Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Webapp vulnerability detection through semi-automated black-box scanning

Author:
Abraão Pacheco Dos Santos Peres Mota

Supervisor:
Dr. Sergio Maffeis

Second Marker:
Dr. ??

January 21, 2018

Contents

Chapter 1

Introduction

1.1 Motivation

In recent years, use of internet applications has skyrocketed across the world. This is exarcebated by the ubiquity and sheer number of devices that are now connected to the internet. The users of these devices place a great deal of trust in the applications and websites they use to power the activities they engage in. These play an ever increasingly influential role in people's lives - as an example, in a not so distant past, people would have to travel to a physical bank branch to deal with account matters or execute transactions. Though physical presence can still be required nowadays, a majority of the population will now take advantage of online banking in their day to day lives, often even from the comfort of the phone in their pocket. This has obvious time and efficiency advantages, and benefit many who use this today. This was not an overnight phenomenon however; online banking initially faced heavy customer reluctance, enough to warrant studies on the cause of this [10]. This is understandable, given that everyone holds their financial situation as a very sensitive part of their lives. Banking is not a solitary example however; with the pervasive use of the internet, its users have gradually become less apprehensive about surrendering important details over a web connection, such as their phone numbers, addresses or medical history. This sensitive information is expected to be kept private when being given away to a web service - it is a conventional unspoken agreement between the user and the service provider that this is the case, although there are laws to enforce this as well. All of this builds up to a massive responsibility placed on the shoulders of web application developers today; their products are expected to uphold a high standard of security, which oftentimes is hard to reach and maintain.

Despite this, it is all too common to hear about web applications that suffer from severe cyber attacks. In some of the most debilitating hacks we have heard of in recent years, it is often the case that they were a result of simple vulnerability mitigation measures not being taken. A vulnerability in this context can be illustrated as an unlocked window into a house - it may not be immediately clear that the house owner has overlooked this security aspect, but if a burglar manages to work out that this is the case, they can maliciously *exploit* this vulnerability in the house, and use that as an entry point to steal all the valuable contents within. Properly closing and locking all the windows in the house would be an obvious mitigation to this, but this is only one of the potential (ingenious) ways for a burglar to make his way into the house. The same principle can be applied to websites; it is important to cover as many bases as possible to prevent a potential information or content leak to malicious users. Some vulnerabilities may be harder to detect than others, and it is unlikely that *all* the possible vulnerabilities will be covered, but any efforts towards can only work in favour of the website developer.

Sadly, due to the immaturity of the web development industry and how quickly technologies emerge in the field, web security is an often overlooked aspect of development. The lack of security as a fundamental tenet for development is also due to a gap in developer education, and a high entry barrier to understand and mitigate potential security vulnerabilities. Though this view is slowly changing, web security is not treated as an important focus for new web developers to understand as part of many online and university courses, so many will get by, even work in professional development roles without having ascertained basic security principles. Common vulnerability

mitigation measures are also hidden in the inner workings of many frameworks developers use and become accustomed to. For example, using *Anti CSRF* (Cross Site Request Forgery) tokens in web forms to prevent action hijacking has become commonplace in web frameworks, and is a feature that is often activated by default [6, 7, 13]. It is very often the case that features like this will be used without knowledge of what they do and how they work (in my own experience, I had been venturing in web development for years before realising that feature existed). However, in the case that the developer changes to use another framework without *secure by default* features, or decides to write an application from scratch, the burden of creating a secure application lies with them even further. These default features become like 'training wheels' for some uneducated developers and without them, these creators will be left without a clue as to how to mitigate vulnerabilities, let alone know that these exist altogether. Experienced web developers are less likely to leave the "windows" of their website unlocked, but it nevertheless makes sense to install security alarms to prevent both obvious and more subtle security risks. If these security systems can automatically work in the background it is ideal, but like a home security alarm, it is no good if no-one intervenes upon detection of a problem. This begs the question; what is the feasibility of creating a tool that can work as an aide to a web developer in detecting and preventing vulnerabilities in a website?

