Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Project Title

Author:
Abraão Pacheco Dos Santos Peres Mota

Supervisor:
Dr. Sergio Maffeis

Second Marker:
Dr. Ben Livshits

April 28, 2018

Abstract

Your abstract goes here

Acknowledgements

Thanks mum!

Contents

List of Figures

List of Tables

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 [**?**]. 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 mitigating any vulnerabilities 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 [**?**, **?**, **?**]. 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.

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 using these to forge exploits 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, albeit through different routes. 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 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 known to cause problems in insecure systems, unveiling the existence of vulnerabilities.

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 [**?**]. 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.

In their latest 2017 report, 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. There are a number of ways a website may process user input, such as the submission of web forms and search boxes, as well as using client provided 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 a server uses this input in querying a database or performing server-side commands, if the input has not been *sanitized* (altered to delete potentially dangerous input combinations), there exists the risk of a leak or permanent corruption of information stored by the web application. Due to its prevalence and modus operandi, this is an appropriate vulnerability to scan for and detect in this project.

Broken Authentication - An application must authenticate its users so that it can confirm their identity over the internet. This is most often provided through a login form. Broken or poor authentication happens when the recommended practices to set this up are not followed. This can encompass a wide range of things, such as allowing the use of weak or default passwords, not changing default administrator account details, 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 cryptographic measures, or entirely foregoing their use. 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 proposed tool could look for a lack of TLS enforcement across pages, and attempt a *connection downgrade attack* (recognising the application does not force the use of HTTPS, and so attempting to make all requests unencrypted by using HTTP instead). The user could make use of this information to craft an attack in attempts of finding sensitive information.

XML External Entities - XML (eXtensible Markup Language) is a widely used markup language to encode dcouments. The described 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. Exploiting this risk without knowledgeable 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, a bank 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, and so provides little markers for a semi-automated tool to find, 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. An example may be disabling error stack traces from services - these may reveal information about the language being used on a database, or the version of the web server being used, which can be leveraged by an attacker in producing an exploit. 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 information into a website, causing a non-intended script to run. This is a severe risk depending on how many users it might affect. XSS may range from Self-XSS which affects only the user injecting this content, but may also be seen in the Stored-XSS variant, whereby a malicious script is stored in a database, and can be retrieved and ran by other users. This poses 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 and understood by different programming interfaces. 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 (assuming these version numbers are disclosed somewhere). Once a CVE (Common vulnerabilities and Exposures - a unique identification method for security vulnerabilities found worldwide) is published online as a result from an exploit, exploiting the web application is trivial. A scanner simply has to match the version number to a found CVE, and can reproduce 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.

In order to stay relevant, it is good not only to look at recent reports, such as the above from 2017, but also reports of different provenance. *HackerOne* is a company that produces surveys harnessing the power of ethical hackers around the world to gather statistics on the current general state of web application security. In their 2017 report [**?**], the survey gathers information on several vulnerability types from 2013 to 2017, and their prevalence across different industries. The most

prevalent types of vulnerabilities included **Cross Site Scripting, improper authentication, Cross Site Request Forgery (CSRF), violation of secure design principles, information disclosure** and **privilege escalation**. This should however be analysed with caution; HackerOne is a profitable company, surveying ethical hackers for hire, some of which, by self admittance, do so simply for profit or to make a living. This may therefore skew the results - it is fair to assume that some percentage of these hackers may aim for the highest ratio of bug bounty award to search / hacking efforts, so we may see more vulnerabilities discovered that are simply easier to uncover, though this may not reflect the true state of the industry. In their latest 2018 report [**?**], it is clear that there are 'favourite' attack vectors by the surveyed attackers, namely, XSS and SQL injection.

The aforementioned lists are not exhaustive, but are useful to consider when attempting to classify and sort different types of vulnerabilities during the development phase of this project. 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
- Use of HTTPS
- HSTS
- Javascript instrumentation
- Remote inclusion and dependency analysis
- Clickjacking
- CSP and sandboxing policies

2.2 Related Work

This project is not the first piece of work aimed at detecting website vulnerabilities. There have been efforts in academia to address this issue ([**?**, **?**, **?**, **?**], 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* [**?**], IBM's *AppScan* [**?**] and *w3af* [**?**]. 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.

