Thesis for The Degree of Licentiate of Engineering

Security Analysis of Web and Embedded Applications

Benjamin Eriksson

CHALMERS

Division of Information Security
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2020

Security Analysis of Web and Embedded Applications

Benjamin Eriksson

Department of Computer Science & Engineering
Division of Information Security
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2020.

Abstract

As we put more trust in the computer systems we use the need for security is increasing. And while security features like HTTPS are becoming commonplace on the web, securing applications remains difficult. This thesis focuses on analyzing different computer ecosystems to detect vulnerabilities and develop countermeasures. This includes web browsers, web applications, and cyber-physical systems such as Android Automotive.

For web browsers, we analyze how new security features might solve a problem but introduce new ones. We show this by performing a systematic analysis of the new Content Security Policy (CSP) directive `navigate-to`. In our research, we find that it does introduce new vulnerabilities, to which we recommend countermeasures. We also create AutoNav, a tool capable of automatically suggesting navigation policies for this directive.

To improve the security of web applications, we develop a novel black-box method by combining the strengths of different black-box methods. We implement this in our scanner Black Widow, which we compare with other leading web application scanners. Black Widow both improves the coverage of the web application and finds more vulnerabilities, including ones in Prestashop, WordPress, and HotCRP.

For embedded systems, We analyze the new attack vectors introduced by combining a phone OS with vehicle APIs and find new attacks pertaining to safety, privacy, and availability. Furthermore, we create AutoTame, which is designed to analyze third-party apps for vehicles for the vulnerabilities we found.

Keywords: Vulnerabilities, Android Automotive, Content Security Policy, Web application scanning

Acknowledgments

There are many people that I want to thank for their positive impact on this journey. This endeavour would not have been possible without the amazing support my supervisor Andrei. Thank you Andrei for always inspiring me and pushing me to try new things, be it traveling across to world for internships, taking on new challenges at work or biking to new cities.

A huge thank you to all my amazing colleagues at Chalmers. You all make coming to work both fun and inspiring! Especially to my PhD buddy Alexander for being there for me since day one, always helping me when I'm lost, whether it is academic, technical, or personal, thank you for being there! Thanks to Iulia for all the fun and insightful discussions, teaching me things outside my bubble. I also want to thank Christoph from Mozilla for an amazing internship, super fun summer, and great supervision.

On the personal side, I owe a lot to Jonas for pushing me in the right direction, giving me the courage to pursue a PhD, and helping me co-author my first paper! A special thanks to Agustin, Alejandro, Ann-sofie, Anton and Matti for our daily trips to Verdansk.

Finally, a big thank you to my wife, best friend, and love of my life, Ann-sofie. Thank you for your immense support during this journey, for motivating me to work and for motivating me to do things outside of work.

Contents

Introduction

Whether you are doing online banking, using the web, or listening to music in your car, you are relying on the underlying systems to work efficiently and securely. More and more services are moving online and privacy concerns are increasing. At the same time, analyzing and improving security is no easy task in large ecosystems such and the web or Android. To combat this, we research and improve the security of the web browsers users use to connect to the web and the web applications they interact with. And as our cars become connected to the Internet and support the use of third-party apps the attack surface, as well as the potential impact of vulnerabilities, increases. We develop novel methods for finding vulnerabilities in these complex ecosystems, mitigations for these problems and we implement them in open-source tools that can be used by both industry and researchers.

Our research efforts can be divided into two major categories, web applications and embedded systems. For web applications we focus both on the client-side, ensuring the security of the web browser, and the server-side, ensuring the security of the application code running on the server. We investigate the interplay between the client-side and the server-side in the form of security policies. In the second paper [5] we research and improve a recently proposed security policy, covering the interplay between client and server. To help developers automatically generate these policies we created AutoNav, an open-source tool capable of analyzing web sites and suggesting policies. Improving the security of server-side code is the focus of the third research paper [4]. Here we develop Black Widow, a black-box web application scanner capable of finding more XSS vulnerabilities than previous efforts. For embedded systems, we are interested in the security implications

of porting a secure system, in this case Android, to new environments such as vehicles. We explore this in the first paper [3]. In addition, to exploring the new attack surface and presenting new attacks, we also create AutoTame, a static analysis tool able to find apps exploiting the new vehicle-specific vulnerabilities we present.

1 Web applications

Any website you visit on the web can be considered a web application. When you visit the website your web browser will send a request to the web application, which will be handled by the application code running on the web server. Once handled, the web application will respond with a complete web page.

For a secure web, it is important to improve the security of both the web browsers and web applications, as well as, the interaction between them. The following sections will cover the multiple different attackers we consider followed by the security concerns of both the client-side and the server-side in more detail.

1.1 Attackers

The web is a complex ecosystem that allows attackers to use a multitude of different attack vectors. To efficiently protect against these attackers it is crucial to understand their capabilities. The security literature [13] divides the attackers into four classes of attackers, injection, gadget, web, and network attackers.

Injection The injection attacker is the classic web application user. They can interact with the application to perform available actions, for example, comment on images, make their posts, leave reviews, etc. By carefully choosing which actions to perform, the attacker might be able to inject their JavaScript code into the web application. As the attacker is not part of the website, this is considered a Cross-Site Scripting (XSS) attack.

Gadget A gadget is a third-party code that is willingly being included on a website. Common examples are analytic scripts and frameworks, such as jQuery. A gadget attacker is an attacker that can change the gadget code, thus being able to attack multiple websites at the same time. Consider if `mail.com` includes the script `evil.com/analytics.js` then a gadget attacker would try to attack `mail.com` by changing the `analytics.js` code.

Web The unique capability of the web attacker is that they can host their website on the web. The attacker-controlled website can be used to redirect users to malware or force users to send requests to other websites.

The attacker-controlled website `evil.com` can force a user to initiates a request to `mail.com`. If the user is already authenticated with `mail.com` then the attacker could potentially *forge* a request to delete all the user's emails on `mail.com`. This is known as a Cross-site request forgery (CSRF) attack.

Network The final and strongest attacker is a network attacker. There are two types of network attackers, passive and active. A passive network attacker is capable of listening to all the traffic between the user and the website, while an active network attacker can also modify the traffic.

With this capability, the attacker can record passwords being sent to the website for later account takeovers. In the case of a bank application, an active network attacker would be able to change the recipient bank account of a transaction while it is being sent to the website.

1.2 Client-side

The focus of client-side security is to ensure clients, like web browsers, are protected from malicious web servers or, more commonly, web servers that have been hijacked by attackers. We can use the CIA triad to define security as protecting the confidentiality, integrity, and availability of services online. This includes ensuring that a website, e.g. `evil.com`, is not able to read or modify your emails at `mail.com`.

Consider the two first requests in the listing below, line 3 and line 4. A user visits `evil.com` and is then requested to fetch `index` from `evil.com` (Line 3) and also from `mail.com`. It is the fundamental security feature Same-Origin Policy (SOP) that decides to allow the first and block the second. SOP isolates different origins, where an origin is defined as a triplet of protocol (e.g. http), host (e.g. evil.com), and port (e.g. 80). This means that `evil.com` and `mail.com` are different origins and, as such, attempts to fetch data from each other should be blocked by the browser, as shown by the red dashed line in **??**. There are some exceptions to this rule, for example, `evil.com` could load images and scripts from `mail.com` (Line 5).