1.2 Objectives

The question raised above effectively highlights the overarching problem this project aims to tackle - improving security for web applications. The goal is to do this through a pragmatic approach; the final objective of this project is to construct a tool which can diagnose vulnerabilities on a target website as a user browses the service. The tool then uses the information gathered to suggest potential exploits that can arise from these vulnerabilities.

My initial proposal in solving this encompasses writing a browser extension to analyse the target website, and applying a wide variety of scans and techniques to detect potential security pitfalls the website may have. A browser extension is an appropriate approach to this as it is a lightweight application running with elevated privileges on the user's browser, giving it the appropriate clearance level to run a variety of automated scans on the user's behalf.

A tool of this kind has 2 immediately clear use cases. The first would be to give this tool to a website owner or developer and create it in such a way so that it gives clear and concise suggestions to quantifiably improve the security level of the target website - for example, if an SQL injection has been detected, show the user where on the website this vulnerability can be found, and make effective suggestions as to how to mitigate this (in an SQLi, it may be done by sanitizing user input). The tool could analyse a range of potential vulnerabilities and generate a quantifiable rating for the website in order to give feedback to the developer on where the website needs improving or immediate work. The slightly alternative use case of this tool provides a more in-depth scan per potential vulnerability, and would be better suited for use by a penetration tester, or an otherwise knowledgeable web security expert. In this use case, the tool would work in a similar fashion, albeit with a different final goal of going 'all the way' by helping the user to detect vulnerabilities and creating exploits using them on the target application.

In both potential use cases, the benefits of the tool are twofold - it can be used as an educational stepping stone for developers to further their understanding of security in web applications. It also provides a pragmatic way to improve website security through slightly different ways. In the first case, the developer applies the given suggestions to their website, making an immediate effect on its security. Alternatively, the penetration tester can show developers the dangers of ignoring security for their website by exploiting open vulnerabilities, which gives further incentive to fix these as soon as possible. A penetration tester with the knowledge of exploiting vulnerabilities will most often also know how to mitigate vulnerabilities against their own attacks.

This project aims to explore the latter of the two approaches mentioned. This is the more encompassing of the two possible use cases, and produces richer information beyond safety improvement recommendations. The extension will be designed to run in real time as the penetration tester navigates the website being targeted, analysing server responses from the website given to

the tester based on different inputs. It will also perform automated fuzzing of detected user input forms and other parameters in an attempt to detect vulnerabilities that can be triggered from tampering or manipulating these. It is not reasonable for the penetration tester to find a combination of inputs that will immediately result in the unveiling of a vulnerability; this is a slow manual process that requires careful crafting of inputs that adapt to received server output. It makes sense to take advantage of computing power in this case to expedite this process by attempting several combinations and permutations of inputs that are well known to be problematic to handle by insecure systems, showing the existence of vulnerabilities. The creation of the tool will have quality of vulnerability detection as a more important design principle over breadth of detection. This results in a more accurate tool that achieves better results, with fewer false-positives.

Chapter 2

Background

2.1 Website Vulnerabilities

This project is rooted in identifying and mitigating web application vulnerabilities. In order to do so, it is essential to understand what these are, their impact, and what steps are necessary to prevent them. Fortunately, there are great wealths of information available to learn more about vulnerabilities. A great starting point is *OWASP*, the Open Web Application Security Project [14]. This is a community driven effort into improving the safety of software across the world, and the organisation has taken extensive steps into creating useful guides for developers wishing to know more. A particularly convenient resource they provide is a consensus of the top 10 security risks that web applications face today.

