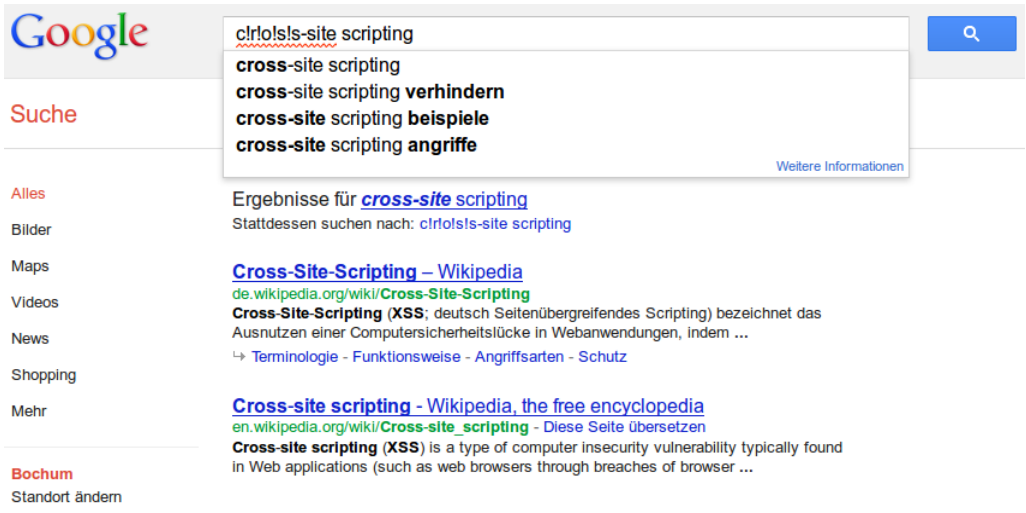
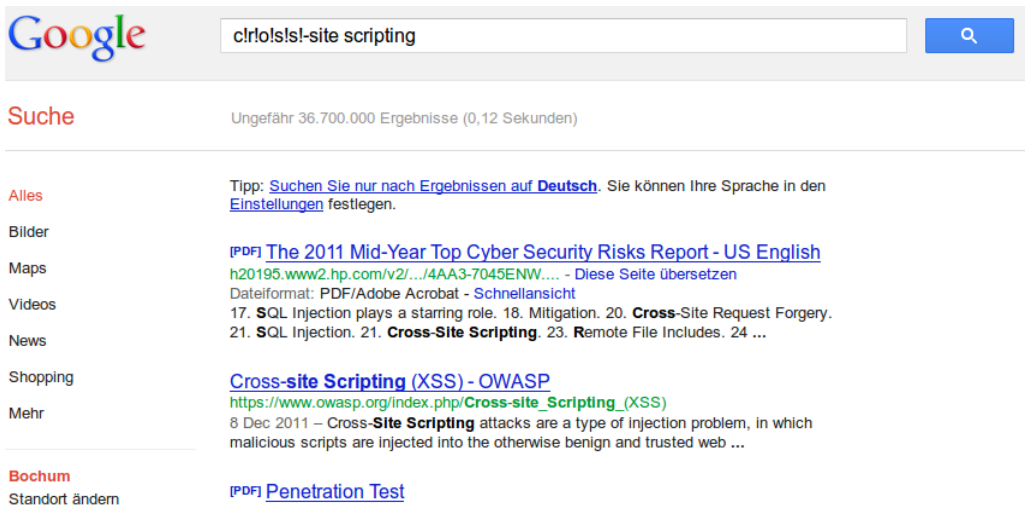
Security patterns	
Access control requirements	Enterprise partner communication
Asset valuation	Enterprise security approaches
Audit requirements	Enterprise security services
Audit trails and logging requirements	Security needs identification for enterprise assets
Authenticator	Execution domain
Authorization	Non-repudiation requirements
Automated i&a design alternatives	Intrusion detection requirements
Biometrics design alternatives	File authorization
Check point	Front door
Controlled execution environment	Security accounting requirements
Controlled object factory	Full access with errors
Controlled object monitor	I&A requirements
Controlled process creator	Information obscurity
Controlled virtual address space	Integration reverse proxy
Demilitarized zone	Known partners
Limited access	Multilevel security
Packet filter firewall	Role rights definition
Password design and use	Role-based access control
Protection reverse proxy	Secure channels
Proxy-based firewall	Security session
Reference monitor	Single access point
Risk determination	Stateful firewall
Vulnerability assessment	Threat assessment

Table 29: List of Security patterns [S06]

Filters

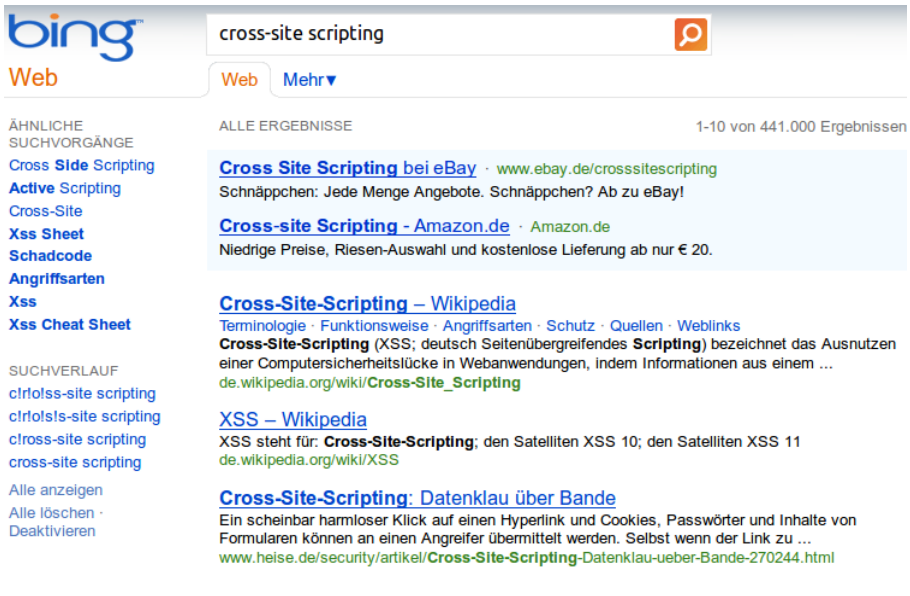
```
1.  <!--.....-->
2.  <form name="encodeHtml" action="">
3.      Enter the text to encode here:<br />
4.      &amp;nbsp;<textarea name="htmlToEncode" cols="100" rows="3"></textarea><br />
5.      <input type="button" name="action" value="Encode HTML" onclick="encodeMyHtml()"
        /><br />
6.      <br />Encoded html:<br />
7.      <textarea name="htmlEncoded" cols="100" rows="3"></textarea><br /><br />
8.      <input type="button" name="action" value="Test Encoded HTML"
onclick="testEncodedHtml()" /><br />
10.     <div id="testEncodedHtmlArea"></div>
11. </form>
12. <!--.....-->
```