SOP helps protect confidentiality as cross-origin data reads are blocked. But SOP is not enough to protect integrity. Consider the POST request on line 6 in the code below where the browser is requested to send an email using `send_email.php` on `mail.com`. If the client is authenticated to `mail.com`, SOP will not block this request and an email can be sent. However, SOP will still block the response. This type of attack is known as a Cross-site request

forgery (CSRF). While this is commonly fixed on the server-side, by including special CSRF-tokens in requests, modern browsers are now stopping CSRF by settings SameSite cookies to *lax* by default [10].

The final action worth mentioning here is navigation. Although fetch and POST are secured in modern browsers, `evil.com` can still forcefully navigate a user to `mail.com` or some other web page hosting malicious or unwanted content (Line 7). To solve this, the new CSP directive `navigate-to` was proposed [15]. As the policy was a draft, and still is, it was the perfect opportunity to research its impact on the web security ecosystem. The results of this are presented in the second paper [5]. To better understand this ecosystem the next section will cover the possible attacks.

```
1 host: evil.com
2
3 FETCH     http://evil.com/index             Allowed
4 FETCH     http://mail.com/index             Blocked
5 FETCH     http://mail.com/img.jpg           Allowed
6 POST      http://mail.com/send_email.php    Allowed
7 NAVIGATE  http://mail.com/send_email.php    Allowed
```

1.2.1 Attacks and Countermeasures

The most notorious client-side attack on the web is XSS. In this attack, `evil.com` can execute JavaScript on `mail.com`, despite SOP. This is accomplished by injected data which contains HTML, for example: `<script>alert(1)</script>`. While any of the attackers in Section 1.1 can launch this attack, the injection attacker and the web attacker are the most common. If the web application on `main.com` is programmed incorrectly, it might output this data as HTML code. In this case, this would result in the JavaScript being executed and a popup being showed to the user. A more malicious attacker could leverage this JavaScript execution to steal passwords and other credentials. Although it is best to fix the problem in the code on the server-side, as we will explain Section 1.3, the browser can also help by using CSP.

Content Security Policy (CSP) [16] is a mechanism for websites to define security policies that the browser will enforce. For example, websites can define a policy to only allow loading JavaScript from `mail.com`. Crucial for XSS is that CSP can be used to block all inline JavaScript, which is the main attack vector for XSS.

A newer type of client-side attack is navigation attacks. In a navigation attack, users on `mail.com` could be redirected away to a malicious website like `evil.com`. This happened in on Equifax in 2017 where their users were

redirected to malware sites [1]. As mentioned in the previous section, the CSP directive `navigate-to` is being developed as part of CSP Level 3 [15] to mitigate navigation attacks. By using this new directive, Equifax could define the following policy to only allow navigations to their domain.

```
navigate-to: equifax.com
```

The `navigate-to` directive will protect against navigation attacks from all the attackers in Section 1.1, except the network attacker. This is because an network attacker could remove the policy in transit. To protect against this the website should use HTTPS.

In the second paper [5] we research this new navigation policy to test both if it introduces new vulnerabilities and what the performance impact on the web is. We discover that it efficiently protects against navigation attack but also introduces new methods to probe users for private data. In particular, the web attacker from Section 1.1 can abuse this policy gain information about the user visiting their website. To help the adaptation of this new policy we develop AutoNav in the second paper [5]. AutoNav is an open-source tool developers can use to scan their websites for outgoing navigations, mostly hyperlinks, and then get suggested navigation policies.

1.3 Server-side

In contrast to client-side security, server-side security focuses on ensuring the security of the server or the application. From the developer's perspective, this means writing code without bugs that attackers can exploit. However, writing bugfree code is hard, as is evident by the billions of credentials that have been stolen over the years due to poor security [9].

The following sections explain how web applications can be attacked (Section 1.3.1) and how we can help developers find and fix bugs (Section 1.3.2).

1.3.1 Web application vulnerabilities

Many possible vulnerabilities can be present in web applications. OWASP's top 10 list [14] is a collection of the most critical web application vulnerabilities, with vulnerabilities ranging from injection attacks to authentication misconfiguration and XSS. The work in this thesis mainly focuses on XSS as it is both very prevalent and easy to exploit [14].

XSS vulnerabilities are caused by a web application reflecting user input as HTML code. Consider a forum where users, more formally an injection attacker (Section 1.1), can post messages. If a user posts `<b>hello</b>` and this is directly added to the HTML code produced by the application then the

browser will interpret the `<b>` tag as HTML. This becomes more nefarious if the user posts a message containing `<script>` tags, as this allows them to execute JavaScript as the application. Using JavaScript attackers can steal cookies and other valuable information.

Protecting against XSS is, in theory, simple. By converting the HTML tag characters < and > with the escaped values *<lt;* and *>gt;*, a large portion of XSS is solved. Depending on the precise context, other characters might need escaping too, for example, quotes (") can be escaped as `&quot;`. In the code below we see two examples where user input, `$name`, and `$url`, are reflected without any escaping. In these cases, an attacker could exploit this to gain JavaScript execution.

```
SERVER-SIDE CODE          =>  GENERATED HTML

Hello $name               =>  Hello <script>alert(1)</script>
<a href="$url">link</a>   =>  <a href="" onclick="alert(1)">link</a>
```

What makes XSS hard to detect in practice is that it is hard to know when the data should be escaped for HTML. If the data is read from a database it can be hard to determine for a developer if a malicious user could control that data. Furthermore, escaping everything for HTML is not good either as it can cause problems when exporting the data to non-HTML platforms. This makes finding the vulnerabilities the major challenge of stopping XSS attacks. Finding these vulnerabilities if the main focus of the third paper [4].

1.3.2 Finding vulnerabilities

No developer is perfect and sooner or later a mistake will lead to a potential vulnerability in the code. At this point, it is important to have systems place to help find these vulnerabilities. This can range from manual security analysis to fully automatic code and application scanners.

Automatic scanning can be divided into two categories: white-box and black-box. White-box analysis can be used if application artifacts, such as source code, models, and code annotations, are available. In this case, the scanner can analyze these artifacts to uncover vulnerabilities. When these artifacts are not available, which is the standard case for penetration testing, black-box scanning can be utilized instead.

Black-box scanning dynamically interacts with the application, similar to how a user would. The scanner probes the application in different ways while analyzing the responses from the application for vulnerable patterns. For example, a black-box scanner can post `<script>alert(1)</script>` to a forum and analyze the response for a JavaScript alert message. If this is

detected it would support the hypothesis that the forum has an XSS vulnerability.

The main challenge of black-box scanning is how to interact with the web application. Modern web applications have complicated workflows where combinations of links, form submissions, and JavaScript actions are required. Additionally, the state of the web application is also important. For example, a prerequisite for adding a product review could be to add a product. In this case, the scanner would need to be able to add products before being able to test the security of the review functionality.

Previous research in the field of black-box scanners focused on one problem at a time. For example, the Jaek scanner focused on exploring JavaScript events [12]. The Enemy-of-the-state instead put the focus on modelling the state of the web applications [2]. In Black Widow [4] we combine the strengths of previous scanners while minimizing their weaknesses, in combination with novel detection methods for XSS vulnerabilities.