At the time of writing, this list contains the following risks, some of which are appropriate to pursue in this project:

Injection - This risk arises from any place on a website that accepts client controlled input. This may be through the more obvious - submission of web forms and search boxes, but also includes more subtle ways of the user providing input, such as URL or HTTP request parameters. Accepting this input per se is not a vulnerability, but the issue lies in blindly trusting this input to not be malicious. Whenever this input is used to query a database or perform server-side commands, if it has not been sanitized (cleaned to delete potentially dangerous input combinations), it has potential to leak or permanently corrupt information for the website owner. Due to its prevalence and modus operandi, this is an appropriate vulnerability to scan for and detect in this project.

Broken Authentication - This can encompass a wide range of things, such as use of weak or default passwords and admin accounts, or poor management of session identifiers such that these can be easily manipulated. It can also include flawed password recovery mechanisms. These risks could be analysed as part of the security analysis ran by the tool, and with support from user input could be combined with scans to effectively detect weak authentication mechanisms.

Sensitive Data Exposure - This risk is a result of using weak or poorly protected cryptographic measures. If a website has left their encryption keys in plaintext for someone to be able to find, or doesn't use HTTPS altogether, it may be exposed to this attack vector. The tool in question could look for a lack of TLS enforcement across pages, or attempt to force a connection downgrade (from HTTPS to HTTP) to guide the user in the right direction of finding sensitive information.

XML External Entities - This vulnerability exists in applications that accept or include XML data from a 3rd party. A malicious user could use this data format to attempt to exfiltrate sensitive data from the handling server. Identifying this risk without user input may prove to be difficult, but could be within the potential vulnerabilities considered by the tool.

Broken Access Control - This risk is comprised of all the possible ways in which an application might allow a user to perform actions that should be restricted to their access level (for example, allowing a non-admin user to read bank balances of arbitrary accounts in the system). Determining

what is and isn't an allowed action on a website varies tremendously per application, so this is a very difficult task to automate, and isn't ideal to try and incorporate as part of the final tool.

Security Misconfiguration - A poorly setup server may suffer from this risk if there are components or services installed by default that are not prepared accordingly to the necessary security measures - such as disabling error stack traces from services; these may reveal more than what a website owner thinks when in the context of being attacked. This vulnerability type is well suited for automatic scans that scour the website for versioning details of services being ran, which may in turn reveal known weaknesses to look for in the case of a negligent set up.

Cross-Site Scripting (XSS) - One of the most well known risks for web developers today is XSS - this is exploited when a user successfully injects Javascript into a website, causing a non-intended script to run. This is a severe risk depending on how many users it might affect, it can range from Self XSS which affects only the user injecting this content, but can also be seen as Stored XSS, whereby a malicious script is stored in a database, and can be potentially retrieved and ran by other users, posing serious risks where credentials and other session information can be stolen. This lends itself well to the purpose of the application to be developed in this project, as it can test a wide variety of known inputs to expose this vulnerability.

Insecure Deserialisation - Serialisation is the process of converting a data structure into a format that can be easily transferred over a connection. This resulting new format must then be deserialised to obtain the initial information back. An attacker could craft a serialized object such that it exploits the process or properties of deserialisation in the target application to obtain access to privileged data. This process will vary depending on the application domain and intended data structures, so it is not ideal for automated tools to attempt to tackle this issue.

Using components with known vulnerabilities - In application architectures that heavily rely on a variety of components or libraries from different sources, it can often be hard to ensure that these are all kept up to date. In instances where they are not, it becomes a simple task for a scanner to produce a map of outdated version numbers to possible exploits that have been discovered on the component since then. Once a CVE (Common vulnerabilities and Exposures - a unique identification method for security vulnerabilities found worldwide) is produced as a result from a scan, it is just a matter of reproducing exploit steps found online to endanger the application.

Insufficient Logging and Monitoring - An ideal web application keeps a track of all activities and accesses that occur. This helps provide accountability for actions. When this is not the case, it weakens the position of the website administrators to pinpoint attack culprits. This is very hard to detect from an outsider perspective, and *insufficient* is an objective term; it wouldn't be fruitful to include this in an externally ran vulnerability scanner.