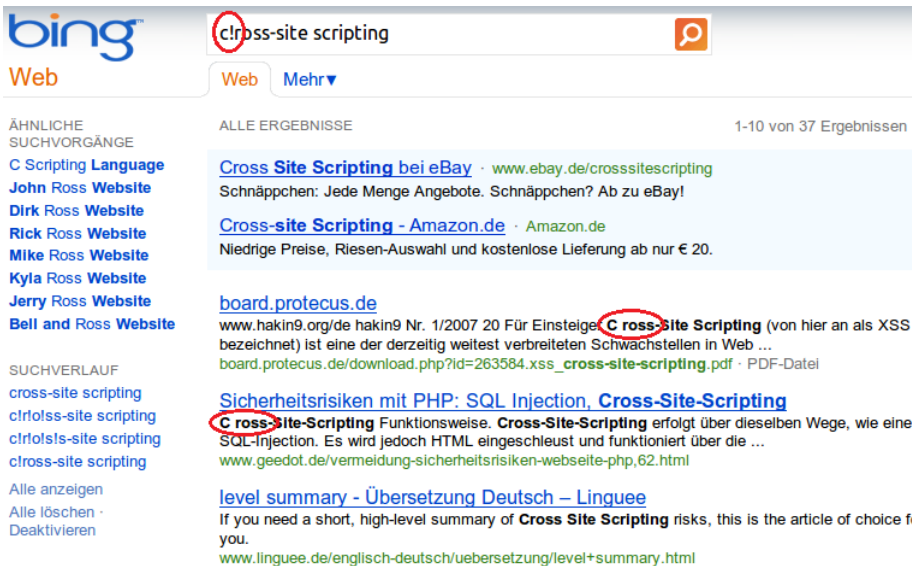
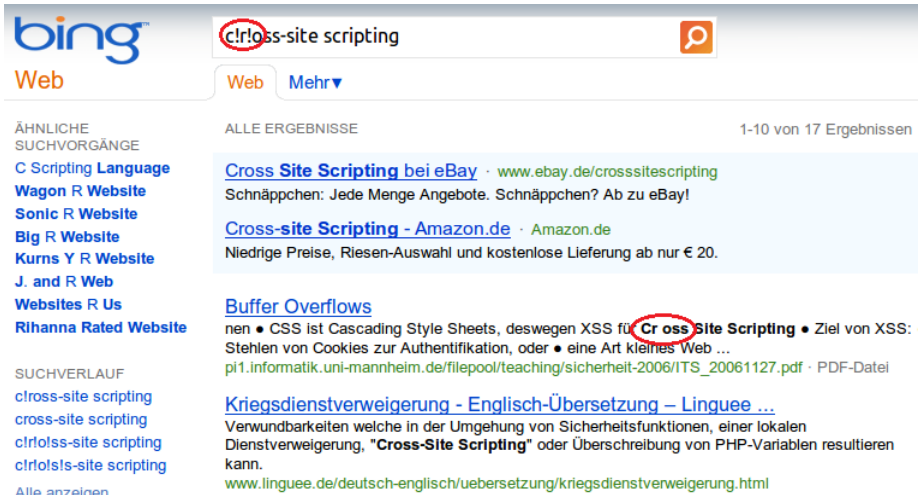
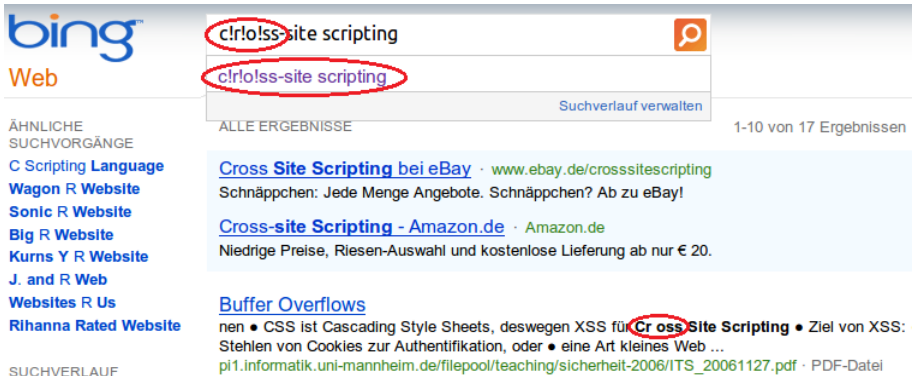
Table 30: Listing 1, example of HTML input form, e-mail filtering via JavaScript