2 Embedded Systems

As embedded systems become more complex and allow for third-party code to run, they too need to ensure client-side security. A great example of such a system is Android. Android is a popular operating system for mobile phones. While Android is mainly developed by Google, it also allows third-party developers to create and distribute apps on the Google Play store. By allowing third-party apps users can quickly create and share their favorite apps without having to wait for the first-party company to develop them.

The downside with third-party apps is that it is hard to ensure that the apps are not malicious or simply poorly implemented and vulnerable. The main defense against apps stealing data or secretly recording your microphone is permission.

Google is currently developing a new version of Android, named Android Automotive, which will run in the infotainment systems of cars. Android Automotive provides an excellent chance to research the security implications of porting a relatively secure platform, Android on phone, to work in a new ecosystem of embedded systems in cars. This is the topic of the first paper [3].

Before explaining the security of Android Automotive, let us first explore the general security mechanisms in Android, the new features in Android Automotive, and finally the attack surface and possible attacks.

2.1 Permission model

If an Android app wants to access your camera, microphone, or location, for example, then the app must ask for permission to use this. The main two permission types in Android are *normal* and *dangerous* [8]. Lower impact APIs, such as Internet access or vibration control only requires *normal* permissions. These permissions are granted when the user installs the app. For more critical APIs, such as location, the *dangerous* permission are used. When an app tries to access such an API the user will receive a pop-up in which they have to grant the app access.

2.2 Android Automotive

Android Automotive [7] is a standalone version of Android which is designed to be used in vehicles. The operating system is used in the vehicle's infotainment system, usually a unit in the middle of the dashboard with a touchscreen. The infotainment system is responsible for presenting information, such as maps and location, as well as entertainment such as music and radio.

Android Automotive introduces new APIs to control the vehicle's heating, ventilation, and air conditioning system (HVAC). It also enables apps to read sensor data including speed, temperature, and engine RPM.

2.3 Attack surface in vehicles

As the connectivity of the infotainment systems is increasing with new features like Internet connectivity, WiFi, Bluetooth, and third-party code running locally, the attack surface is ever increasing. We have seen previous attacks on infotainment systems where attackers were able to take over a 2014 Jeep Cherokee by exploiting the Uconnect system [11].

Many of the low-level systems, both in Android Automotive and the vehicles, are already well explored. Communication protocols like WiFi and Bluetooth and tried and tested implementations. Similarly, the internal systems in the vehicle isolated into different networks with firewalls between, ensuring that the infotainment system cannot send break commands or turn off the engine.

No solution is perfect and researching how to break the internal firewalls and communication protocol implementations would be interesting. However, the target for the first research paper [3] was to identify what Android Automotive apps can do *within* the specification. That is, what attacks can apps perform without exploiting low-level vulnerabilities such as buffer overflows.

2.4 Attacks and Countermeasures

The attacks I've found in the research on Android Automotive can be divided into three categories, disturbance, availability, and privacy attacks.

Disturbance attacks is a novel vector since it targets a new asset, the attention of the driver. Android was not designed with this in mind since it is not critical what the user focuses on. In the first paper [3] we demonstrate how malicious apps can take over the stereo and play music on max volume while simultaneously overriding user input to turn down the volume. To counter this we develop AutoTame, a set of static analysis methods that can detect Android Automotive apps using dangerous APIs.

Availability attacks focus on acquiring as much system resources as possible, rendering other apps and the system itself unusable. A classic availability attack is the Fork Bomb which has been demonstrated to work on classic Android previously [6]. In a phone, the impact of such an attack is quite limited, simply reboot the phone and it is back to normal. However, for the infotainment system, which is used for navigation, having the user trying to figure out how to reboot or having to pull over to fix it is more severe. This problem is best solved on the OS level by limiting the number of processes an app can spawn, similar to how most desktop operating systems solve it. As a short term mitigation, we also develop static analysis methods to detect apps trying to abuse this.

Finally, privacy attacks try to gain and exfiltrate sensitive information. There are many interesting methods for abusing other apps, for example, an app without Internet permission can ask the web browser to open a URL to leak data. This is known as the confused deputy problem. In our research, we found that the default music player in Android Automotive was able to exfiltrate without giving any visual clues to the user. We also found new methods for acquiring sensitive information. In particular, Android Automotive allows apps to read sensor data like gear and RPM, which can be combined to calculate the velocity. This is significant since reading the gear and RPM does not require any permissions while reading the velocity requires elevated permissions. To counter this combinable sensors values should also require the app to request permissions.

Thesis structure

Paper 1: On the Road with Third-Party Apps: Security Analysis of an In-Vehicle App Platform [3]

This paper aims to uncover new security vulnerabilities and attack vectors from the porting of Android to the vehicle-specific Android Automotive. We

systematically investigate the attack surface of locally running third-party apps in vehicles. We present new attack vectors with the focus on disturbing the driver, potentially affecting road safety. We implement these attacks and test them both in Android Automotive emulators and physical testbeds supplied by Volvo Cars. To mitigate the problems we find we suggest enhancements to the permission model and improved API control. In addition, we develop AutoTame, for code analysis of vehicle-specific applications.

Statement of contributions This paper was in collaboration with Jonas Groth and Andrei Sabelfeld. Benjamin was responsible for finding and evaluating the attacks, designing the countermeasures and creating AutoTame.

Appeared in: Proceedings of the International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS), 2019.

Paper 2: AutoNav: Evaluation and Automatization of Web Navigation Policies [5]

This paper performs a first investigation of the new `navigate-to` CSP directive. We systematically analyze the potential vulnerabilities introduced by `navigate-to` with respect to the full web ecosystem. We demonstrate multiple attacks such as detecting if users are logged in to different websites, probing active shopping carts and bypassing third-party cookie blocking. We propose multiple countermeasures to these problems by enhancing the specification, browser implementations and web development practices. To aid web developers in adapting this new policy we develop the black-box scanner AutoNav. AutoNav can automatically crawl a website and infer navigation policies. We also introduce a simplification and wildcard algorithm to allow developers to easily optimize policies for performance or security. We evaluate AutoNav and the viability of `navigate-to` by an empirical study on Alexa's top 10,000 websites.

Statement of contributions This was a collaboration with Andrei Sabelfeld. Benjamin was responsible for analyzing the potential vulnerabilities, designing and implementing AutoNav, and performing the evaluation.

Appeared in: Proceedings of the Web Conference (WWW), 2020.

Paper 3: Black Widow: black-box Data-driven Web Scanning [4]

In this paper we improve the state-of-the-art in web application vulnerability scanning by designing a black-box scanner by using a novel combination of

black-box scanning techniques. We analyze the main challenges black-box scanners face in terms of vulnerability detection and code coverage. Based on this we develop Black Widow, a data-driven web scanner capable of following complex workflows and interact with JavaScript events. We evaluate our scanner on 10 web applications, including Drupal, HotCRP, Prestashop and WordPress, and show that our scanner improves code coverage by between 63% and 280% compared to the other scanners. In addition, we also find more XSS vulnerabilities than other scanners, including some in modern production software, including HotCRP, osCommerce, PrestaShop and WordPress.

Statement of contributions This paper was written in collaboration with Giancarlo Pellegrino and Andrei Sabelfeld. Benjamin was responsible for developing the new scanning method, designing and implementing the method in Black Widow and performing the evaluation.