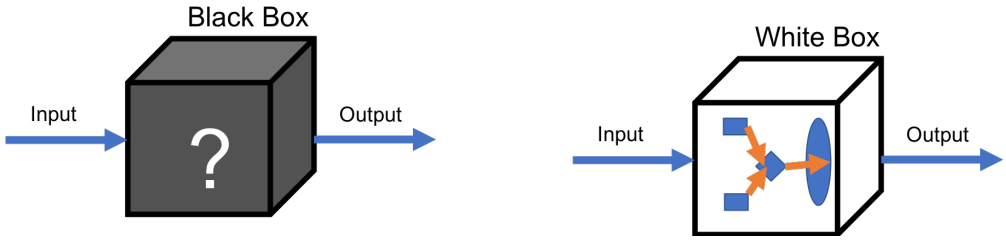

Figure 2.1: A black box scanner derives behaviour strictly from I/O, whilst a white box scanner has all the inner workings of a (web) application at its disposal for analysis

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 Automated Vulnerability Scanners

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 [**?**], 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 [**?**]. More recently, in 2015, Parvez, Zavarsky and Khoury released a study on the effectiveness of scanners in detecting a limited set of vulnerabilities [**?**].

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!" [**?**].
- *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).

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`,

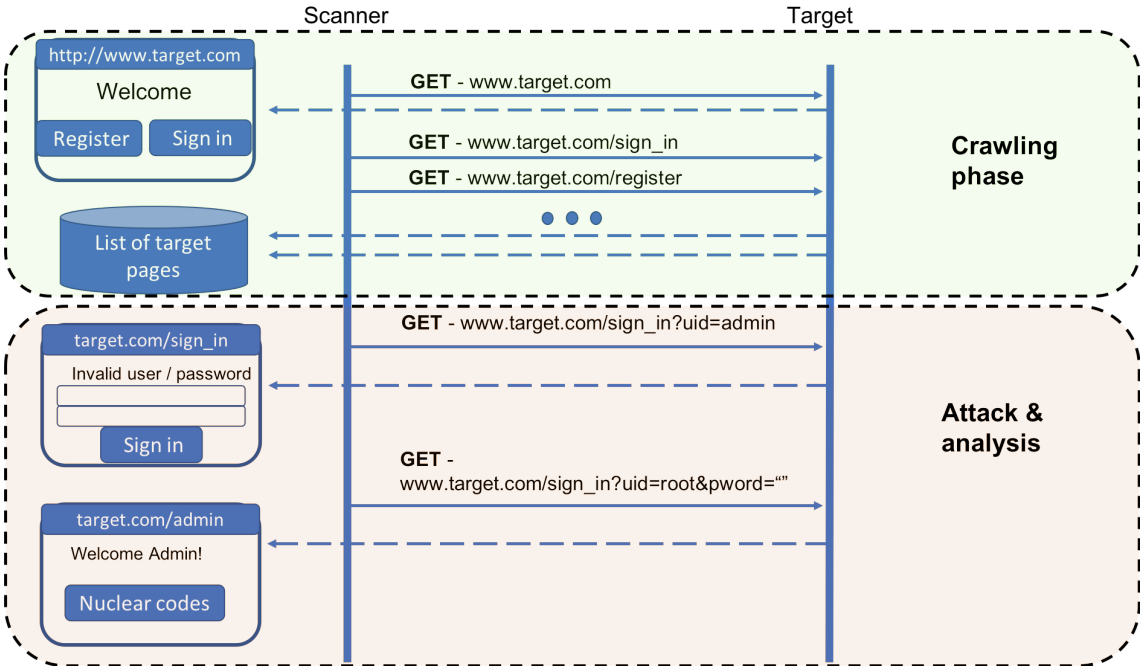

Figure 2.2: The typical structure of a vulnerability scanner. The crawling phase builds up a database of potential pages to attack. During attack, malicious inputs are fired towards pages to try and trigger undesired behaviour - the analyser reads response contents with some heuristics to determine what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

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 [**?**]. 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 [**?**]. 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 2.5.4. 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.