Google search input filter escaping in 4 stages	
Stage 1	
Stage 2	
Stage 3	

Stage 4	

Table 31: Google search input filter escaping in 4 stages

Microsoft Bing input filter escaping in 5 stages	
Stage 1	

Stage 2	
Stage 3	
Stage 4.0, without activating the search button	

Stage 4	
Stage 5.0, without activating the search button	
Stage 5	

Table 32: Microsoft Bing input filter escaping in 5 stages

Tools

Figure 59: DOMXSS Scanner

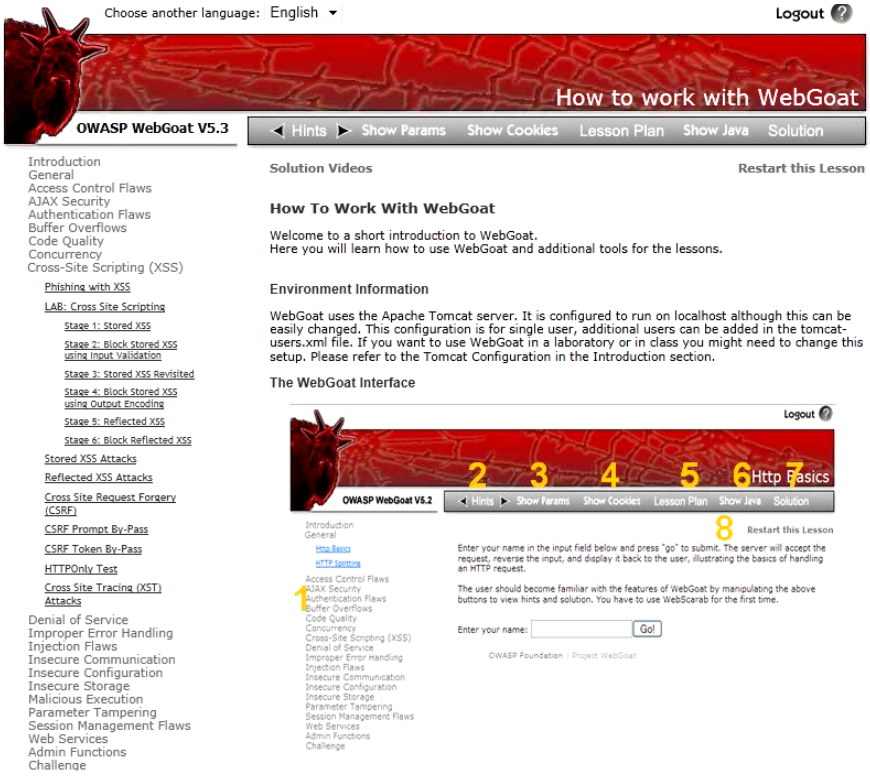


Figure 60: WebGoat Browser Interface

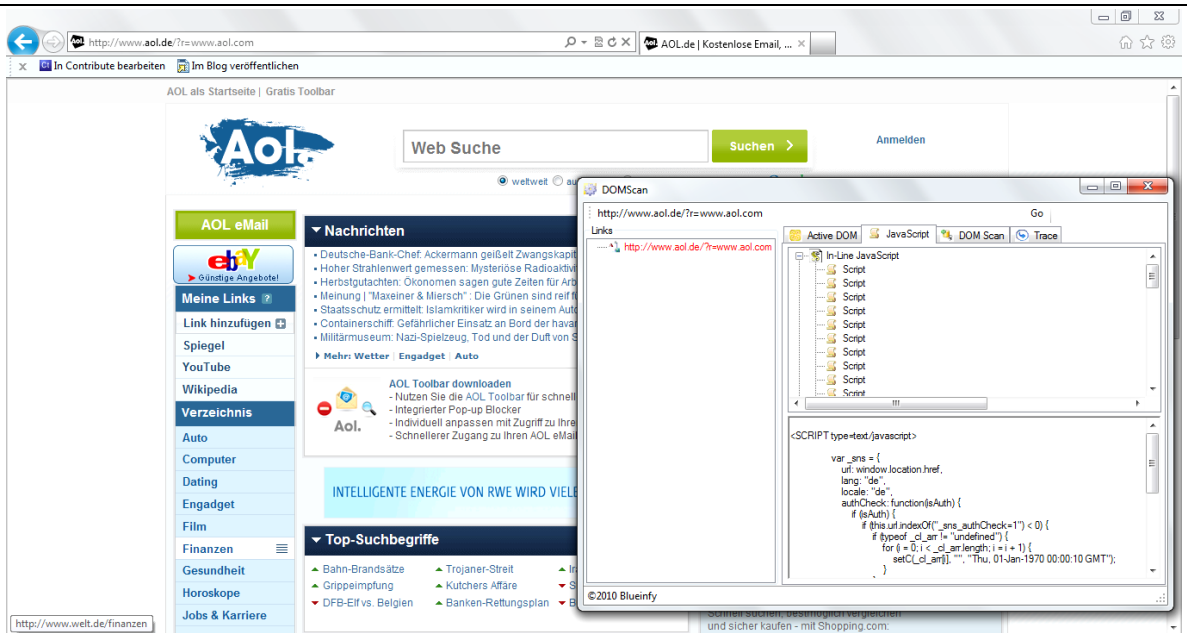


Figure 61: DOMScan Interface

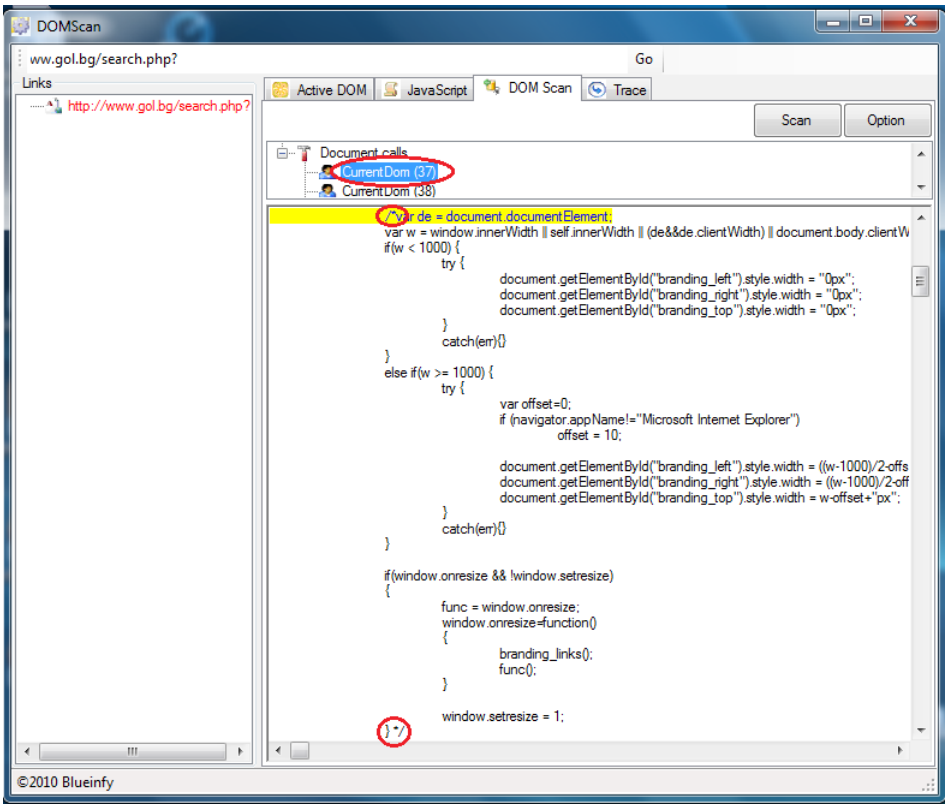


Figure 62: Drawback in DOMScan DOM calls detection

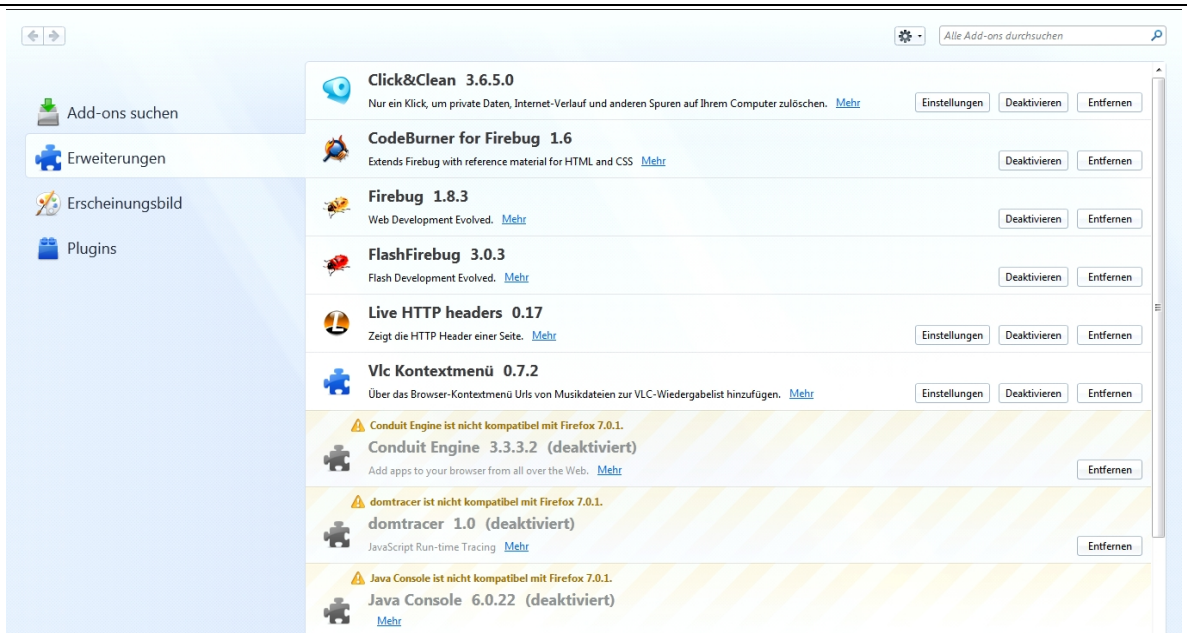


Figure 63: DOMTracer incompatible with newest version of Firefox

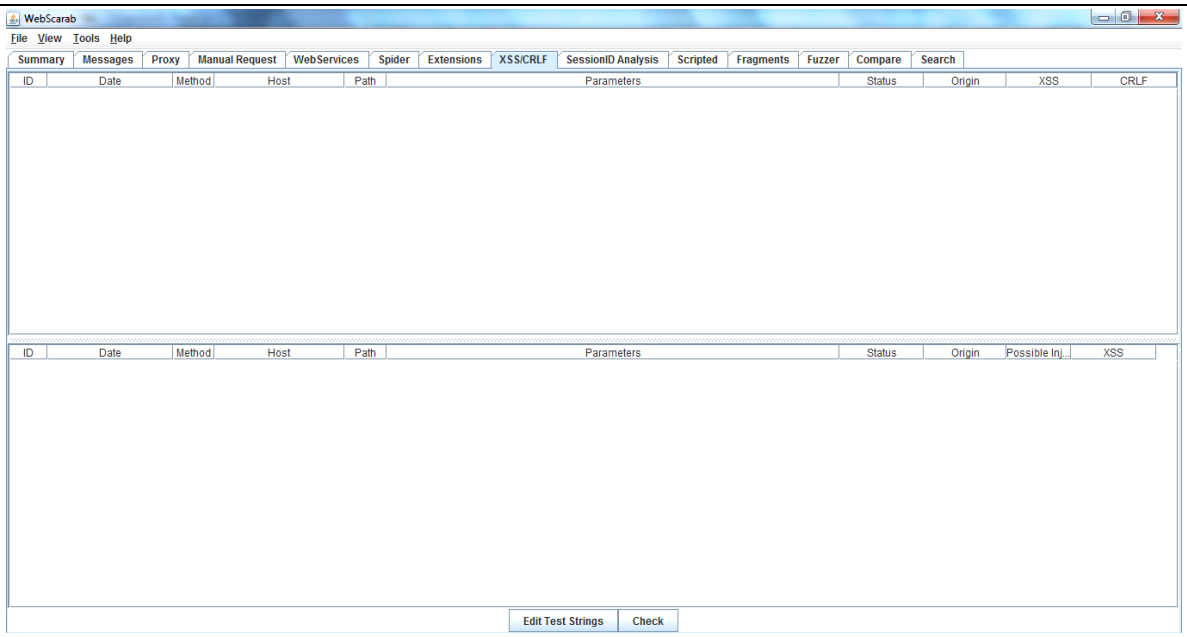


Figure 64: Web Scarab Interface

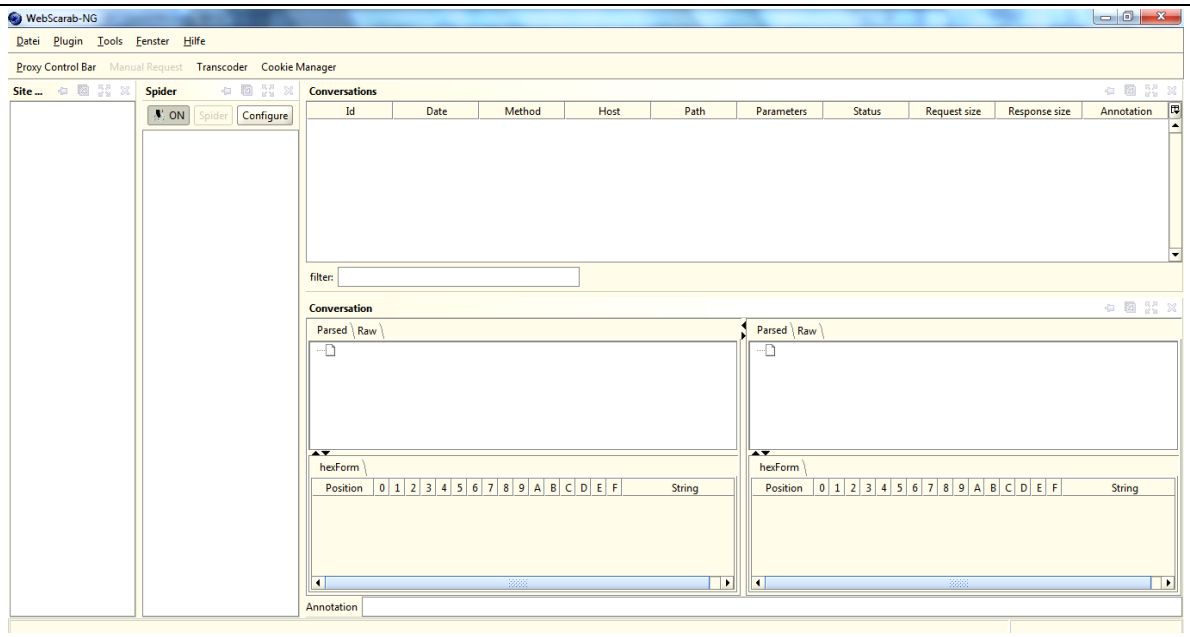


Figure 65: Web Scarab-NG Interface

simple view | collapse all | expand all | clear all | export all

Id	URL	Type