To appear in: Proceeding of the IEEE Symposium on Security & Privacy (IEEE S&P), 2021.

Bibliography

[1] ars Technica. Equifax website borked again, this time to redirect to fake flash update, 2017. https://arstechnica.com/information-technology/2017/10/equifax-website-hacked-again-this-time-to-redirect-to-fake-flash-update/.

[2] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security Symposium 12*, pages 523–538, 2012.

[3] B. Eriksson, J. Groth, and A. Sabelfeld. On the Road with Third-Party Apps: Security Analysis of an In-Vehicle App Platform. In *International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)*, 2019.

[4] B. Eriksson, G. Pellegrino, and A. Sabelfeld. Black Widow: black-box Data-driven Web Scanning. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[5] B. Eriksson and A. Sabelfeld. AutoNav: Evaluation and Automatization of Web Navigation Policies. In *Web Conference (WWW)*, 2020.

[6] Y. Fratantonio. android-forkbomb, 2013.

[7] Google Inc. Automotive, 2018. https://source.android.com/devices/automotive/.

[8] Google Inc. Permissions overview, 2018. https://developer.android.com/guide/topics/permissions/overview.

[9] Information is beautiful. World's biggest data breaches & hacks, 2020.

[10] MDN Web Docs. Samesite cookies, 2020. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite.

[11] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015, 2015.

[12] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *International Symposium on Recent Advances in Intrusion Detection*, pages 295–316. Springer, 2015.

[13] P. D. Ryck, L. Desmet, F. Piessens, and M. Johns. *Primer on Client-Side Web Security*. Springer, 2014.

[14] The OWASP Foundation. Owasp top 10 - 2017, 2017.

[15] M. West. Content security policy level 3, 2018.

[16] M. West, A. Barth, and D. Veditz. Content security policy level 2, 2016.

1

On the Road with Third-Party Apps: Security Analysis of an In-Vehicle App Platform

Benjamin Eriksson, Jonas Groth, Andrei Sabelfeld

VEHITS 2019

Abstract. Digitalization has revolutionized the automotive industry. Modern cars are equipped with powerful Internet-connected infotainment systems, comparable to tablets and smartphones. Recently, several car manufacturers have announced the upcoming possibility to install third-party apps onto these infotainment systems. The prospect of running third-party code on a device that is integrated into a safety critical in-vehicle system raises serious concerns for safety, security, and user privacy. This paper investigates these concerns of in-vehicle apps. We focus on apps for the Android Automotive operating system which several car manufacturers have opted to use. While the architecture inherits much from regular Android, we scrutinize the adequateness of its security mechanisms with respect to the in-vehicle setting, particularly affecting road safety and user privacy. We investigate the attack surface and vulnerabilities for third-party in-vehicle apps. We analyze and suggest enhancements to such traditional Android mechanisms as app permissions and API control. Further, we investigate operating system support and how static and dynamic analysis can aid automatic vetting of in-vehicle apps. We develop AutoTame, a tool for vehicle-specific code analysis. We report on a case study of the countermeasures with a Spotify app using emulators and physical test beds from Volvo Cars.

1 INTRODUCTION

The modern infotainment system, often consisting of a unit with a touch-screen, is mainly used for helping the driver navigate, listening to music or making phone calls. In addition to this, many users wish to use their favorite smartphone apps in their cars. Thus, several car manufacturers, including Volvo, Renault, Nissan and Mitsubishi [42, 50], have chosen to use a special version of Android for use in cars, called *Android Automotive* [19]. Other manufacturers such as Volkswagen [49] and Mercedes-Benz [31] are instead developing their new in-house infotainment systems. In contrast to the in-house alternatives, Android Automotive is an open platform with available information and code. This justifies our focus on Android Automotive apps.

Android Automotive For the manufacturers, a substantial benefit of using an operating system based on Android is gained from relying on third-party developers to provide in-vehicle apps. A multitude of popular apps already exists on the Android market, which can be naturally converted into Android Automotive apps. Further, Android Automotive is a stand-alone platform that does not require a connected smartphone, contrary to its competitors MirrorLink [34], Apple CarPlay [1] and Android Auto [17].

Safety, security, and privacy challenges While third-party apps boost innovation, they raise serious concerns for safety, security, and user privacy. Indeed, it is of paramount importance that the platform both safely handles these apps while driving and also safeguards the user's privacy-sensitive information against leakage to third parties. Figure 1.1 gives a flavor of real-life safety concerns, by showing a user comment on a radio app with almost half a million downloads. The user points out that they had to stop driving when a shockingly loud ad was suddenly played, adding that ads "shouldn't attempt to kill you" [51].

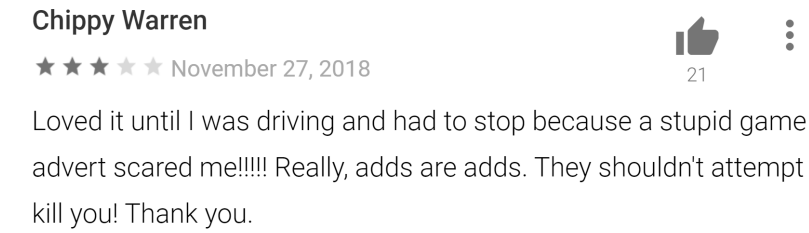

Figure 1.1: Top comment on a radio app. A user was shocked by the volume of an ad and had to stop driving.

While the Android Automotive architecture inherits much from regular Android, a key question is whether its security mechanisms are adequate for in-vehicle apps. However, compared to the setting of a smartphone, in-vehicle apps have obvious safety-critical constraints, such as neither being able to tamper with the control system nor being able to distract the driver. Further, car sensors provide sources of private information, such as location and speed or sound from the in-vehicle microphone. In fact, voice controls are encouraged for apps in infotainment systems, as to help keep the driver's hands on the wheel, opening up for audio snooping on users by malicious apps. A recent experiment done by GM collected location data and radio listening habits from its users with the goal of creating targeted radio ads [8]. This clearly highlights the value of user data in vehicles. Similar data could potentially be collected by apps using the radio API to record the current station [15]. Thus, a key question is whether Android's security mechanisms are adequate for in-vehicle apps.

Android Permissions Android's core security mechanism is based on a *permission model* [21]. This model forces apps to request permissions before using the system resources. Sensitive resources such as camera and GPS require the user to explicitly grant them before the app can use them. In contrast, more benign resources such as using the Internet or NFC can be granted during installation. However, there are several limitations of this model with implications for the in-vehicle setting. From a user's perspective, these permissions are often hard to understand. Porter Felt et al. [40] show that less than a fifth of users pay attention to the permissions when installing an app, and even a smaller fraction understands the implications of granting them. Understanding the implications of giving permissions is even harder. There are immediate privacy risks, such as an app having permission to access the car's position and the Internet can potentially leak location to any third party. More advanced attacks would only need access to the vehicle

speed. This may not seem like a privacy issue but by knowing the starting position, likely the user's home address, and speed, it is possible to derive the path that the car drives [16].

Analyzing Android Automotive Security To the best of our knowledge, this is the first paper to analyze application-level security on the Android Automotive infotainment system. To assess the security of the Android Automotive app platform, we need to extend the scope beyond the traditional permissions.