2.3 Browser extensions

I propose to create the tool described as a browser extension in Google Chrome. An extension seems to be ideal as a method to accomplish the intended functionality for 2 main reasons.

- *Extension privileges* - A browser extension works as a self-contained application that is trusted enough to expand the browser experience. Chrome predefines a potential permission set an extension can tap into. Each extension must declare which of these it will be using in a `manifest.json` file, as well as list of which domains it expects to be ran on. Upon installing the extension, the user is asked whether they wish to allow this behaviour. As required by the project, an extension can contact remote servers (allowing it to replay user actions on their behalf), as long as the appropriate cross origin permissions are appropriately set up. If an extension is compromised, the application of the least privilege principle can mitigate potential damages because it will only be able to possibly affect the declared domains at the previously granted permission level. It is therefore good practice to declare the minimum amount of domains and permissions required to get the extension to work.

  An extension has a `background.html`, a page which runs invisibly as the user browses. This is where most of the logic and state of the extension is stored, but it cannot directly interact with target pages. For this, there are *content scripts*; these are small Javascript programs that are injected into the selected domains and can interact with the Document Object Model (DOM) of a website and modify it according to the programmed needs. It is important for extension behaviour to not break functionality of existing sites. For example, if a web application is using JQuery 1.0 and the extension is using JQuery 2.0, it would be unacceptable for the newer dependency on the library to break the existing functionality, so content scripts are isolated from the existing page scripts. This is done through *isolated worlds*, which create a sandboxed copy of the website DOM to avoid clashing between content and scripts and existing scripts. Because the content script can interact only with the website, it may be desirable to establish communication with the extension core to send information back to the `background.html` to process. For this reason, there exists a message listening API between the `background.html` and the content script to allow communication.

  It is important to keep these separate for security reasons. If a content script is compromised by a malicious page targeting the extension, the attacker still has to exploit another vulnerability in the background page to exfiltrate any data. Additionally, the entire extension itself runs in a separate process to the browser and other extensions. This means if an extension is compromised and begins to degrade the system, this process can simply be halted without killing the browser or other innocent extensions.

  Additionally, extensions run on HTML5, CSS3 and support Javascript running on Chrome's own engine, with native JSON capabilities. They support the latest web technologies and as such are not heavily restricted on the use of these, as other methods used in black box scanners are. Therefore, browser extensions run at the appropriate level of abstraction to perform the necessary technical operations without much hinderance.

- *User operability* - Because of the features described above, the extension can be designed in such a way that allows it to sit between the user and the web application and seamlessly create the desired semi-automated experience. Wherever possible, depending on the type of vulnerability, it would be ideal to give the user interactive visual cues for when the scan has detected something that merits further research. If the vulnerability is attached to a specific form that takes user input for example, the extension should have the capabilities of identifying the DOM object for this, and add a small tooltip around the visual area of this object to initiate the deeper scan analysis. If the vulnerability is not necessarily caused by visible means, then the extension is also able to create other appropriate visual cues, such as a fixed position element. Being able to implement these characteristics should result in a more intuitive user experience.

  This also means that the user will not be doing any heavy lifting - the extension will be passively searching for vulnerabilities as they use the website, but should also be able to

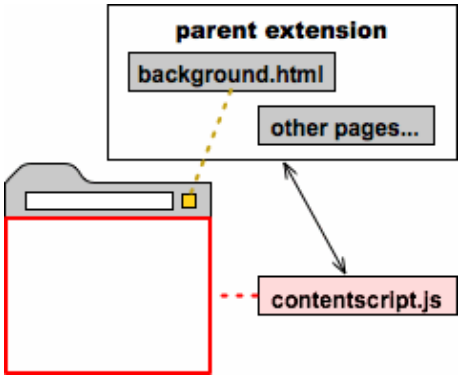

Figure 2.3: An extension is split into 2 main areas. The `background.html` page has access to all extension privileges and is where the business logic of the extension is stored. It cannot interact with user webpages. Content scripts can be injected to interact with the DOM of a page. The `background.html` can contact the content script via a message passing API. Image courtesy of Google Developer documentation. [?]

be activated at user will. During the phase where the tool begins to replicate user actions to investigate a vulnerability (explained further in 2.4), if implemented appropriately, the extension will be able to repeat the human steps of submission and navigation at a much faster rate than a human would do, with several more inputs.

