Large-Scale Detection of DOM-based XSS based on Publisher and Subscriber Model

Trong Kha Nguyen and Seong Oun Hwang
Department of Computer and Information Communications Engineering, Hongik University, Korea
nguyentrongkha92@gmail.com, sohwang@hongik.ac.kr

***Abstract*—Cross-site scripting (also referred to as XSS) is a vulnerability that allows an attacker to send malicious code (usually in the form of JavaScript) to another user. XSS is one of the top 10 vulnerabilities on Web application. While a traditional cross-site scripting vulnerability exploits server-side codes, DOM-based XSS is a type of vulnerability which affects the script code being executed in the clients browser. DOM-based XSS vulnerabilities are much harder to be detected than classic XSS vulnerabilities because they reside on the script codes from Web sites. An automated scanner needs to be able to execute the script code without errors and to monitor the execution of this code to detect such vulnerabilities. In this paper, we introduce a distributed scanning tool for crawling modern Web applications on a large scale and detecting, validating DOM-based XSS vulnerabilities. Very few Web vulnerability scanners can really accomplish this.**

I. Introduction

Nowadays, single page Web application (SPA) JavaScript frameworks provide a more fluid user experience similar to a desktop application. An SPA is an application delivered to the browser that does not reload the page during use. Browsers render, generate HTML Web pages and adapt to business logic [1]. However, this process will be a potential vulnerability if input is not processed properly. One of the well-known attacks on the client-side is Cross-site scripting (XSS) which impacts greatly on user credentials [2], even controls users browser [3].

DOM XSS is a type of XSS attack which relies on inappropriate handling, in the HTML page, of the data from its associated DOM (Document Object Model). Among the objects in the DOM, there are several which the attacker can manipulate in order to generate the XSS condition. The DOM XSS attack is difficult to detect by server-side attack detection and prevention tools, because the malicious payload usually does not get to the server and hence cannot be sanitized by the server-side code, like in the case of other XSS attacks.

In order to detect Web vulnerability as well as DOM-XSS, we usually use Web vulnerability scanners. Input of a Web scanner is a seed URL. It crawls and scans the rest of Web pages from seed URLs to find Web vulnerability. Crawling step is important because it affects correctness of a Web scanner [4]. However, current techniques used in open-source Web vulnerability scanners are HTML parser and AJAX crawling which cannot see any dynamically created content [5]. Nowadays, one Web site has many URLs. When a target expands to many Web sites, it needs a cooperative system scanner to reduce the cost of scanning. In this case, distributed systems can solve very well.

We present a Web vulnerability scanner which has the following features:

- Our system can improve distributive DOM-based XSS detection for a Web scanner by using hooking technique in both crawling and scanning steps. Furthermore, by testing on a browser, our system reports no false positives.
- By using publisher and subscriber model, our system is not only scalable in deployment but also flexible to support scanning of other vulnerabilities. It is particularly suitable for crawling on a large scale.
- We provide an attack vector database where people can search their Web site and fix vulnerabilities to make Internet safer.

II. Background

Web Browser is a client-side application, and its basic function is to fetch the content from the Web servers and displays it in the browser's windows. Web browsers implement HyperText Transfer Protocol (HTTP) and its secure version (HTTPS), that specifies the form of requests and responses between the browser and the Web server. Every HTTP request to a server contains several HTTP headers: a domain and path to access the server, a content access method, and other kind of data. The Web server responds with an HTTP response containing the state (such as ”200 OK”), the content requested, and other headers.

Once the content is fetched from the server by means of this protocol, it is displayed by the browser. Originally Web pages are simple HTML pages containing simple elements such as paragraphs, buttons, and input boxes etc. With the evolution of the Web, new type of Web applications appeared *mashups*. These applications include content from multiple sources. For example, a housing rental Website combines the information about the houses and maps them to Google maps. Internally, the browser implements the Document Object Model (DOM), that is, a tree representation of the fetched Web pages.

HTML markup injection vulnerabilities are one of the most significant and pervasive threats to the security of Web applications. They arise whenever, in the process of generating HTML documents, the underlying code inserts attacker-controlled variables into the output stream without properly screening them for syntax control characters. Such a mistake allows a party controlling the offending input to alter the structure - and thus the meaning - of the produced document.

978-1-5090-5510-4/16 $31.00 © 2016 IEEE
DOI 10.1109/CSCI.2016.186

In practical settings, markup injection vulnerabilities are almost always leveraged to execute attacker-supplied JavaScript code in the client-side browsing context associated with the vulnerable application. The term cross-site scripting, a common name for this class of flaws, reflects the prevalence of this approach.