Attack surface For a systematic threat analysis, we need to analyze the attack surface available to third-party apps. This includes analyzing what effects malicious apps may have on the functions of the car, such as climate control or cruise control, and on the driver. We demonstrate *SoundBlast*, representative of disturbance attacks, where a malicious app can shock the driver by excessive sound volume, for example, upon reaching high speed. We also demonstrate availability attacks like *Fork bomb* and *Intent storm* which render the infotainment system unusable until it is rebooted. Further, we explore attacks related to the privacy of sensitive information, such as vehicle location and speed, as well as in-vehicle voice sound. We show how to exfiltrate location and voice sound information to third parties. In order to validate the feasibility of the attacks, we demonstrate the attacks in a simulation environment obtained from Volvo Cars. Based on the attacks, we derive exploitable vulnerabilities and use the Common Vulnerability Scoring System (CVSS) [13] to assess their impact.

To address these vulnerabilities, we suggest countermeasures of permissions, API control, system support, and program analysis.

Permissions We identify several improvements of the permission model. This includes both introducing missing permissions, such as those, pertaining to the location and the sound system in the car, as well as making some permissions more fine-grained. For location, there are ways to bypass the location permission by deriving the location from IP addresses. At the same time, the location permission currently allows all or nothing: either sharing highly accurate position information or not. The former motivates adding missing permissions, while the latter motivates making permissions more fine-grained. We argue that for many apps, like Spotify or weather apps, low-precision in the location, e.g. city-level, suffices.

API control In contrast to permissions, API control can use more information when decided to grant an app access to a resource. For example, using high-precision data could be allowed only once an hour, or during an activity like running. Our findings reveal that apps currently need access to the

microphone in order to use voice controls. We deem this as breaking the principle of least privilege [44]. To address this, we argue for full mediation, so that apps subscribe to voice commands mediated by the operating system, rather than having access to the microphone. Similarly, location data can also be mediated to limit the precision and frequency of location requests, making it possible to adhere to the principle of least privilege. These scenarios exemplify countermeasures we suggest to improve API controls for in-vehicle apps.

System We argue for improvements to the operating system in order to protect against apps using too much of the system's resources. Malicious apps can cause the system to become unresponsive or halt, either by recursively creating new processes or coercing other system processes to use up all the resources. The countermeasures consist of limiting the number of requests an app can make, limiting the resources system processes can use, or completely blocking some capabilities for third-party apps, like creating new processes.

Code analysis While previous methods protect the device from malicious apps, our vision is to also be able to stop the apps before they make it to the device. This can be accomplished by analyzing the code in the app store, before the app is published. This does not only protect against malicious apps but also poorly written apps that fail to adhere to security best practices. This could, for example, include apps not using encryption for data transmissions, which is currently a big problem [41]. Other problems include vulnerable apps with high privileges being exploited by malicious apps or colluding malicious apps sharing data over covert channels. Thus, we investigate how static and dynamic program analysis can be leveraged to address the vulnerabilities.

We design and develop AutoTame, our own static analysis tool for detecting dangerous use of APIs, including the new automotive APIs. AutoTame is open source and will be freely available at the time of publication. Further, we explore several state-of-the-art techniques, based on tools like FlowDroid [3] and We are Family [4].

Threat model The threat model in this paper defines the attacker as being able to install one or more apps, with the victim's permission, on their infotainment system. Similar to previous research [45], we assume that the victim is more inclined to install an app that asks for fewer permissions. This means that, while one app with access to both Internet and GPS might be considered suspicious, two apps, one with access to the Internet, the other with access to GPS, would be more acceptable. Such a model incentivizes apps

to collude and share information over covert channels. Using this model we analyze how much damage can be done by a user mistakenly installing malicious apps.

Case study An ideal evaluation of our countermeasures would be a large-scale of apps from an app store, in the style of the studies on Google Play, e.g. [3, 10, 37]. Unfortunately, Android Automotive is at this stage an emerging technology with no apps yet publicly available for a study of this kind. Nevertheless, we have been granted access to Infotainment Head Unit emulators and physical test beds from Volvo Cars allowing us to perform a case study with an in-vehicle app version of Spotify. We use this infrastructure to evaluate our countermeasures.

Impact At the same time, an early study of Android Automotive security has its advantages. Because our analysis comes at an early phase of Android Automotive adoption by car manufacturers, it has higher chances for impact. We have reported our findings to both Volvo Cars that participated in our experiments and Google. We are in contact with both on closing the vulnerabilities we point out and on experimenting with the countermeasures.

Contributions The paper offers the following contributions:

- We present an attack surface for third-party in-vehicle apps, identifying classes of disturbance, availability, and privacy attacks (Section 3).
- We propose countermeasures, based on fine-grained permissions, API control, system support, and information flow (Section 4).
- We overview prominent representatives of techniques and tools for detecting security and privacy violations in third-party apps (Section 4.4).
- We present our own static analysis tool, AutoTame, for detection of dangerous API usage (Section 4.4).
- We evaluate the countermeasures on a case study with the in-vehicle app Spotify (Section 5).

2 BACKGROUND

As cars become more connected and their infotainment systems more powerful, people expect the car to interact in a seamless way with their other devices. In contrast to most other personal devices, a software bug in a car

can have lethal consequences. For example, in 2015 Miller and Valasek [33] showed that it was possible to remotely take over a 2014 Jeep Cherokee by exploiting their infotainment system Uconnect. More recently, in May 2018, researchers found multiple vulnerabilities in the infotainment system and Telematics Control Unit of BMW cars which made it possible to gain control of the CAN buses in the vehicle [48]. These type of attacks show that remote take over attacks of connected vehicles is a possibility and a real threat.

Attackers do not necessarily need to take control over the braking or steering system to endanger or distract the driver. For example, an attacker can make a malicious infotainment app that disturbs or shocks the driver at a certain speed level. In order to shock the driver, the app may, for example, play loud music or rapidly flash the screen.

In addition to security, privacy is also a concern as cars become more capable of collecting data about their users. In accordance with the new EU regulation, GDPR [11], the user has to be informed about how the data is used and agree to their data being used in the described way. Previous research projects have explored the possibility to automatically track and analyze how privacy-sensitive information is leaked from Android apps [43], either deliberately through advertisement networks or inadvertently through insecure communication means [41].

2.1 Experimental Setup

With access to Volvo Cars internal testing equipment, both the attacks and countermeasures were tested on their infrastructure. In particular, the code is tested on Volvo's *Infotainment Head Unit emulators (IHU)* emulators and physical test beds. All of the Android code is developed for Android SDK version 26 and 27, which corresponds to Android 8.0 and 8.1.

2.2 Automatic analysis of Android apps

Automatically analyzing Android apps can be done through two major strategies, static analysis or dynamic analysis. Static analysis only considers the code while in dynamic analysis the code is executed and the program's behavior is analyzed. Which ever method is chosen, a decision on what to look for in the analysis has to be made. In this paper, two tracks are evaluated, how privacy-sensitive information flows through the app and scanning apps for common vulnerabilities.

2.3 Android Automotive

Today, the Android system is officially used in all types of devices, from phones and tablets to watches, TVs and soon cars [18]. Android Automotive is a version of Android developed specifically for use in cars. It is essentially Android with a User Interface (UI) adapted for cars and a number of car specific APIs. The car specific APIs allow for control over vehicle functions, such as the heating, ventilation, and air conditioning (HVAC), and reading of sensor data, e.g. speed, temperature and engine RPM [19]. Android Automotive is not be confused with Android Auto which is already available on the market today. Unlike Android Auto, Automotive is a completely stand-alone system that is not dependent on a smartphone. In Android Auto, apps run on the users Android phone which then renders content on a screen in the car. The apps and the Android system thus runs separated from the car.