2.4 Project Contribution

These points lead us nicely onto the intended innovation behind the project idea - to create a semi-automated web application vulnerability scanner. The essential difference between this and the aforementioned black box scanners is that it intends to work as an aide alongside a pentester, not as a fully automated background process.

The proposition of creating a tool that is driven by user input seems appealing for a number of reasons:

- **Full Experience** - A scanner that analyses a web application for vulnerabilities as a user is using the application has fewer limitations as to what is within its scope. As pointed out by [?], scanners struggle to scour through more complex constructs of the internet such as convoluted Javascript forms, AJAX requests and Flash objects. As a result, many scanners ignore these features altogether, dismissing potential vulnerabilities in the process. A user driven experience allows the user to interact as they would normally with these technologies, and can analyse inputs and outputs accordingly.

- **Educational** - As mentioned above, one of the issues that leads to the need of vulnerability scanners is the lack of security-aware developers. By developing a tool that works alongside developers, they can refine their skills in this area if they already know some security basics. There is also room for learning for website owners as penetration testers may demonstrate the severity of exploits caused by vulnerabilities using the tool, and mitigating these.

- **Simpler crawling module** - Since the tool is not scouring the entire site at once but is rather following the more natural workflow of a human user, the equivalent crawling module will only need to keep a much smaller representation of the website as opposed to before. A proposed methodology to simplify this process given the project specification is to create a crawler based on recommendations and an 'action replay' mechanism.

  The recommendation algorithm will need to be based off of similar existing algorithms in automatic crawlers. Since the tool must be driven by user input, the crawler would analyse contents and interactions of the web application with the user (as it would do automatically), and suggest specific features to the user as a starting point for the scan. This can be done by passively reading the contents of web requests between the user and the web application

and flagging up any that seem to exhibit behaviour of an insecure system, such as passing user inputs in clear text, or using client-side inputs to control important application sensitive logic. Once a user chooses to follow a recommendation, they may investigate the flagged feature more closely.

At this point, the 'action replay' algorithm begins - once a user has elected a potential target to test for a vulnerability, the tool begins to record user actions. Depending on the selected feature and what vulnerabilities may be discoverable, the tool can then suggest a stopping point of recording, or wait for the user to determine when their actions that detect a vulnerability are enough. The tool then takes these actions and analyses their outputs - it may be the case that a user has found a vulnerability on their first go. It could also be the case that this input didn't trigger a vulnerability, but is worth looking into further. At this point, the crawler will begin to fuzz different versions of input that may be more effective at showcasing vulnerabilities. In recent research by Parvez et al. into evaluating black box scanners [?], one of the final recommendations for a better scanner was to add interactive multistep options to the scanner, which is a main focus of this method. To the best of my knowledge, this 'action replay' algorithm is a novel approach in this area.