This is not an exhaustive list, but is useful to consider when attempting to address specific aspects for development in the tool. There are several other features or protocols that can be checked to better ascertain the security of a website, such as:

- Use of iFrames
- Cookies
- Appropriate headers
- CSRF mitigations
- Use of HTTPS
- HSTS
- JS instrumentation
- Remote inclusion and dependency analysis
- DoS
- Clickjacking
- CSP and sandboxing policies

2.2 Vulnerability Scanners

This project is not the first piece of work aimed at detecting website vulnerabilities. There have been efforts in academia to address this issue ([1, 5, 9, 3], to name a few), where tools are often

built as part of the related paper to demonstrate the feasibility of the proposed techniques in detecting vulnerabilities. There are also other commercial and open source tools designed to solve this problem such as *Acunetix* [11], IBM's *AppScan* [8] and *w3af* [16]. In order to appreciate the aspects in which this project aims to innovate in, it is fruitful to understand some aspects used to refer to these scanners.

2.2.1 Black Box

Much of the reading in this area makes reference to *black box* scanners. This term is simply used for the tools that analyse a website without access to its source code. The opposite of this would be a *white box* scanner - this sort of program works through static analysis on the raw code of the web application.

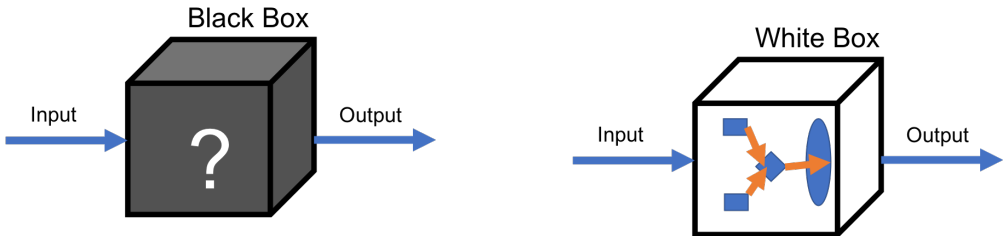

Figure 2.1: A black box scanner derives behaviour strictly from I/O, whilst a white box scanner has all the inner workings of a (web) application at its disposal for analysis

A white box scanner takes into consideration the implementation of the service, so it may be designed with the specific framework or language used for the application in mind. Conversely, a black box implementation does not care about these details and looks only at outputs from the service. A major differentiation between the two approaches is their use case; a white box scanner emulates the experience of a web developer who is looking to detect potentially exposed parts of his code, whereas a black box scanner portrays the setting for a malicious attacker, where they are only working with the exterior interface of the application to forge an intrusion. The black box approach is the one I will be using in the extension implementation as it aligns best with the project requisites.

2.2.2 Related work

A large majority of web vulnerability scanners are designed to be ran passively. This means an auditor of the security of a web application would start the tool, and depending on the length and intensity of the scan, either grab a coffee while it executes or leave it running overnight. Either way, these are expected to produce reports on where a website is suffering from a vulnerability, which can then be interpreted to produce a fix. This automation is a great bonus because it makes life easier for the auditor, as they can get on with other tasks in the meantime; depending on the initial monetary and time investment needed to set up the scan, doing so automatically can be a cost-effective solution for many businesses.

There have been studies committed to evaluate the effectiveness of these black box scanners. Doupé, Cova and Vigna did an extensive analysis of the tools available in 2010 [4], including tools ranging from free to $30,000+ worth. In the same year, Bau, Bursztein, Gupta and Mitchell published their own paper with a similar analysis [2]. More recently, in 2015, Parvez, Zavarsky and Khoury released a study on the effectiveness of scanners in detecting a limited set of vulnerabilities [15].

Doupé et al. found that *crawling* was one of the major limiting factors of web vulnerability scanners; according to them, "...crawling is arguably the most important part of a web application vulnerability scanner; if the scanner's attack engine is poor, it might miss a vulnerability, but if its crawling engine is poor and cannot reach the vulnerability, then it will surely miss the vulnerability". To better understand this, it is important to contextualise what *crawling* is.