2.4 Android's Permission model

The Android operating system controls access to many parts of the system, such as camera, position and text messages, through permissions. These permissions can be of one of four types; *normal*, *dangerous*, *signature* or *signatureOrSystem*. The first two are the most common and can be granted to any third-party app. Normal permissions give isolated accesses with minimal risk for the system and user, these are automatically granted by the operating system. Dangerous permissions, on the other hand, give accesses to private user data and control over the device that may harm the user. These permissions have to be explicitly granted by the user on a per application basis. Both Android's *coarse* and *fine* location permissions are examples of dangerous permissions, since both supply high precision data. The difference between them is that *fine* location has access to the GPS while *coarse* uses cell towers and WiFi access points. Finally, there are the *signature* and *signatureOrSystem* permissions, which requires the app to be pre-installed or cryptographically signed [20].

2.5 Covert channels

A *covert channel*, as defined by Lampson, is a communication channel between two entities that are not intended for information transfer [25]. In Android, a number of different covert channels exist that use both hardware attributes and software functions to communicate. Apps can for example communicate by reading and setting the volume, sending special intents or cause high and low system load [29, 45].

Table 1.1: The attacks are divided into three different categories. Which asset and permissions the attacks affects and requires are listed along with the needed user interaction.

Name	Category	Asset	User interaction	Permission	Severity
SoundBlast	Disturbance	Driver's attention	Start app	None	Medium[a]
Fork bomb	DoS	CPU resources	Start app	None	Medium
Intent storm	DoS	CPU resources	Start app	None	Medium
Permissionless speed	Privacy	Current speed	Start app	None	Low
Permissionless exfiltration	Privacy	Data Exfiltration	Start app	None	Low
Covert channel	Privacy	Data Exfiltration	Start app	Channel dependent	Low

[a] The score is subject to the limitation of CVSS3 on lacking support for physical damage and safety risks [9].

3 ATTACKS

This section focuses on the implementation decisions regarding the attacks presented in Table 1.1. The category and asset columns in the table give an understanding of what the attack is targeting. More specifically, the asset is what the attack is trying to take control over. In the case of denial-of-service (DoS) attacks, this is usually some type of resource. Privacy attacks, on the other hand, try to acquire and exfiltrate data such as speed or location. User interaction and permission are used to judge how easy the attack is to execute. The values are finally combined to create a severity score based on the Common Vulnerability Scoring System (CVSS3) [13]. A shortcoming of CVSS3 is that possible physical damage or safety risks are not considered in the scoring. Distraction vulnerabilities, like the one exploited by SoundBlast, and other automotive vulnerabilities will be underrated. These shortcomings are currently being revised for CVSS3.1 [9]. The exact vectors and scores for each attack are presented in Table 1.3. Table 1.2 present the same attacks together with suitable countermeasures to mitigate the underlying vulnerabilities.

3.1 Disturbance

SoundBlast The SoundBlast attack relies heavily on the `AudioManager` class in Android. This class supplies functions which are used to control the volume of different audio streams in Android. Cars also have the more specific `CarAudioManager`, however, this class requires special permissions. Different audio streams are used to differentiate between volumes, e.g. music volume, ringer volume, alarm volume, etc. A malicious app can use the permissonless audio API to max the volume and shock the driver. The attack is further improved by using a `ContentObserver` to listen for changes in volume and force the volume to the maximum as soon as it changes. Using the vehicle's sensors, the attacker can also design the attack to only active when traveling at high speeds.

Testing the SoundBlast attack shows that it is possible to set any volume on all the different audio streams in Android, without needing any permissions. In addition, the attack can also detect changes in volume and max the volume accordingly. The changes are also detected even if the driver uses the hardware controls on the IHU or steering wheel. Killing the app is the only way to regain control of the volume.

3.2 Availability

Fork bomb A fork bomb is a program that creates new instances of itself until the system runs out of resources, either freezing the device or force a reboot. While this might be acceptable on a phone, in a vehicle setting this is problematic. Since the IHU usually handles navigation, freezing the device might distract drivers trying to fix it, or frustrate them by having to stop and reboot.

Forking in Android is not possible by default, resulting in the need for a vulnerability to leverage in order to accomplish forking. Unlike previously successful fork bomb attacks on Android [2], our attack takes an application-level approach by creating a shell, which in turn has the power to fork itself. Similar to other programming languages, Android also supports a version of `exec`, which can be used to run external programs. However, this is not enough to create a new process that can copy itself. By using `exec` to run `sh -s`, a new shell is created, which in turn can execute the fork bomb.

When testing this attack it is able to fully grind both the emulator and test bed to a halt, requiring a power cycle to regain control. It is thus able to render the infotainment system unusable until the system is rebooted.

Intent storm The intent storm attack uses Android intents to continuously restart the app itself. Similar to the fork bomb presented in section 3.2, the intent storm attack tries to use up all the CPU resources, making the IHU unusable. The difference, however, is that the intent storm does not use the resources itself, but rather forces another system process, the *system_server*, to use up all resources. The fast activity switching required is made possible with threads and intents. As soon as the app starts, it spins up 8 threads which all ask Android to start its own main activity. Using multiple threads increases the pressure on the *system_server*, making the device less responsive.

During the tests, the *system_server* process was forced by the attack to use 100% of the CPU, making the IHU unusable. In some cases, an error message popped up on the device prompting the user to either kill or wait for the app. Regardless of which alternative was picked, the attack would

continue without interruption since a request to restart the app had already been sent. Similar to the fork bomb in section 3.2, this would grind the IHU to a halt. However, in some cases, the IHU would automatically restart after a few minutes.

3.3 Privacy

Permissionless speed In Android Automotive, apps have direct access to the current speed. However, since speed is privacy sensitive it requires a permission. By combining other permissionless sensor values, such as the current RPM and gear, and knowledge about the wheel size, the speed can be derived. The effectiveness of this attack does depend on the sampling frequency of the sensors. The hardware test beds only contained the IHU and not the full car, meaning that the efficiency of the attack is yet to be tested.

Permissionless exfiltraion The Android permission model clearly states that any app wanting communicate on a network requires Internet permission. However, by using intents it is possible to force another app with Internet permission to leak the data. Depending on how the intent is crafted, different apps will handle them, for example, the web browser will open URLs, music player opens music files, etc.

While the implementation details differ depending on which app handles the intent, the common procedure is to encode the data, split it into chunks and send a separate intent for each chunk.

While the default web browser can be used, there are better options for exfiltrating data. By changing the data type to `audio/wav` and using the URL `http://evil.com/music.wav?d=[data]`, the music player will load the URL instead. The stealthiness of this method depends on which music player is used. Using the native Android music player, a small popup with a play button will appear. By returning a malformed wav file from the server, the music player will show a more subtle error message.

If a web browser is used, the attacker can have the server redirect the request to a deep link, giving control back to the exfiltration app. Not only does this give the app the ability to leak more data, but it also enables two-way communication with the attacker's server, all without using the Internet permission.