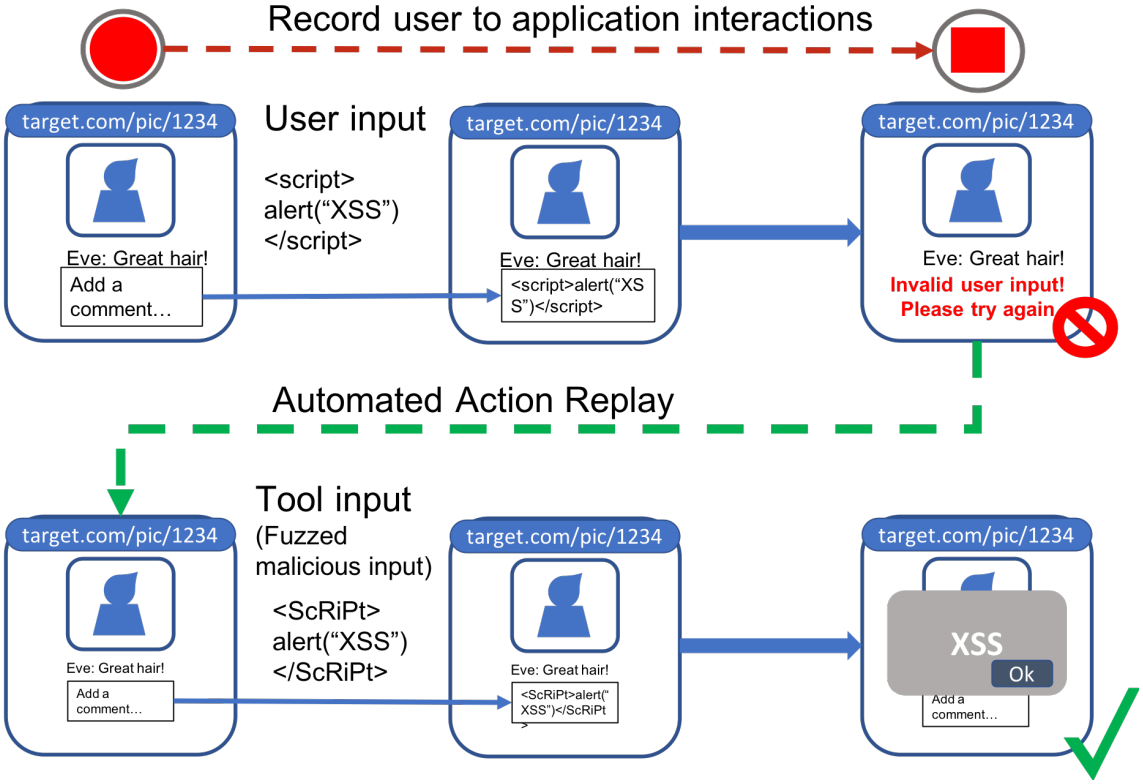

Figure 2.4: A visualization of the proposed action replay algorithm. The tool records user input for a time period determined by the user. The tool then replicates actions using fuzzed inputs to try and uncover vulnerabilities if the first attempt was unsuccessful.

- **Human judgement** - The previously mentioned paper by Parvez et al. also mentions that choosing the right attack vectors to exploit vulnerabilities is still a big challenge for black box scanners [?]. Huang et al. recently built a scanner that performed well against open-source competition [?]. However, these authors recently acknowledge that *"there is no silver bullet for web application security; threats will continue to grow and evolve"*. Herein lies much of the motivation for making this project user-centric: vulnerability detection and even exploitation may be automated [?], but for now, efforts in having to build an AI that is able to do this are astronomical - it takes years for teams of experts to build tools that do this. Following its recent success, the winner of the DARPA Cyber Grand Challenge (an AI-only capture the flag contest) was pitted against human professionals at the 2016 DEF CON CTF Challenge, where it came last [?]. These efforts show a bright future for fully automated

security systems, but these may still be some years away in the full making. This project does not admit defeat in attempting to automate the process of countering malicious agents, but rather aims to increase the likelihood of doing a good job against them by covering our bases with higher quality base defences. In fact, even limited domain knowledge has shown to be useful for human penetration testers; a group of students with 'average' security skills achieved a higher success rate on their own than some vulnerability scanners in analysing web applications [**?**]. It is hoped that with the correct suggestions provided by the tool, this project lowers the minimum requirements of a successful user to be 'below average' in their security domain knowledge.

2.5 Limitations

In order to be able to effectively evaluate this project at a later point, it is important to delineate realistic expectations in terms of what it can and cannot do.

2.5.1 Time taken

Since the extension is user driven and not fully automatic, the suggested process will be inevitably slower than if it was otherwise automated - there will be waiting times as users will not instantly attempt to uncover recommended vulnerabilities. There may also be some intellectual effort involved in crafting initial input to try and do this, which adds to this. Both of these factors add to 2 different metrics: the total scan time and the time taken from recommendation to decision of whether a vulnerability is found or not.