A typical vulnerability scanner will loosely consist of 3 different modules:

- *Crawling module* - This is the first part of the scanner that gets executed. A crawler will recursively follow every possible link in a webpage so that the tool can build up an internal representation / database of what the target website looks like. As mentioned above, this stage will make or break the scan - though it is unlikely that a crawler will be able to find 100% of the available subpages on the target website, any missed links will result in the scanner not considering those pages, which in the worst case scenario may miss a page which is the root of many vulnerabilities on the target site. Acunetix themselves, the providers of the commercial vulnerability scanner, say "if you can't crawl it, you can't scan it!" [12].
- *Attacker module* - At this stage, the scanner attempts to chip away at any potential cracks in the website. It goes through every stored entry in the previous phase and scans the content of the associated page. Depending on this content (e.g. a form or a URL parameter), the scanner issues web requests with specially crafted input designed to create *interesting* responses from the web server.
- *Analysis module* - This module reads the outputs from the target to the attacker module input, and scans for any significant or telling change from the server side response. If the web server generates a wildly different response to a normal input (e.g. it issues a page containing an SQL error message), then this is flagged up as a potential vulnerability. This module can have a feedback loop to the attacker module to refine attack methods. This is done through *fuzzing* - creating mutations of the benign input and testing these on the application until a more malicious input is generated that triggers a vulnerability. The scan completes after all potential mutations and attack vectors have been exhausted, or a resource cap is met (time elapsed, bytes sent etc).

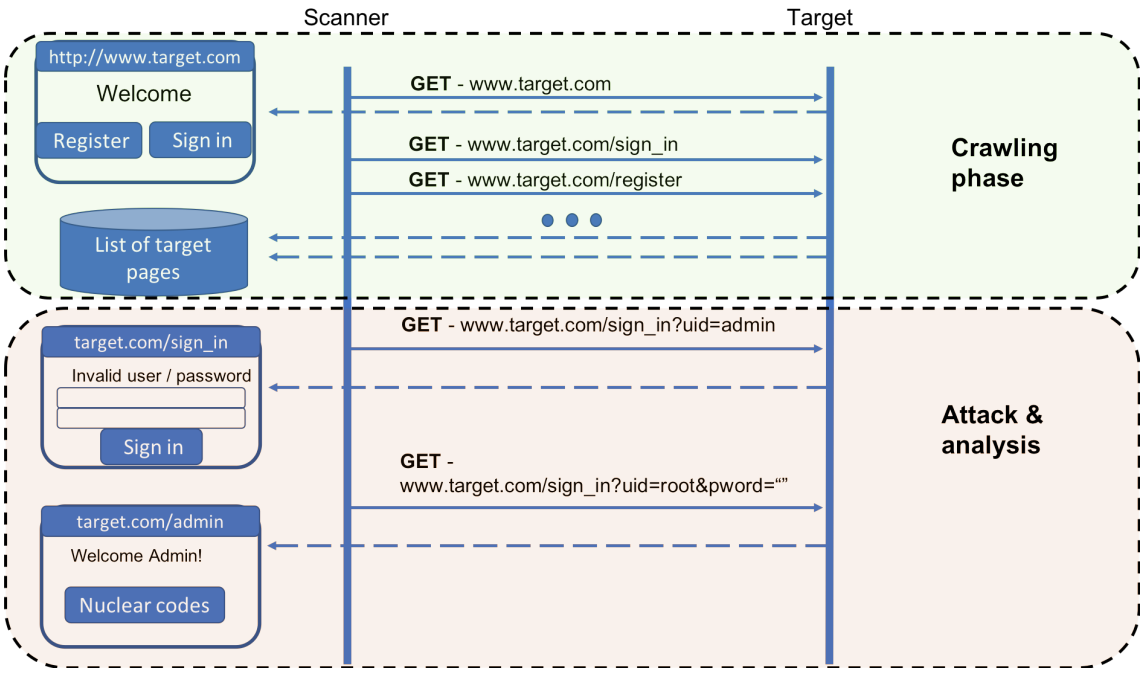

Figure 2.2: The typical structure of a vulnerability scanner. The crawling phase builds up a database of potential pages to attack. During attack, malicious inputs are fired towards pages to try and trigger undesired behaviour - the analyser reads response contents with some heuristics to determine what responses seem to indicate vulnerabilities.