In order to test this, a proof-of-concept code was developed that would record audio for five seconds and then upload it using the described method. The code only needs permission to record audio, but not to use the Internet.

Testing this attack shows that it is possible to send data to the Internet without using the Internet permission. The attack was successful using Chrome, the standard music player, video player and image viewer. If the device has not been configured with a default application for opening the type of data, it will ask the user to pick one.

Covert channels Previous work on covert channels in Android have used both vibration and volume settings to transmit data between colluding apps [45]. While these are still viable in Android Automotive, there are also additional new interesting APIs. In particular the new climate control API for temperature. Since the temperature is represented by a floating point value, the bandwidth is more than tenfold that of the volume settings. However, changing the temperature does currently require a signature permission, making it hard for third-party apps to acquire.

In contrast to previous work on covert channels, which relied on time synchronization, our attack is based on asynchronous messages. This forces the receiver to send an acknowledgment for each of the received values. While this lowers the bit rate, in contrast to synchronous communication, it greatly increases the reliability of the communication.

With this implementation, two apps can collude to leak privacy- sensitive information to the Internet. One app requests permission to privacy-sensitive information but not the Internet and then acts as a sender. The second app requests Internet permission but not permission to access any sensitive data. The second app can now receive sensitive information which it does not have permission for and leak it to the Internet.

4 COUNTERMEASURES

The vulnerabilities are very different in nature and, as such, the mitigation techniques differ. Some vulnerabilities can be mitigated by several different techniques while others can only be mitigated by one. An overview of the attacks together with mitigations for the underlying vulnerabilities are presented in Table 1.2.

4.1 Permission

The current permission model can be improved both by adding new permissions for unprotected resources, and also by refining some very broad permission. The SoundBlast attack, from Section 3.1, relies on changing the volume through an API called *AudioManager* which does not require any sort

of permission. At the same time, there exists an API called *CarAudioManager*, which does require a permission. Cars usually have more advanced sound systems than phones so a different API with more settings does make sense as does the need for a permission. Still, when conducting experiments with the emulator the *AudioManager* is present and usable by third-party apps, thus allowing an attacker to circumvent the permission required by *CarAudioManager*.

In addition to audio, Android allows apps to get the location of the device by using GPS. This can, for example, be used by apps to give weather information. However, due to these systems having high precision and allowing for multiple requests within short time intervals, apps often excessive information.

There are multiple methods for preserving the user's privacy while still maintaining an acceptable level of functionality in apps using location [12, 32]. Which method is optimal is highly dependent on the type of information the app needs. A simple approach is to truncate location, effectively creating a grid of possible locations. A grid will better protect the privacy of the user, but at the same time degrade the functionality of some apps [32]. In order to handle apps like fitness trackers, which requires fast updates and high precision, truncation is not feasible. Fawaz and Shin [12] argue that in order to preserve privacy, a choice has to be made between tracking distance and speed, or tracking the path of the exercise. They present a method for tracking the distance and speed by supplying the exercise tracker with a synthetic route, that has correct distance and speed but a forged path. Furthermore, they argue that navigation apps with Internet access, usually used for real-time traffic information, are the hardest to handle since they can potentially leak the location. This problem could be solved by using state-of- the-art information flow tracking to ensure that the location is never leaked.

4.2 API control

In some scenarios, permissions are not enough. This is usually the case when access to a resource can be abused over time. For example, in the current Android model, apps are allowed to record audio from the microphone at all times, as long as it has been granted the permission once. This means that a restaurant app that uses voice commands to find close by restaurants, can listen to everything the user says, at all times. Since voice commands are more prevalent in vehicles, where the user's focus is on driving, it is reasonable to believe that more in-vehicle apps will use this functionality. One solution to this problem is to use a voice mediator, which is a special service that has access to the microphone and allows for third-party apps to subscribe

to certain keywords. The app would only receive sentences that contain the keywords it subscribed to, effectively removing its capabilities to eavesdrop. Similar to the voice mediation, the same method can be used for location. By using a location mediator apps can subscribe to arbitrary precision for location data. The mediator can also introduce a trade-off between the refresh rate and precision of the requests, mitigating real-time tracking.

4.3 System

Some problems are best solved at the operating system level. These problems include resource management, e.g. how much CPU time or memory an app should be allowed to use. One method of limiting the impact of availability attacks is by limiting how frequent a resource can be acquired. Android already does this to a great extent when it comes to memory and CPU usage by third-party apps. However, some system processes, the *system_server* process in particular, can use all of the CPU, effectively starving the rest of the system. This lack of rate limiting was exploited in the intent storm attack in Section 3.2. While not tested, we speculate that this vulnerability could either be countered by rate limiting the CPU usage of the *system_server* process or limit incoming intents to the *system_server*.

Similar to CPU limiting, memory usage requires limitations too. When Android is running low on memory it will start to terminate apps in the background. This can sometimes result in the termination of apps that the user wants to run in the background. In the case of vehicles, navigation apps are a good example of apps that should not be killed of while driving. A possible method for ensuring that the navigation works while driving is to prohibit Android from terminating important apps. This protects against both malicious apps using up the memory, and legitimate memory hungry apps.

Akin to permissions, SELinux policies are policies which limit what the processes in an OS can do. These policies play a crucial role in protecting the vehicle's subsystems from Android. The policies are also suitable for specifying what an app is allowed to do. However, not how many times it can do it. As Bratus et al. [5] explains, "SELinux does not provide an easy way to control the use of the fork operation once forking has been allowed in the program's profile", which shows that SELinux is not suited to stop attacks like fork bombing. While it might be infeaseable in many situations, blocking forking altogether could be a solution.

4.4 Code analysis

Automatic analysis techniques can be used to scan apps, both before installation and during runtime, to find vulnerabilities and block attacks. In the following sections tools using these techniques are described in more detail.

Vulnerability detection Both AndroBugs [26] and QARK [27] are tools that can be used to scan Android apps for known vulnerabilities. QARK is capable of finding many common security vulnerabilities in Android apps [22]. QARK can, for example, find incorrect usage of cryptographic functions, trace intents and detect insecure broadcasts. In addition, QARK can also generate exploits for some of these vulnerabilities. While not able to generate exploits, AndroBugs can detect vulnerabilities based on heuristics in the code. For example, multiple dex files suggests a master key vulnerability (CVE-2013-4787) [35]. The tools work well together since AndroBugs can quickly scan multiple apps with heuristics and then QARK can perform a deeper analysis of the interesting apps.

AutoTame To scan for dangerous use of the new automotive APIs, we developed a special tool built on the Soot framework, which can analyze both Java and Android bytecode. The tool has a list of dangerous APIs, e.g controlling the HVAC system, change audio volume or spawning shells. Using Soot, our tool decompiles the APK and analyses each function in the app while testing if it matches any of the ones in the list. AutoTame performs a full application analysis. The main advantage of this is that it does not require any entry point analysis. Compared to many other languages, Android apps do not have a single main function from which execution starts. Therefore a full analysis ensures that any dangerous use of an API is detected. However, without knowing the entry points, dead code could be flagged, potentially leading to false positives. In addition to only detecting if the volume is changed, AutoTame can also give extra warnings if the volume is set to a high numeric value or if `getStreamMaxVolume` is used. If a match is found the app can be removed or marked as potentially dangerous. The tool was able to flag the SoundBlast attack, as well as the fork bomb.