1	http://www.aol.de/?r=www.aol.com	Untrusted code
2	http://www.aol.de/?r=www.aol.com	Untrusted code
13	http://www.aol.de/?r=www.aol.com	Untrusted code
14+	**http://www.aol.de/?r=www.aol.com**	**Untrusted code**

Execute security checks for similar records ☑ | Show similar records | Export record

Security notes:

Loading of scripts from an untrusted origin.

Global ID:

http://www.aol.de/ads/load_v7_intl.html#SCRIPT

Document URL:

http://www.aol.de/ads/load_v7_intl.html

Data used:

URL:

```
http://de.at.atwola.com/addyn/3.0/1065.1/2590140/0/-1/size=140x45;noperf=1;alias=100001950;kvpg=aol.de;kvmn=100001950;target=_blank;aduho=120;grp=8590
98882;misc=859098882
```

HTML:

```
<script type="text/javascript" src="http://de.at.atwola.com/addyn/3.0/1065.1/2590140/0/-1/size=140x45;noperf=1;alias=100001950;kvpg=aol.de;kvmn=100001
950;target=_blank;aduho=120;grp=859098882;misc=859098882"></script>
```

31+	http://www.aol.de/?r=www.aol.com	Reflected input
40+	http://www.aol.de/?r=www.aol.com	Untrusted code