With this knowledge in mind, it is now easier to appreciate how important the crawling phase is, as it sets an upper bound to how far a scanner can go. Crawling in itself is however not a trivial task to implement. A naive approach to this would be to start at the target page given, scan for any `<a>` anchor links in this page and filter this list to only include links that belong to the target domain (e.g. if our target was `facebook.com`, we might be interested in `facebook.com/account`,

but we would discard any links to `google.com`, as Google links belong to a different domain. As a general rule of thumb, the links we explore have the target link as a prefix to the URL). This method of crawling pages would quickly come up against issues though. A lot of the interesting state of a web application is often hidden behind a login form, so if a valid user account isn't created then a scanner will not be able to explore the full array of actions available. The method described above would not consider this, so would skip out on all the intriguing parts of a website that requires you to have an account. There are also other technologies used in the web that make this slightly harder, such as Flash objects that contain useful sublinks. These objects are not straightforward to analyse, and make automated crawling difficult.

This combination of components quickly complicates things for creation of an effective crawler implementation. A competent crawler needs to go above and beyond being just a database of pages - it has to emulate human interactions with an application without any prior domain knowledge. To do so, it has to somehow derive an internal representation of what the website looks like, and refer to this state when crawling and in the analysis phase. Keeping state was an issue raised by Doupé et al. in [4]. As an example of its importance, if during the attacking phase the scanner is logged in, and an attack request causes the scanner to log out of the application, all the subsequent requests will execute in a different context where the scanner is logged out, invalidating their intended purpose. Doupé, Cavedon, Kruegel and Vigna later addressed this by creating a scanner that generated an internal state machine representation of a web application based on heuristically unique server responses [3]. The techniques demonstrated in that paper were useful in creating a more effective crawler that remembered state so as to not waste computation efforts. The resulting scanner achieved higher code coverage rates for web applications analysed over other otherwise similar, open-source scanners. Higher code coverage rates mean that a scanner is exercising more of the application's source code, thus increasing likelihoods of finding a vulnerability.

It is clear that an automated crawler based web vulnerability scanner is a difficult tool to create. The shotgun approach used by the crawler can also have negative consequences for the application as discussed in 2.4.4. Though state of the art crawlers can now handle the aforementioned hurdles and other intricacies of different web applications, the creation of an efficient, fully automated crawler could warrant a project of its own.

These points lead us nicely onto the intended innovation behind the project idea -

why should you use human input: organic, realistic experience make use of the entire webpage - no need to skip AJAX, JS, flash etc - although these might be out of scope of the project

using the website as intended allows us to keep a simpler representation of the website if the scanner works at the same level as the user then they

whyJohnny found that a bunch of students with some knowledge had better results than scanners

Human judgement is needed anyway - scanner won't know for sure. It's a human process to hack, and people will spend hours in crafting specialised attacks.