The JavaScript language is popular with attackers because of its versatility, and the ease with which it may be employed to exfiltrate arbitrarily chosen data, alter the appearance of the targeted Web site, or to perform server-side state changes on behalf of the authenticated user. Consequently, most of the ongoing browser-level efforts to improve the security of Web applications focus on the containment of attacker-originating scripts. The most notable example of this trend is undoubtedly the Content Security Policy, a mechanism that removes the ability to inline JavaScript code in a protected HTML document, and maintains a whitelist of permissible sources for any externally-loaded scripts. Several related approaches, such as the NoScript add-on, the built-in XSS filters in Internet Explorer and Chrome, client-side APIs such as toStaticHTML, or the HTML sanitizers built into server-side frameworks, also deserve a note.

In order to protect from the attacks like XSS, W3C proposed a Content Security Policy (CSP). This policy should be written by a Web page developer and contains a set of sources from which the remote content can be loaded and executed on the page. CSP applies to scripts, objects, style sheets, images, media, iframes, and fonts. Since CSP is oriented on disallowing to fetch the content from the forbidden sources, the enforcement of CSP described in the draft does not forbid to make an HTTP request, just the HTTP response should be seen by the browser as an empty response. It does not seem a completely secure solution since the HTTP requests (that may contain some information belonging to the user) can anyway be sent to arbitrary remote servers.

One of the most rudimentary goals of a successful XSS attack is the extraction of user-specific secrets from the vulnerable application. Historically, XSS exploits sought to obtain HTTP cookies associated with the authenticated user session; these tokens - a form of ambient authority - could be later replayed by the attacker to remotely impersonate the victim within the targeted site.

In an application where theft of HTTP cookies is not practical, exfiltration attempts are usually geared toward the extraction of any of the following:

- Cookie stealing may be the ultimate goal of an attack because it is a personal data. In applications such as Web mail systems, shopping or banking sites, the information about the user may be of immense value on its own merit.
- Tokens are used to defend against cross-site request forgery attacks. Under normal circumstances, any Web site loaded in the victim's browser is free to blindly initiate cross-domain requests to any destination. Because such a request is automatically supplanted with user's ambient credentials, it is difficult to distinguish it from a request that arises in response to a legitimate user action. To prevent malicious interference, most Web sites append session-specific, unpredictable secrets - XSRF tokens - to all GET URLs or POST request bodies that change the state of user account or perform other disruptive tasks.
- Location hijacking. An untrusted JavaScript code can either influence the document's location directly or the string variables that are forming a URL. As a result, the script can navigate the page to a malicious Web site without the knowledge of the user.
- History sniffing. In this attack a malicious script can check whether the user has ever visited a specific URL. JavaScript code can create an invisible link to the target URL and then use the browser's interface to check how this link is displayed. Since the browser displays visited and unvisited links in different colors, JavaScript program can deduce whether the URL has been visited by the user or not.
- Behavior tracking. A script running on a Web site can gather precise information about the user's mouse clicks and movements, scrolling behavior, what parts of the page are highlighted, and clipboard content. Such capability of JavaScript can be useful for the user's interaction with the Web site. However, a malicious script can also leak this information to the remote servers.

III. DOM-based XSS Attacks

XSS is a vulnerability that allows an attacker to send malicious code (usually in the form of JavaScript) to another user. This vulnerability is being used more and more in real world attacks and can have a very damaging impact on affected Web sites. XSS is one of top 10 vulnerability on Web application [6]. An attacker can steal or hijack session, carry out very successful phishing attacks and effectively can do anything that the victim can.

While a traditional XSS vulnerability exploits server-side code, DOM-based XSS is a type of vulnerability which affects the script code being executed in the client's browser [7]. DOM-based XSS simply means a Cross-site scripting vulnerability that appears in the DOM (Document Object Model) instead of part of the HTML. The DOM is a convention for representing and working with objects in an HTML document. Basically all HTML documents have an associated DOM, consisting of objects representing the document properties from the point of view of the browser. Whenever a script is executed on the client-side, the browser provides the code with the DOM of the HTML page where the script runs, thus, offering access to various properties of the page and their values, populated by the browser from its perspective.