Figure 66: DOM Snitch activity log

Explanation of the proposed S3 Meta-Model

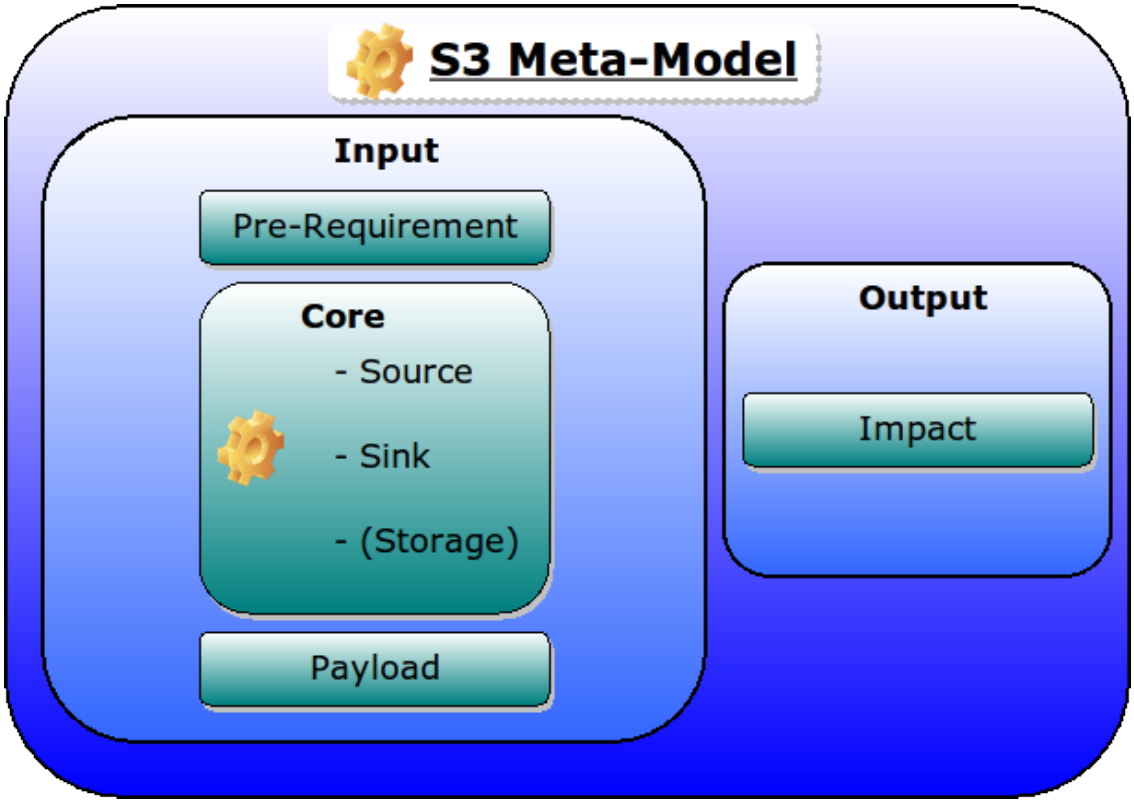


Figure 67: S3 Meta-Model

Explanation on the S3MM parameters:

- Prerequirement: denotes Browser version (Browser engine version) and other attack techniques, required for the successful execution of DOMXSS
- Core: represents the fundamental criteria, the DOMXSS workflow on the implementation layer of the Web-App logic can be described with, based on Stefano Di Paola and Amit Klein classifications
- Payload: denotes the exploit on itself, note that established P-XSS and NP-XSS payloads could be inherited.
- Impact: evaluates the attack severity, modification ability and attack propagation capabilities.

On behalf of our S3 Meta-Model, the following goals could be achieved:

- complete understanding on the attack workflow, concerning the Implementation environment
- if not all parameters could be evaluated, the attack should be considered for additional research.
- Standardization on attack description, respecting the Implementation environment Business logic of the Web-Application

- An easy descriptive model

Application Flow Analysis

Q/A TEAM	INFOSECURITY TEAM
Functions known	Functions unknown
Application understood	Application unknown
Rely on functional specifications	Rely on crawlers + experience + luck
Coverage known	Coverage unknown
Highlight key Business logic	Highlight “found” functionality

Table 33: Functional vs. Security testing [RL10]

EFD	Execution Flow Diagram – Functional paths through the application logic
ADM	Application Data Mapping – Mapping data requirements against functional paths

Table 34: Standards & Specifications of EFBs [RL10]

Graph(s) of flows through the application	Nodes represent application states
	Edges represent different actions
	Paths between nodes represent state changes
	A set of paths is a flow

Table 35: Basic EFD Concepts [RL10]

Execution Flow Action	Something that causes a change in state
	A human, server or browser-driven event
Action Types	Direct
	Supplemental
	Indirect

Table 36: Definition of Execution Flow Action and Action Types [RL10]

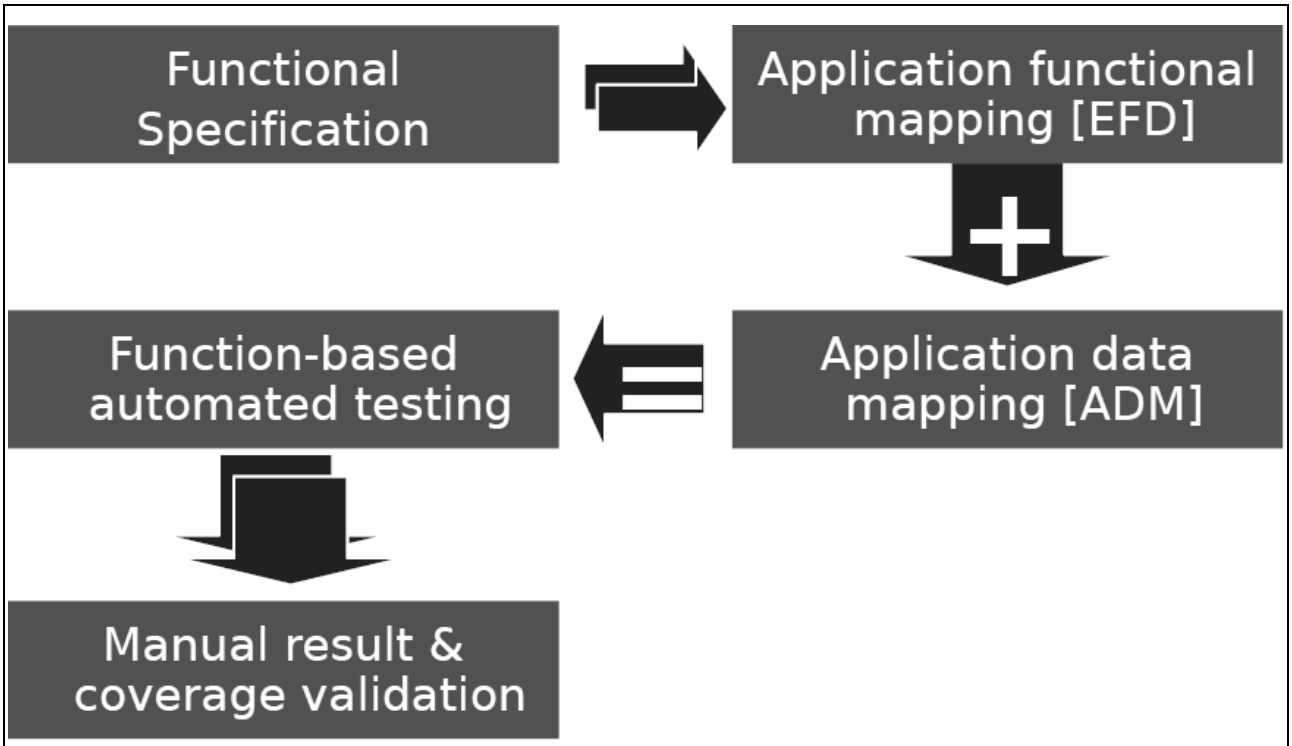


Figure 68: Improving the Testing process of Web Application Scanners [RL10]