"C) Adding interactive multistep options to scan. Out of the three scanners, only one scanner had that feature. While the option to set login credentials helps to provide application's login info for the scanning, it does not instruct the scanners how to visit the pages and where to put attack vectors to exploit and to detect the vulnerabilities." - Analysis of effectivness of black box web app scanners

"Our experimental findings confirmed that choosing the right attack vectors for the detection and exploitation of stored XSS and stored SQLI remains a big challenge for black box scanners." - Analysis of effectiveness of black box web application scanners in detection of stored sql injection and stored xss vulnerabilities (2015)

"Scanners need to be improved to use right attack vectors in the right situations" - Analysis of effectiveness of black box web application scanners in detection of stored sql injection and stored xss vulnerabilities (2015)

2.3 Browser extensions

2.4 Limitations

crawling does find more stuff to look through

2.4.1 Human review

2.4.2 Breadth of work

2.4.3 Self security

My own extension is not free from attacks / bugs http://slideplayer.com/slide/4352044/

2.4.4 Ethics & Handling of Results

each potential vulnerability may have 100's of different known alternatives to fuzz through, scans might take the website down if intensive

Chapter 3

Evaluation

Project evaluation is very important, so it's important to think now about how you plan to measure success. For example, what functionality do you need to demonstrate? What experiments to you need to undertake and what outcome(s) would constitute success? What benchmarks should you use? How has your project extended the state of the art? How do you measure qualitative aspects, such as ease of use? These are the sort of questions that your project evaluation should address; this section should outline your plan.

3.1 Intended functionality

3.2 Metrics of success

3.2.1 Correct reporting

3.2.2 False positives

3.2.3 Code coverage

3.2.4 Usability

3.3 Experiments

3.3.1 Benchmarks

3.3.2 Ease of use

3.3.3 Educational purpose

3.4 Intellectual contributions

Chapter 4

Project Plan

You should explain what needs to be done in order to complete the project and roughly what you expect the timetable to be. Don't forget to include the project write-up (the final report), as this is a major part of the exercise. It's important to identify key milestones and also fall-back positions, in case you run out of time. You should also identify what extensions could be added if time permits. The plan should be complete and should include those parts that you have already addressed (make it clear how far you have progressed at the time of writing). This material will not appear in the final report.

Bibliography

[1] Cova M. Felmetsger V. Jovanovic N. Kirda E. Kruegel C. Balzarotti, D. and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. *2008 IEEE Symposium on Security and Privacy*, 2008.

[2] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345, May 2010.

[3] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 523–538, Bellevue, WA, 2012. USENIX.

[4] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In Christian Kreibich and Marko Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] Cavedon L. Kruegel C. Felmetsger, V. and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. *Proceedings of the USENIX Security Sym posium (Washington, DC, August 2010)*, 2010.

[6] Django Software Foundation. Django project documentation - cross site request forgery protection. https://docs.djangoproject.com/en/2.0/ref/csrf/. Accessed: 10/01/2018.

[7] Rails Guides. Rails guides - csrf countermeasures. http://guides.rubyonrails.org/security.html#csrf-countermeasures. Accessed: 10/01/2018.

[8] IBM. Ibm security appscan. https://www.ibm.com/security/application-security/appscan. Accessed 20/01/2018.

[9] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: A web vulnerability scanner. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 247–256, New York, NY, USA, 2006. ACM.

[10] Tuire Kuisma, Tommi Laukkanen, and Mika Hiltunen. Mapping the reasons for resistance to internet banking: A means-end approach. *International Journal of Information Management*, 27(2):75 – 85, 2007.

[11] Acunetix Ltd. Acunetix web vulnerability scanner. https://www.acunetix.com/. Accessed 20/01/2018.

[12] Acunetix Ltd. Acunetix web vulnerability scanner - crawling. https://www.acunetix.com/vulnerability-scanner/javascript-html5-security/. Accessed 20/01/2018.

[13] Taylor Otwell. Laravel docs - csrf protection. https://laravel.com/docs/5.5/csrf. Accessed: 10/01/2018.

[14] OWASP. Open web application security project. https://www.owasp.org/index.php/Main_Page. Accessed 20/01/2018.

[15] M. Parvez, P. Zavarsky, and N. Khoury. Analysis of effectiveness of black-box web application scanners in detection of stored sql injection and stored xss vulnerabilities. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 186–191, Dec 2015.

[16] w3af Org. w3af - web application attack and audit framework. http://w3af.org/. Accessed 20/01/2018.