DOM-based XSS vulnerabilities are much harder to detect than classic XSS vulnerabilities because they reside on the script code from the Web site. An automated scanner needs to be able to execute the script code without errors and to monitor the execution of this code to detect such vulnerabilities. Very few Web vulnerability scanners can really accomplish this. In comparison, classic XSS vulnerabilities are easier to detect as the detection process does not require the capability of executing and monitoring script code.

In software, data flow can be thought as in water flow in aqueduct systems which starts from natural sources and ends to sinks. In software security the sources are to be considered starting points where untrusted input data is taken by an application.

Sinks are meant to be the points in the flow where data depending from sources is used in a potentially dangerous way resulting in loss of confidentiality, integrity or availability (the CIA triad).

DOM XSS source:

Name	Function
location	document.URL document.documentURI document.URLUnencoded document.baseURI location location.href location.search location.hash location.pathname
cookie	document.cookie
Referrer	document.referrer
Window Name	window.name
History	history.pushState() window.onpopstate
Indirect	localStorage sessionStorage IndexedDB Database
Other	opener parent/top/frames[i].obj event.data of onmessage event

DOM XSS sink:

- Direct Execution sinks
- Set Object Sinks
- HTML Manipulation sinks
- Style sinks
- XMLHttpRequest Sink
- Set Cookie sink
- Common JavaScript libraries

IV. Technical Challenges

In this section, we discuss some challenges in order to achieve the following goals.

- Design a distributed scanning tool, can crawl modern Web application which is hard [8]. Like the other crawlers, the crawler component in our system needs to deal with characteristics: scalable, distributed data structure.
- Design a sematic-preserving JavaScript tracing system which is a technical challenge in our experiences. Data flow analysis is not easy. When I scan Web site, we have to preserve the context of program without change anything.

V. Distributed scanning system

Figure 1 illustrates the publisher and subscriber design overview of our system consist of 3 components: crawler, scanner engine and report. In this section we detail 3 components of our tools.

A. Crawler

A crawler which is sometimes referred to a spider, bot or agent is software whose purpose is to perform Web crawling [9]. Crawling Web pages is not a programming task, but an algorithm design and system design challenge because of Internet Web site content is huge.

Distributed crawlers have many challenges need to be solved when crawling at large-scale [8], especially in URL frontier and URL-seen contents. The system has to maintain a high speed access data structure in those components on distributed system.

- Data structure to maintain the set of URLs that have been discovered
- Data structure to maintain the set of URLs that have yet to be downloaded.
- Dispenses URLs according to their priority and politeness. Because if we do not do this feature, the crawler will be become a DDoS attack tools.

The crawler maintains a partition in the URL space in a system. But in our system, the URL space is maintained by redis cluster. It scales easily when the number of URLs increases.

A subscriber has an event checking which can be done by node JS libraries. It checks if a URL task queue has a new job, and then a subscriber will get the job from publisher system.

Depending on the size of target we want to scan and security perspective, a crawler needs to cover all of pages in a site, or in a domain. Furthermore, when we find vulnerability at large scale, we follow prerequisites like the other crawlers used by the other search engine on the Internet.

Using the traditional algorithm alone is not enough for crawling modern Web sites since the client-script execution can change the state of Web application. As a result, we cannot in general rely on URLs to identify states, except for a few that can be reached by a URL. For this reason, the crawler has to identify the states based on the DOMs.

We use hook technique and collect URLs by inject a hook script when a page load which implemented in subscriber components. By this way our system can do like a normal browsers which have Webkit and JavasScript engine.

Crawling is performed by multiple worker threads on subscriber, typically numbering in hundreds. The first step of this loop is to remove an absolute URL from the shared URL queue for downloading on publisher. Each worker repeatedly performs the steps needed to get an URL from URL queue and downloads and finds URL in Web site content.

URL seen test. The purpose of this component is to avoid adding multiple instances of the same URL to the URL queue. It is a stop condition to avoid loop in crawler. The most important thing in URL seen test is that in-memory data structure needs to be accomplished. We store URLs in a dataset on Redis cluster. At this point, we need to use a high-speed hashing or compressing and matching.

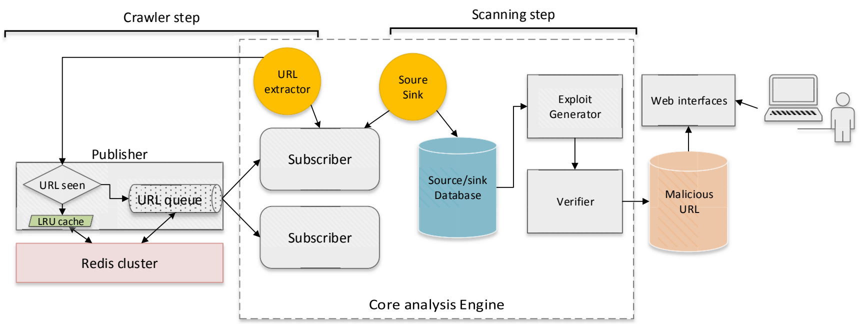


Fig. 1. Overview of the proposed system design.