Total scan time may be especially hard to measure, and can be seen as a disadvantage of this scan. It is not known whether a user will eventually go through every possible page that is relevant, so it becomes very difficult to claim an end to the scan. On the contrary, automated crawlers *will eventually* come to an end their search, and thus put an upper bound on how many resources they can analyse. Human driven crawling may stop or resume at any given point. Not being able to best assess the total scan time is an acceptable tradeoff of this project as it will simply not be avoidable given human driven interaction is being explored.

The time taken to ascertain whether a vulnerability exists or not from a recommendation can however be bounded. This period begins when a user decides to investigate a recommendation on their own or by suggestion of the tool, and finishes either when enough proof of a vulnerability has been observed, or when all possible fuzzing opportunities in exploiting it have been exhausted - the extension will have a list of possible fuzzing mutations per type of vulnerability, so this work is bounded by that list. Again, since this is not a fully automated tool, the extension is expected to take longer in this metric due to human interaction when comparing it to automated scanners, which is only natural.

2.5.2 Breadth of work

Section 2.1 describes many potential kinds of vulnerabilities that the project may choose to tackle. Some of these are harder to detect than others, and thus require more development work. This large scope may make it tempting to try and undertake too many things at once. At the time of writing, it is also hard to assess just how difficult it may be to implement scanning for a specific type of vulnerability. The extension will be developed with aims of finding revelant vulnerabilities - sorted by both volume of occurrence and contemporary relevance as found on surveys. One of the most recurring vulnerabilities in both aspects is Cross Site Scripting, so developing the tool to be able to detect this class of weaknesses is a good starting point. Other features that may seem small (such as analysing cookies or the use of iFrames on a page), may also be beneficial to implement. As the project develops, it becomes harder to weigh the costs of effort to implement versus the success rate of focusing on a specific feature, so some further time should be allocated to allow for this meta research.

2.5.3 Self security

The security of the extension being developed should not be taken for granted. As mentioned above, Google Chrome has several mechanisms in place to safeguard extensions from falling prey to malicious attackers. Namely, these are:

- *Isolated Worlds*
- Privilege separation
- Predefined permissions
- CSP (Content Security Policy)

Although these practices make it much easier for a developer to avoid serious mistakes, it is still possible to write vulnerable extensions. The *threat model* in this case (the way we choose to archetype our potential enemy), is by means of a web attacker. This would be someone who sets up a 'honeypot' website, expecting the extension to scan it but in the process attempt to compromise the extension by different means. Due to the priveleges granted to an extension, this may result in the jeopardizing of sensitive user data, such as their passwords. A recent paper by Carlini, Felt and Wagner reinforces the notion that Chrome's existing techniques are effective in preventing extension compromise, but also list some developer practices that could result in vulnerabilities, such as the unrestricted use of the `eval` function by Javascript (which is known to be dangerous as it executes given strings as commands), and injecting website data into HTML [**?**]. Many of the notions mentioned in the paper are also cited in Google's own documentation on how to write Chrome extensions [**?**]. Following these best practices will decrease potential risks associated with this project.

2.5.4 Ethics & Handling of Results

This project aims to build a tool that helps users find vulnerabilities in web applications. Obviously, this may raise ethical ramifications as to how the tool interacts with websites, and how its output is handled.

As the tool needs to send requests on the user behalf when executing (especially so during the 'action replay' phase described above), the rate at which this is done may be of concern. Conventional automated scanners may fire 100's of different fuzz attempts at a web application, *per vulnerability, per existing page* - a shotgun-like approach. This approach is very intensive, and for web applications set up for smaller amounts of traffic, may result in a Denial of Service (DoS). This tool aims to reduce this by only passively reading user crawled content, and only actively interacting with the website whenever a flagged vulnerability has been detected.

A related concern is to do with the scope of testing and uses on a real web application. For testing and evaluation purposes, the tool will be ran against existing, known to be vulnerable applications, such as DVWA [**?**]. Ideally, it should also be ran against other web applications beyond my control. However, for some of these, it may be the case that running the unhinged extension may infringe usage terms and agreements. For this purpose, a proposed restrained mode for the tool may be built, such that it does not actively take any action when browsing web applications, but rather only passively reads network traffic to deduce and recommend potential vulnerabilities.

Naturally, as per the project ideals, any vulnerabilities that may be found as a result of running this extension will duly be reported to the appropriate developers.