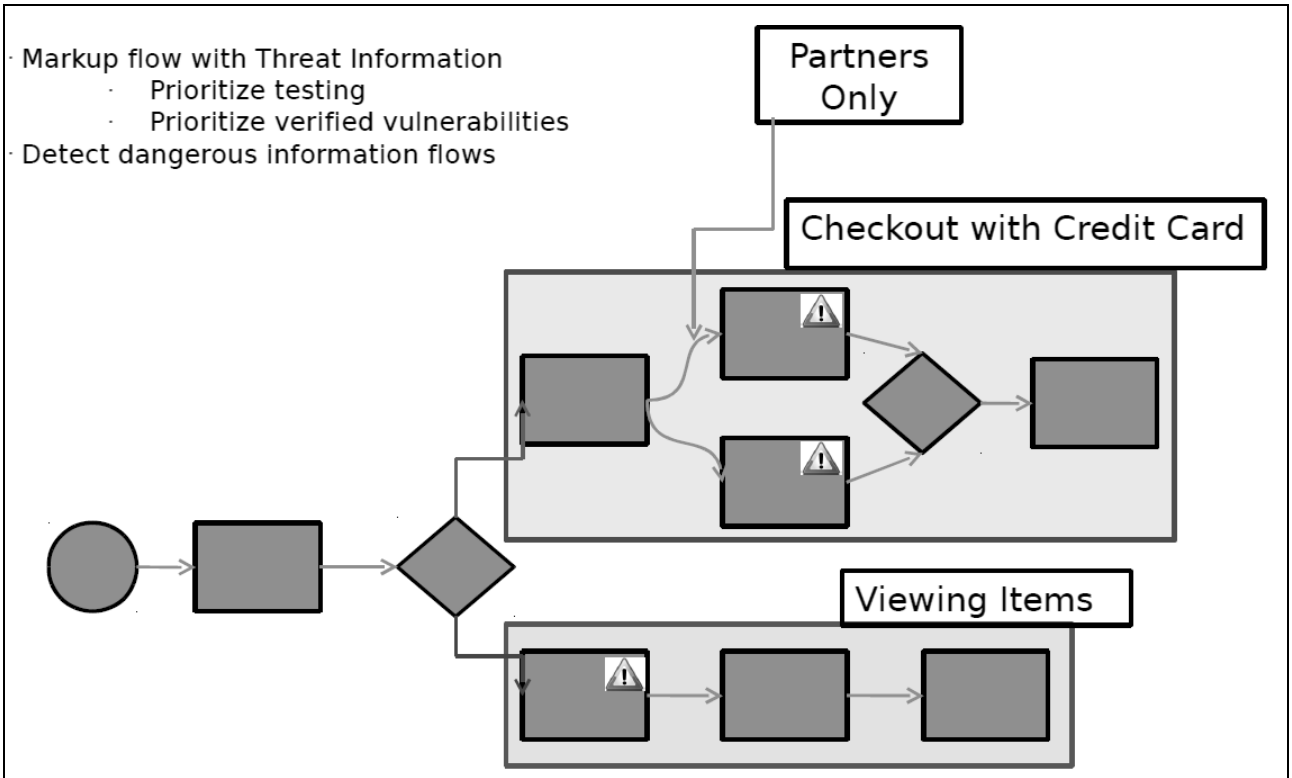


Figure 69: Flow based Threat Analysis, Example [RL10]

DOMXSS on mobile devices

iOS Paradigms

1. iOS v.5.0.1 (9A405)
2. Apple WebKit 534.46, Safari v. /5.1 Mobile/ 9A405/Safari/75334.48.3

3 Case studies:

- aol.de on safari

Figure 70: DOMXSS – aol.de on iOS v.5.0.1 (9A405)

S3 Meta-Model: aol.de on safari for iOS v.5.0.1 (9A405)		
Prerequirements:		none
Core:	Source:	Input.Value
	Sink:	jQuery v.1.3.2 (19.02.2009, out-of-date)
	Storage:	none
	Payload:	aa&cb=alert and aa&cb=window.open("http://www.google.de","_blank","FULLSCREEN")
Impact:		high

- ma.la[123] on safari

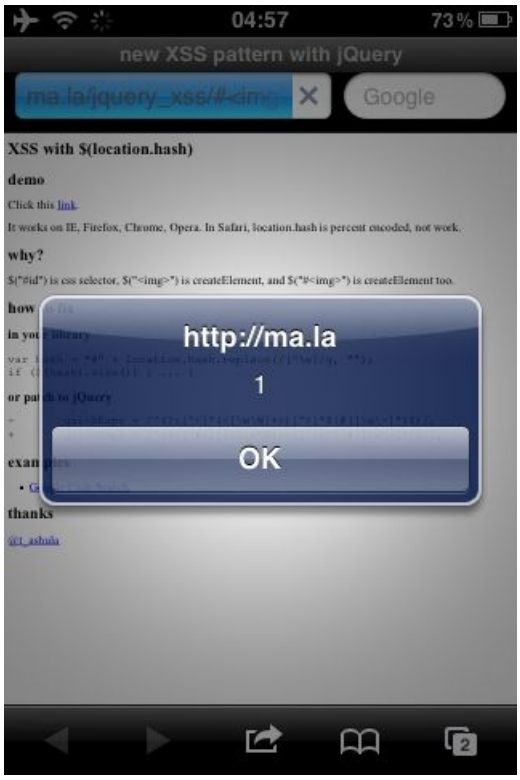


Figure 71: DOMXSS – ma.la on iOS v.5.0.1 (9A405)

S3 Meta-Model: ma.la on safari for iOS v.5.0.1 (9A405)		
Prerequirements:		none
Core:	Source:	location.hash
	Sink:	createElement
	Storage:	none
	Payload:	#<img src=/ onerror=alert(1)>
Impact:		high

123new XSS pattern with jQuery: http://ma.la/jquery_xss/

- aol.de error response on safari + disabled JavaScript

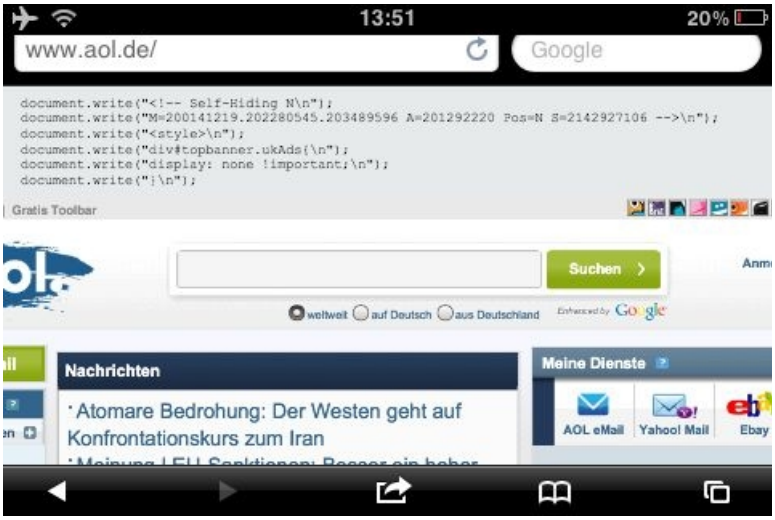


Figure 72: Error response on safari with disabled JavaScript – aol.de, concerning SbO