URL cache. We maintain a LRU (Least Recently Used) cache in publisher node to reduce IO queries to Redis cluster. If the URL can be found in URL cache, it means the URL had been seen before. So we don't need to put that URL into URL queue.

URL queue is a queue used to store URLs to be crawled. Subscriber will get a job from this queue and fetch URL by itself.

URL extractor is one of the most components in our system. It is built on a headless Web browsers on subscriber, implemented hook technique which is described in Section 3. The output is a list of URLs which will be sent to publisher and continue the crawling process.

After the crawling step, we will have a URL dataset of the Internet Web sites including method, headers, forms, Web service, and so on.

1) Hooking: Function hooking is a family of techniques that allow one to intercept function calls to inspect the parameters or alter the behavior of a program. In this section, we present two features of JavaScript that we use to hook functions: *function redefinition* and *set functions*.

Function redefinition is a technique for overwriting JavaScript functions and object methods. Consider the following example, which shows the use of function redefinition that logs any call to the function *alert*. This is achieved first by associating a new name to the function alert, and then by redefining the alert function. The redefinition still behaves as the original alert, however, it adds a call to log its use:

```
alert(”Helloworld!”); //show a pop up window
var orig-alert=alert;
function alert(s){
console.log(”call to alert”+s); //hook
return orig-alert(s);
}
alert(”Hello world!”);// message is also shown in the console
```

While function redefinition can be used to hook arbitrary functions to function calls, it cannot be used when functions are set as an object property. To hook a function in these cases, we used *set function* which are bound to object properties that are called whenever the property is changed. For example, one can hook the function `myHook` to the property `prop` of the object obj as follows:

```
Object.defineProperty(obj,”prop”,set:myHook).
```

We hook addEventListener to capture the registration of a new handler. We need to inject our own hook function whenever the addEventListener function is called. We installed hook function as below:

```
installHook(Element.prototype, ”addEventListener”, myHook)
```

`Element.prototype` is a special object that defines the basic behaviors of all DOM nodes.

```
function installHook (obj, f, hook ) {
var orig = obj[f];
obj[f] = function() {
hook (this, arguments );
return orig.apply (this, arguments );
}
}
```

Hooks an attribute: we can install as a set hook function. We need to check if the JS application overwrite set function by itself.

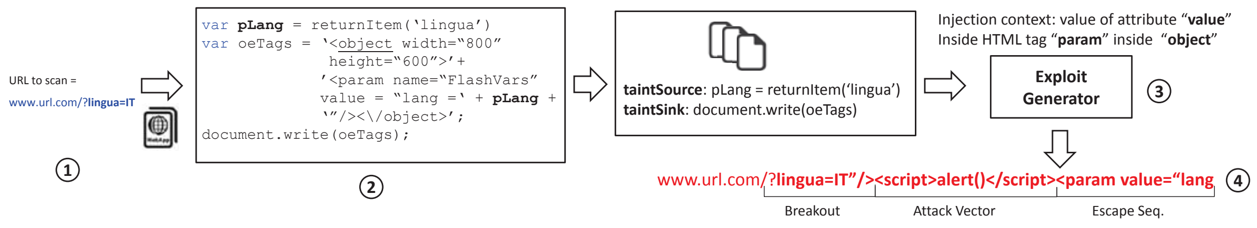
Hooking timing event handlers: installHook (window, ”setInterval”, myHook).

The hook information is propagated using immediately invoked function expressions which are injected when a page onLoadStarted event occur. In order to get a URL from hook function, we define a global variable, for example, window.crawledUrls = [].

In the hook function, we push URL collected into a global variable such as window.crawledUrls.push(url from hook).