Chapter 3

PROJECT X

Chapter 4

Evaluation

4.1 Intended functionality

In order to be able to perform the functionality as proposed above, the project should be able to demonstrate the following characteristics:

1. **Recommendation system** - The browser extension must be able to passively analyse web traffic and produce visual cues that indicate to the user that it has detected a potential vulnerability. From these, the user should be able to initiate a session where they can try and detect a vulnerability.

2. **Passive mode** - The extension is designed to eventually be able to take active steps to detect a vulnerability. It's undesirable for the extension to do this if these active steps may infringe terms of service of using the application. Active steps in this context encompass any automated features of the extension that behave on behalf of the user, such as sending web requests, filling out forms etc. Passive steps include the extension reading and analysing web traffic explicitly performed by the user, including suggesting recommendations based on the results of this analysis (the extension should not be liable for any user behaviour that may infringe terms of service, so if passive mode is not enabled, the user has no excuses so as to say they were not aware that the tool infringed these terms without their knowledge). Therefore, the extension must have a clearly visible toggle such that the user can enable or disable active vulnerability detection steps.

3. **Action Replay** - As described before, the tool should be able to record user inputs in a targeted domain from when they initiate a vulnerability scanning session until the user declares it is finished. The tool should then be able to replicate these steps automatically, while changing and fuzzing inputs when doing so. Each replication should attempt to reset the web application state inbetween tries to accurately represent fuzzing user input from the state they defined as a starting point for that particular scan.

4. **Educational functionality** - In order to be able to meet its purpose of being educational to users who don't necessarily run the extension with prior web security knowledge, the tool should have a limited library of plaintext knowledge to explain different vulnerability types, how these may be detected, and suggestions on how to exploit them thereafter. Snippets of this knowledge may be presented under the recommendations shown to users when using the extension. A more detailed explanation may be provided in the pop-up page that appears when the user clicks on the extension icon in the browser.

4.2 Metrics of success

To appropriately judge the success of the extension, we must have some quantifiable metrics, described below

1. **Time to vulnerability** - As mentioned in 2.5, measuring the total time taken to perform a scan in this human driven context is unfair; the tool does not set a boundary on total scans

unlike an automated scanner does. Measuring time taken is however a useful metric, so in order to not completely circumvent using time, another more appropriate metric is suggested - *time to vulnerability*. This will be measured as the total number of seconds taken from the moment a scan is initiated (the user clicks the visual cue given by the tool), until the scan is finished. A scan may finish under two circumstances: a vulnerability has been successfully reported or detected (this may happen at the first attempt by the user, or at a later try when the tool has fuzzed some inputs during the 'action replay' phase), or when the tool reports it could not find a vulnerability. These 2 conditions are bounded in terms of possible time taken, allowing the total time to be measured.

2. **Interaction volume** - A metric that is sometimes found when evaluating automated scanners is that of bytes sent and received by the application [**?**]. This quantifies the impact the tool has when stressing the website by fuzzing inputs. The smaller this metric is, the more efficiently the tool is performing its job, by detecting vulnerabilities with fewer web requests sent.

3. **Number of replays needed** - A similar metric to interaction volume, this number measures how many 'replays' are required by the tool to successfully detect a vulnerability. Both of these metrics measure the efficiency of the tool in detecting vulnerabilities, but this metric more accurately tests the efficiency of the fuzzing engine provided by the tool. Ideally, the tool uses the inputs which are known to work with higher success rates first, and the more esoteric fuzzes would come last, when there is already a decreased likelihood of them working anyway. It also encourages the tool to only include meaningful and relevant fuzzing techniques per vulnerability type, otherwise the tool *could* theoretically fuzz forever. The fewer number of replays needed by the tool in order to find a vulnerability, the better it is performing.

4.3 Experiments

4.3.1 Test benches

In order to test my tool during development, it is not necessary to build a vulnerable web application from scratch. Not only would this be time consuming (beyond the purposes of building a tool to exploit such an application), it would also bias the ensuing development of the extension so that it is tailored to successfully detect every injected vulnerability in the web application. Fortunately, there already exist tools dedicated to this purpose.