Google Android v.2.3.5

1. Android internal Browser: Mozilla/5.0 (Apple WebKit /533.1 version/4.0 mobile Safari/533.1)
2. Mozilla Firefox for Android: Mozilla/5.0 (Android; Linux armv71; rv:10.0) Gecko/20120129 Firefox/10.0 fennec/10.0

3 Case studies:

- aol.de with Android native Browser

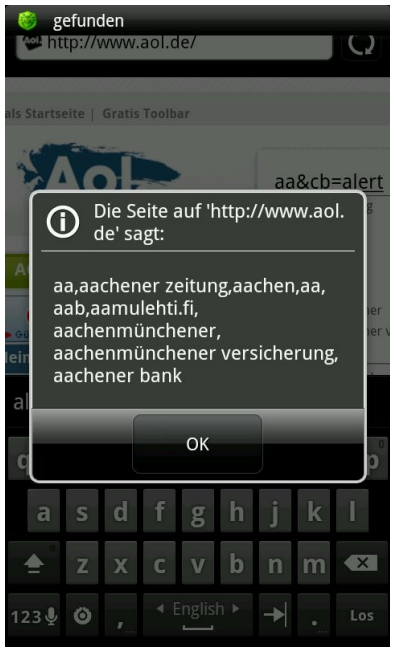


Figure 73: DOMXSS – aol.de on Android 2.3.5

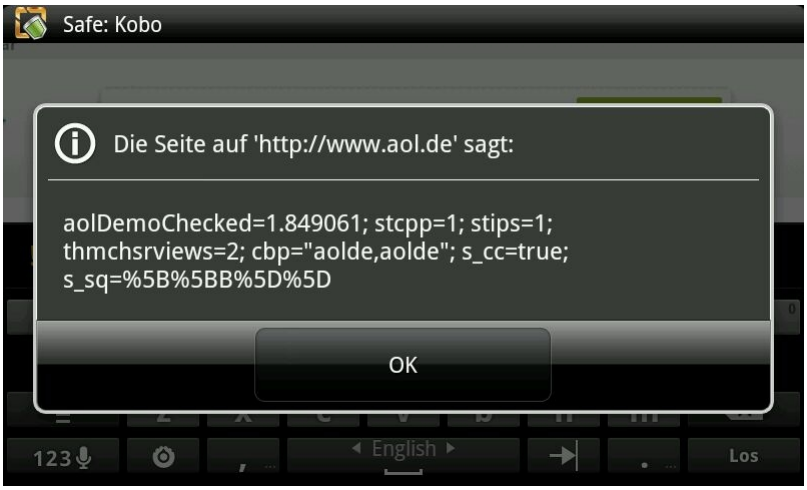


Figure 74: DOMXSS – aol.de on Android 2.3.5 with diocument.cookie payload

- ma.la with Firefox for Android

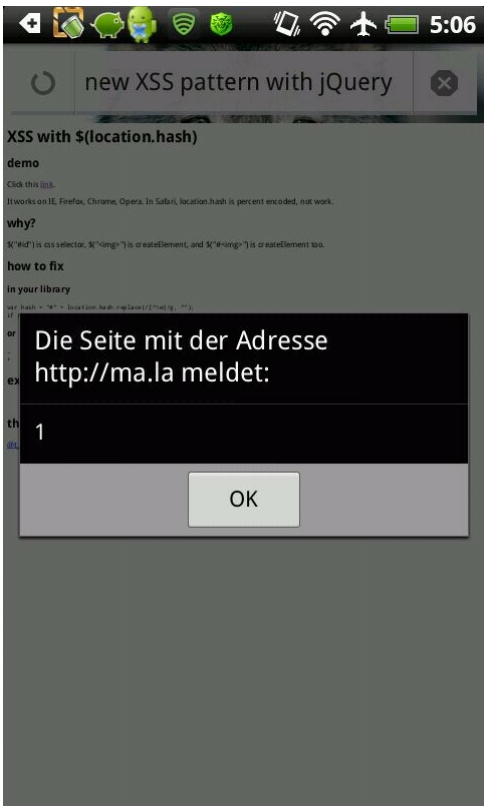


Figure 75: DOMXSS – ma.la on Mozilla Firefox for Android 2.3.5

- student.mit.edu with Android native Browser

Figure 76: DOMXSS – student.mit.edu on Android 2.3.5

S3 Meta-Model: student.mit.edu on Android 2.3.5 native Browser		
Prerequirements:		none
Core:	Source:	location.search
	Sink:	location.search
	Storage:	none
	Payload:	<IMG """><SCRIPT>alert(document.cookie)</SCRIPT>">
Impact:		high (3 times sanitized, the attack still works)

0day exploits on domxssscanner.com

Full disclosure of the 1st 0day exploit:

Scenario:

We decided to inspect the DOM XSS Scanner online tool for DOM-based XSS vulnerabilities, because the tool on itself represents a Web-App. We tested it via DOMinator and registered a possible Source: location.href, and a possible Sink at location.href.split(). As next, we decided to approve our results with DOMScan, which detected further several possible DOMXSS vulnerabilities.

That's why, we decided to test a PoC on the DOM XSS Scanner search input field, crafting a known exploit (Phase 1 of the attack scenario), demonstrated as in Figure 76.

Attack payload:

```
http://student.mit.edu/catalog/search.cgi?search=
<IMG """><SCRIPT>alert(document.cookie)</SCRIPT>">&style=verbatim
```

Vulnerable Web-Application Source Code:

Please, see Table 37.

Results:

The following figures represent the further deployment of the attack scenario, step-by-step:

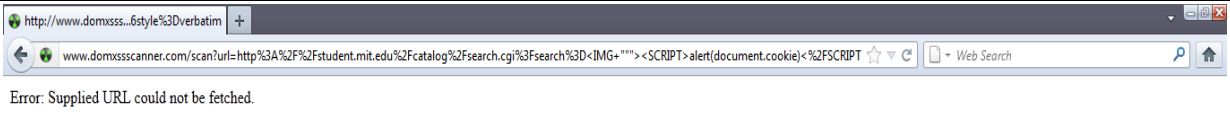


Figure 77: Phase 2: Error response

Figure 78: Phase 3: first Popup in Mozilla Firefox

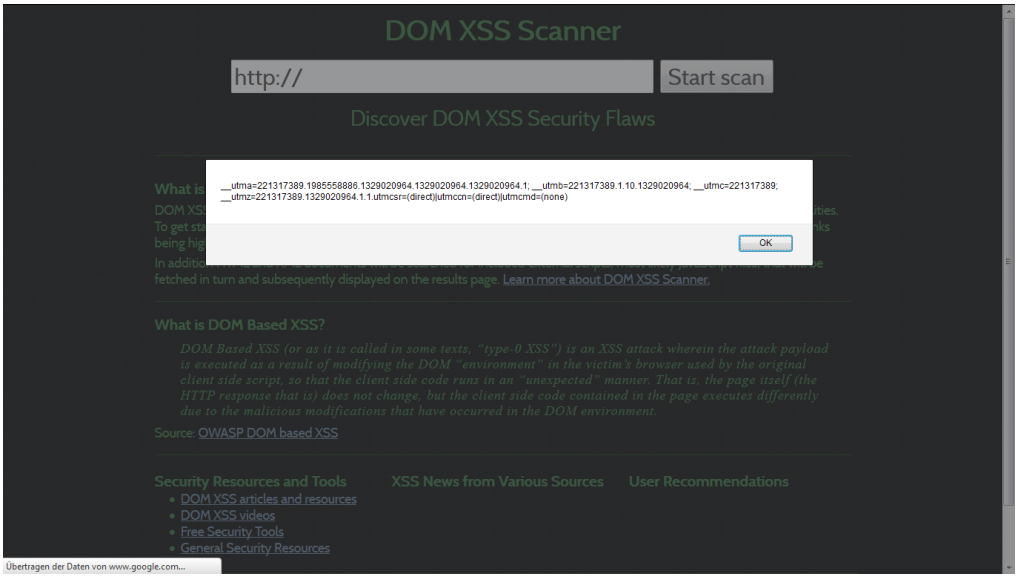


Figure 79: Phase 3: second Popup in Mozilla Firefox

Figure 80: Phase 3: Popup in Safari

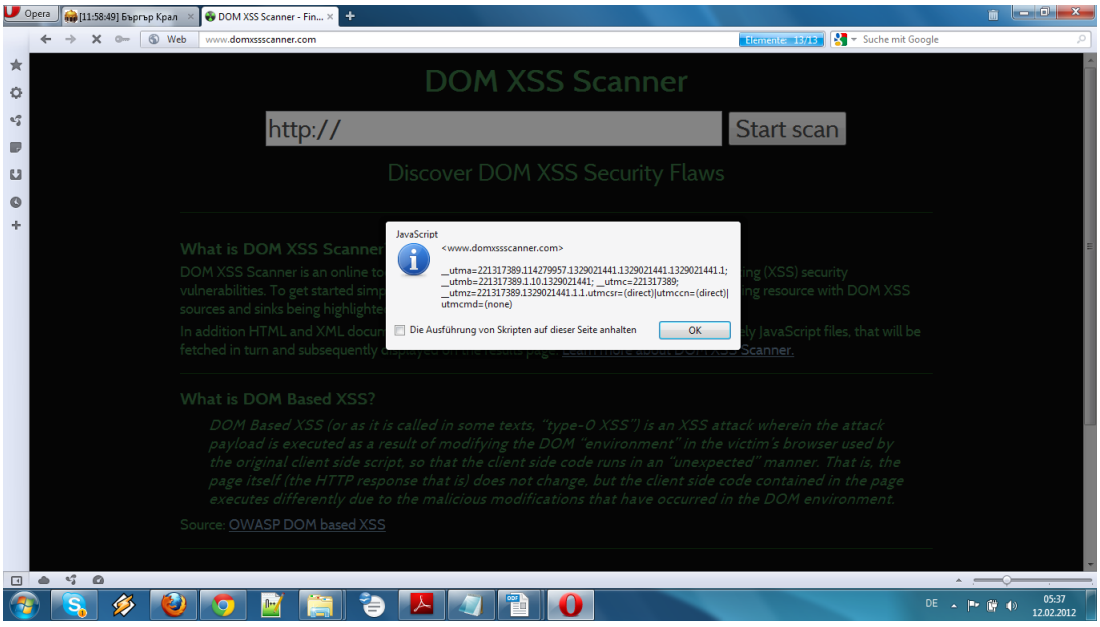


Figure 81: Phase 3: second Popup in Opera

Figure 82: Persistent XSS in the source code after successful injection

DOM XSS Scanner vulnerable code snippet after the exploit:

<script type="text/javascript">
var addthis_config = {
services_custom: {
name: "Flattr",
url: "http://flattr.com/submit/auto?url=http://student.mit.edu/catalog/search.cgi?search=<IMG """><SCRIPT>alert(document.cookie)</SCRIPT>">&style=verbatim&title=DOM XSS Scanner - Scan http://student.mit.edu/catalog/search.cgi?search=<IMG """><SCRIPT>alert(document.cookie)</SCRIPT>">&style=verbatim&user_id=rg&category=software&language=en_GB",
icon: "http://api.flattr.com/button/flattr-badge-large.png"}
}
</script>

DOM XSS Scanner code snippet after successive URL scanning:

<script type="text/javascript">
var addthis_config = {
services_custom: {
name: "Flattr",
url: "http://flattr.com/submit/auto?url=http://bornnoir.com&title=DOM XSS Scanner - Scan http://bornnoir.com&user_id=rg&category=software&language=en_GB",
icon: "http://api.flattr.com/button/flattr-badge-large.png"}
}
</script>

Table 37: 0day DOM XSS Scanner vulnerable code fragment at flattr.com module

Figure 82 and Table 37 demonstrate that the unsanitized code fragment in the DOM XSS Scanner allows P-XSS, which stores temporarily the attack payload, another example of DOMXSS, temporary until another Web-App URL is applied for scanning via the security tool.

Further deployment of the attack:

We decided to increase the attack impact by PoC on attack propagation. We applied the payload to the URL-shortening service– TinyURL!, and exchanged it via the instant Messenger – Skype. We experienced that after invoking the link, the attack executes successful in the default Browser, specified for the OS. This was tested under Windows 7 Home Premium and Ubuntu 10.10.

Evaluation of the results:

Browser	Recent version	0day on DOM XSS Scanner	Skype propagated TinyURL masked attack payload (straight attack)	Skype propagated TinyURL masked exploited DOM XSS Scanner by the attack payload
Mozilla Firefox	10/01/00	Successful, 2 Popups, 2. provides the client cookie	Successful, 2 Popups, 2. provides the client cookie	Successful, 2 Popups, 2. provides the client cookie

Opera	11.61 Build 1250	Successful, 2 Popups, 2. provides the client cookie	Successful, 2 Popups, 2. provides the client cookie	Successful, 2 Popups, 2. provides the client cookie
Google Chrome	17.0.963.46 m	Successful, 2 Popups, 2. provides the client cookie	Successful, 2 Popups, 2. provides the client cookie	Successful, 2 Popups, 2. provides the client cookie
Safari	5.1 (7534.50)	Successful Popup	sanitized	Successful Popup
IE	9.0.8112.16421	Successful Popup	sanitized	Successful Popup

Note, that the Browser Plugins:

- NoScript[124] (Mozilla Firefox, Seamonkey)
- NotScripts[125] (Google Chrome, Chromium, Iron)

successfully sanitize the 0day in default settings.

Impact:

The attack should be considered as severe, though we didn't applied DoS on the domxssscanner.com. Nevertheless if the attack payload is designed to access the XHR-object, it could be possible to deploy Session Hijacking, password stealing etc. Because of the easiness of the attack propagation and URL masquerading, the impact of the attack is increased and allows the Cross-Origin access. Even more, the attacker can higher the attack using CSRF, because of the facts: recent multi-tab Browser surfing and the easiness to involve unaware “Confuse Deputy“ End-user.

Disclosure of the 2nd 0day exploit:

1. To escape the <noscript>-tag sanitization of the AddThis Client API module – apply to the search field of the DOM XSS Scanner Web GUI:

http://</noscript>

2. Consequentially appears an Error Response.
3. Refresh the address bar entry with the: www.domxssscanner.com .
4. Proceed as explained in the full disclosure of 1st 0day exploit.

124 https://addons.mozilla.org/de/firefox/addon/noscript/
125 https://chrome.google.com/webstore/detail/odjhifogjcknibkahlpidmdajjpkkcfn