B. Scanner

1) Source/sink analyzer: Source/sink analyzer is a module which can dynamically trace and find source sink in JavaScript code. In DOM XSS, source is a entry point for untrusted data, and sink will execute the untrusted data. This component will collect source/sink and save it into database.



```
var pLang = returnItem('lingua')
var oeTags = '<object width="800"
              height="600">'+
             '<param name="FlashVars"
             value = "lang =' + pLang +
             '"/><\/object>';
document.write(oeTags);
```

taintSource: pLang = returnItem('lingua')
taintSink: document.write(oeTags)

URL to scan =
www.url.com/?lingua=IT
1
2
Injection context: value of attribute "value"
Inside HTML tag "param" inside "object"
Exploit Generator
3
www.url.com/?lingua=IT"/><script>alert()</script><param value="lang
4
Breakout
Attack Vector
Escape Seq.

Fig. 2. Payload generator.

The scanner will get source/sink to make a suspicious URL and send it to verifier component which will described in detail in next section. In this section, the scanner will inject a script and wait for page load. The result will be marked with an ID which corresponding to a source/sink object. These information showed stack trace direct to source/sink by V8 stack trace.

In this part, we need to check XHR response which will be handled in Web application (usually by function `innerHTML`)

The following are all cases of DOM XSS to be covered:

- DOM XSS / Javascript injection
- DOM based open redirection
- Second order DOM injection (XHR, WebSocket)
- WebStorage manipulation.
- DOM Clobbering

2) Exploit and verifier: This module will read source/sink information based on ID on database. The ID content will follow DOM flow of source/sink including content, beginURL, taintID, taintSource and taintSink.

The context parser is responsible for finding the context of the injection using the taintSource and taintSink information. It generates a HTML parse tree based on the given input. Using this parse tree and the character-precise taint information, it figures out the exact context in which the injection takes place.

In order for an exploit to work, the tainted string is then replaced with an attack vector. The attack vector to be injected must be adjusted to the context where the attack vector is going to be injected in a Web page.

As shown in Figure 2, exploit payload will be generated by

Exploit = BreakOutSequence + AttackVector + EscapeSequence.

Verifier is a module which sends a suspicious URL to original server and receives response. The goal of this module is to verify the correctness of suspicious URL to archive no false positive in our system.

Verifier is a browser that executes suspicious URL which is generated by exploit generator engine on the normal browser like Chrome or Firefox. After that, if the browser has an alert box, the suspicious URl will become a malicious URL. We can use a malicious URL to attack or fix vulnerability.

VI. Evaluation

A. Crawling Evaluation

First, we use the Web Input Vector Extractor Teaser (WIVET) Web application [10] to assess the capabilities of existing crawlers and compare these to our tool. A test is static if the unique URL is placed in the HTML document without the use of a client-side script. Otherwise, if the client-side program generates, requests, or uses URLs, then the test is dynamic. WIVET features 11 static tests and 45 dynamic tests. We tested the effective of our tool in common Web application and popular Javascript Framework on the Internet. We crawled Alexa top 1000 Web sites in three months.

B. Scanning Evaluation

- Accuracy. We test the accuracy of our scanning against two DOM-based XSS benchmarks namely IBM JavaScript test suite and Google's Firing Range. We confirm that KAKALOT is able to detect all DOM-based XSS vulnerabilities and has zero false negative rates with respect to these benchmarks.
- Distributed system. Message queue task is a kind of distributed system [11]. Our architecture is event architecture.
- Flexibility. By adapting publisher and subscriber model we achive this feature. We can easily integrate with other module in subscriber component.

VII. Related work

The security of Web applications becomes a major concern and it is receiving more and more attentionS from governments, corporations, and research communities. There are many Web vulnerability scanners available on the Internet that are listed by OWASP organization [12]. There are 2 kinds of Web vulnerability scanner: commercial and open sources softwares which have their own contribution on Web application security.

In the context of present work, we will compare with tools were listed by State-aware crawler by prior work [13]. Most of them have lacked crawling of Web sites which were generated by Java-script. To the best our knowledge, we first developed a system based on Google Java Script engine - V8. It can crawl dynamic Web site generated by Java Script effectively.

There are many scanners developed based on HTML parser and AJAX crawling techniques for their crawling steps. OWASP ZAP [14] is a good project which has been developed in a long time by OWASP aiming to scan and detect vulnerability on Web siteS flexibly. It uses Selenium [15] to open browsers and execute AJAX crawling technique, but cannot crawl at large scale or has no distributed mode.

Nikto [16], Wapiti [17], w3af [18] also use HTML parser to extract URL. Arachni [19] also uses AJAX crawl technique based on PhantomJS to make their own crawler. It supportes distributed crawl and tests by using their ruby RPC protocol libraries. Recently, Mesbah et al. have proposed to combine Web application crawling with dynamic program analysis to infer the state changes of the user interface [20]. However, this approach relies on a number of heuristics which do not cover all the interaction points of the client side and there is no Web scanner using this approach. As a result, the largest part of the Web application remains unexplored, which ultimately limits the capability to detect vulnerabilities.