As previously mentioned, DVWA is a vulnerable web application written in PHP, using MySQL for database interactions [**?**]. This application contains a good selection of predefined vulnerabilities to test against:

- Brute Force Login
- Command Execution
- CSRF
- File Inclusion
- SQL Injection
- Upload Vulnerability
- XSS

Another existing tool is WebGoat [**?**, **?**], supported by OWASP. This is an actively supported project by the open source community, and also contains a healthy amount of potential vulnerabilities for developers to test against:

- Access Control Flaws
- AJAX Security
- Authentication Flaws
- Buffer Overflows
- Code Quality
- Concurrency
- Cross-Site Scripting (XSS)
- Improper Error Handling
- Injection Flaws
- Denial of Service
- Insecure Communication
- Insecure Configuration
- Insecure Storage
- Malicious Execution
- Parameter Tampering
- Session Management Flaws
- Web Services
- Admin Functions

Similarly to the above, there exist more testing tools of this kind, such as WackoPicko [**?**] and HacmeBank [**?**]. A more extensive list of existing applications of this kind has been produced by OWASP [**?**]. For the purposes of the experiments in this project, I will test against DVWA, WebGoat and WackoPicko, in order, as needed per vulnerability. This provides a good variety of implementations of vulnerabilities to test against; if the tool successfully finds the targeted vulnerabilities in these applications then it is likely to do well in other contexts.

4.3.2 Test methodology

The main method of testing my extension is to pit my extension against other existing automated vulnerability scanners. Some potential candidate scanners include *OWASP ZAP* [**?**], *w3af* [**?**] and *burpsuite* [**?**]. All the scanners will analyse the aforementioned vulnerable applications used in testing. To avoid the case where the produced extension overfits its scanning model to the test vulnerable applications, this experiment will include more applications from the OWASP vulnerable web apps list [**?**], not previously used in testing.

Since the browser extension makes use of user input, it is important to test the application with people of different security backgrounds. This relies on adequate classification in this group by the person undertaking the test. There will be 3 proposed categories of user experience when testing the tool; advanced, intermediate and beginner. An advanced user is expected to be well versed in web security, and have had prior experience in diagnosing (perhaps exploiting) vulnerabilities. An intermediate user may be someone getting to grips with this area, perhaps a student who is only now learning about these concepts, but hasn't *necessarily* got experience in detecting or exploiting vulnerabilities. Beginner users are expected to be web savvy, people who are acquainted with using browsers and web applications, but are not necessarily interested or knowledgeable in web security. It is hoped that enough data is gathered to be able to have at least 5 unique people per suggested user group - it may be particularly hard to find advanced users that are willing to test the tool, whereas users who fit the other categories should be much easier to find.

To quantify how well the tool performs its job, the success rates of all 3 groups will be analysed when using the extension to find vulnerabilities. A very successful implementation of the project will have made it easy for non-experts to detect vulnerabilities, meaning that results from the beginner group would not vary very much from those in the intermediate and advanced groups using the tool. Therefore, inter-group vulnerability detection success rates will be analysed. From these success rates, it may be possible to extrapolate data on how educational the project was to users in the beginner and intermediate groups. This data could be further backed up by an additional quiz on whether the user has understood the type of vulnerability they detected, and whether they understand the ramifications of doing so by providing an example of a potential exploit they might design as a result.

Additionally, comparing each group success rate to the success rates of each of the automated scanners is useful to be able to validate the claim that a user driven, semi-automated approach is advantageous. This should be done within the context of what vulnerabilities the extension is able to detect - if an automated scanner can find more types of vulnerabilities than the ones the extension has been designed to find, then these will not be counted in the results. For a fair comparison in that aspect, it is assumed that with more development time, the extension may be developed further to be able to identify another type of vulnerability.

Another means of testing the success of the application would be to put it to test against real world applications. By activating the described passive mode as needed, and browsing web application domains, it is possible that a user of the extension finds vulnerabilities in the target application. Should this happen, it would be a great validating factor for the success of the tool.

Chapter 5

Conclusion

Appendix A

First Appendix

Bibliography