The closest to our approach is PunkSpider [21] which was proposed at DEFCON 21. However, PunkSpider also uses fuzzing to detect vulnerability on the Internet, but it does not support DOM XSS. When it was proposed at the conference, it uses Apache Nutch with depth=3 to crawl and extracts URLs into an indexed partition on Hadoop cluster. The developers of PunkSpider developed a fuzzing engine based on Map Reduce framework from URL sets. At the time we write this paper, they developed their own crawler tool by using a python HTML parser library, BeautifulSoup [22]. Therefore, it cannot crawl the Web sites which are generated by JavaScript.

Crawling a Web site is simple, but crawling at large scale is hard. Techniques used to crawl Web site adequately are also hard because dynamic Web sites are a challenge. Even Google Web crawler which uses AJAX crawling technique can not see content generated by Javascript [5].

VIII. Conclusion

To improve Web site security, we proposed a system which can crawl modern Web application including Java Script event and modern techniques by using hooking technique in both crawling and scanning steps. The evaluation shows that our novel system architechure is flexible, large scale and effective to crawl modern Web applications, particularly suitable for crawling on a large scale.

Acknowledgement

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the Global IT Talent support program (IITP-2016-H0905-15-1004) supervised by the IITP (Institute for Information and Communication Technology Promotion). This research was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2014R1A1A2054174).

References

[1] M. Mikowski and J. Powell, Single Page Web Applications: JavaScript End-to-end, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2013.
[2] CWE, Xss, https://cwe.mitre.org/data/definitions/79.html.
[3] Someone, XSS, https://www.youtube.com/watch?v=Fg4huLUaHjk.
[4] A. Doupé, M. Cova, and G. Vigna, Why johnny cant pentest: An analysis of black-box Web vulnerability scanners, in Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, ser. DIMVA10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 111131. [Online]. Available: http://dl.acm.org/citation.cfm?id=1884848.1884858.
[5] Google, Deprecating our ajax crawling scheme, https://Webmasters.Googleblog.com/2015/10/ deprecating-our-ajax-crawling-scheme.html.
[6] OWASP,"Top 10", https://www.owasp.org/index.php/Category:OWASP_ Top_Ten_Projec
[7] Amit Klein, DOM Based Cross Site Scripting or XSS of the Third Kind, Web application security consortium, 2005, 7.
[8] Olston, Christopher and Najork, Marc, Web crawling, Found. Trends Inf. Retr., vol. 4, no. 3, pp. 175246, Mar. 2010. [Online]. Available: http://dx.doi.org/10.1561/1500000017.
[9] Lawrence, Steve and Giles, C. Lee, Accessibility of information on the Web, Nature, vol. 400, pp. 107109, 1999.
[10] WIVET, Web input vector extractor teaser, https://github.com/ bedirhan/wivet.
[11] Distributed Systems: Principles and Paradigms (2Nd Edition). Upper Saddle River, NJ, USA: Prentice- Hall, Inc., 2006.
[12] OWASP, Vulnerability Scanning Tools, https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools.
[13] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, State of the art: Automated black-box Web application vulnerability testing, in Security and Privacy (SP), 2010 IEEE Symposium on, May 2010, pp. 332345.
[14] OWASP, Owasp zed attack proxy project, https://www.owasp. org/index.php/OWASP Zed Attack Proxy Project.
[15] Selenium, Selenium browser automation, http://seleniumhq. org/.
[16] Nikto, Nikto Web scanner, http://wapiti.sourceforge.net/.
[17] Wapiti, Wapiti Web scanner, https://github.com/sullo/nikto.
[18] A. Riancho, w3af: Web application attack and audit framework, http://w3af.org/.
[19] A. Riancho, Arachni Web scanner, http://www.arachni-scanner.com/.
[20] A. Mesbah, A. van Deursen, and S. Lenselink, Crawling ajax-based Web applications through dynamic analysis of user interface state changes, ACM Trans. Web, vol. 6, no. 1, pp. 3:13:30, Mar. 2012. [Online]. Available: http: //doi.acm.org/10.1145/2109205.2109208.
[21] punkspider, Massive attacks with distributed computing, https: //www.punkspider.org/.
[22] P. Spider, Punk spider sources - bit bucket, https://bitbucket. org/punkspider/punkscan.