Taint tracking Taint tracking can help detect privacy leaks where sensitive information, such as the user's location, is being sent to a remote server. FlowDroid [3] is a tool for static taint analysis on Android, that can detect these flows. The taint analysis works by tainting private sources of information, such as the user's location. If the location is written to a variable, then this variable also becomes tainted. If at a later time this tainted variable

Table 1.2: List of all developed attacks and which countermeasure(s) can be used to mitigate each attack the underlying vulnerabilities.

Attacks / Countermeasures	Permissions	Location granularity	SELinux	AutoTame	FlowDroid	We are Family	Rate limit
SoundBlast	✓			✓			
Fork bomb			✓	✓			
Intent Storm							✓
Permissionless speed		✓			✓	✓	
Permissionless exfiltration		✓			✓	✓	
Covert channels	✓	✓			✓	✓	

is written to a public sink, e.g an Internet connection, a leak from a private source to a public sink will be detected.

What makes FlowDroid special is its highly accurate modeling of Android's life cycles. This is important as an app can be started in many different ways. In addition to life cycles, FlowDroid is also able to track callback functions, enabling it to track leaks via button clicks and other UI events. Important for the car API used in this paper is that FlowDroid can track dynamically registered callback functions, which is used to establish the connection to the car.

In order to make FlowDroid fully functional with Android Automotive apps, we extended the tool with new sources and sinks. Some of the sources added were used to acquire the car's manufacturer, model and year. For sinks, we added functions for writing to the climate control APIs.

Observable flows Taint tracking is not always enough to find all privacy leaks. For this reason, a more powerful tool that can detect observable implicit flows is introduced. The *We are Family* paper by Balliu et al. [4] presents a two-fold hybrid analysis solution. The first stage is a static analysis that transforms the application and adds monitors. These monitors will aid the dynamic analysis tool in the second stage to find implicit flows. The added monitors are in this case used to track the program counter label and analyze the current taint value, making it possible to detect potential leaks during runtime on the device. The dynamic tool developed in the paper is an extension of TaintDroid [10]. By using the transformed program together with TaintDroid, the new tool is able to detect observable implicit flows, something TaintDroid was not able to do.

5 SPOTIFY CASE STUDY

To test some of the countermeasures, an in-depth case study was performed on the Spotify app. The motivation behind using Spotify is that it was the only third-party app available on the emulator and test bed, making it the

most realistic app to test. It was also much larger in size than the proof-of-concept attacks. The larger size will show how well the methods handle real apps.

5.1 Permissions

The first analysis that has to be performed is to gather an understanding of the permissions the app uses. Spotify needs permission to Internet, Bluetooth and NFC, for data transfer. Furthermore, it also requires permission to change audio settings, run at startup, and prevent the device from sleeping. Since Spotify is a music streaming app that should be able to run in the background, as well as talk to other Bluetooth devices, these permissions seem innocuous. Shifting focus to the *dangerous* permissions, Spotify does require permission to read the accounts on the device, contacts stored on the device, the device ID, and information about current calls. It is not clearly motivated why this information is necessary, and while some connection between the Spotify user and the device user is reasonable, having access to all contacts seems excessive. Spotify does not ask for the location permission, instead, they use IP-addresses for location [47]. In addition, Spotify can also record audio and take pictures, as well as read and write access to the external storage. Taking pictures is necessary to scan QR-codes and the microphone will be used in Spotify's driving mode [46]. Access to external storage is reasonable since it allows for offline storage of music, however, it does include access to other photos and media files beyond Spotify's.

5.2 Vulnerability detection

To ensure that the app does not have any known vulnerabilities QARK is used to scan the app. While QARK didn't find any severe vulnerabilities, it did find cases where a vulnerability could arise, e.g. by using a WebView in an older version of Android (API $\leq$ 18). Moreover, it also points out interesting entry-points into the app, one of them leading to a version of Spotify meant for another automotive system. In addition, a malicious third-party app can also send intents to Spotify to search and play arbitrary music, skip songs, or even crash the app. QARK did not find any vulnerabilities relating to the vehicle APIs, motivating the need for further analysis.

5.3 AutoTame

Using AutoTame, multiple warnings about both changing the volume and querying for max volume was found. Further manual analysis proved that

the maximum volume was used directly to set the volume, as shown in Figure 1.2.

```
1  int i = this.c.getStreamMaxVolume(0);
2  this.c.setStreamVolume(0, i, 0);
3
```

Figure 1.2: Decompiled code setting volume to max

5.4 Information flow analysis

The permissions give an upper bound on what the app is capable of doing. A more precise understanding of the app is achieved by analyzing it with FlowDroid, using implicit flow tracking. Using these settings the information flow analysis found 13 leaks in the app. One interesting leak was `getLastKnownLocation` being leaked into a dynamic receiver registration. As shown in Figure 1.3, FlowDroid was able to track the sensitive location through different assignments, function calls and control flows. While this case might be quite benign, as it only leaks one bit, it still shows the capabilities of the technique.

The analysis also over-approximates some leaks, especially when the information being sent is based on information being received. A concrete example of this is when threads try to communication using `sendMessage` and `obtainMessage`. Since the obtained information could contain sensitive information, it is flagged as a leak. This could potentially be solved using dynamic information flow tracking.

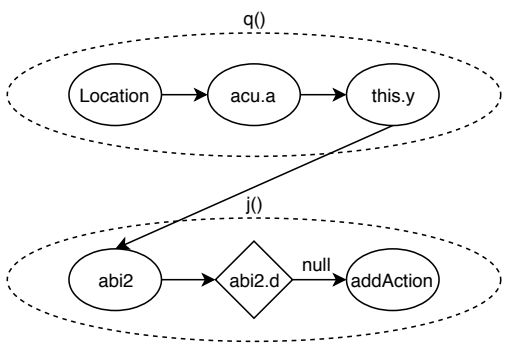

Figure 1.3: The publicly observable addAction function is implicitly dependent on the private location information.

5.5 Summary

To summarize these findings, we see that a more robust and at the same time more fine-grained permission model would be beneficial, as it would allow apps like Spotify to use lower precision location data instead of privacy-invading high precision data. In addition, vulnerability detection methods succeed in finding a bug that could be exploited to terminate Spotify. Finally, static analysis proved successful for automatically detecting privacy leaks.

6 RELATED WORK

Previous security and privacy research on vehicles have to a large extent focused on low-level problems relating to the internal components. Koscher et al. [24] showed that with physical access to the CAN bus it is possible to control both the speedometer, horn and in-vehicle displays to distract the driver. Miller and Valasek [33] gained similar access to the CAN bus, this time remotely. A similar vulnerability found in an infotainment system used in cars from Volkswagen was also recently discovered by researchers in the Netherlands [7]. They showed that it was possible to connect to the car via WiFi to exploit a service running in the infotainment system to gain remote code execution system. The most recent study on attacks against vehicles were done by researchers at Tencent Keen Security Lab [48], where they found multiple vulnerabilities in the infotainment system and Telematics Control Unit of BMW cars, resulting in control of the CAN buses.

A contribution of our paper is to show that even without access to the internal buses or exploiting low-level vulnerabilities, it is possible to cause distractions and leak private information.

A more high-level study was done by Mazloom et al. [30] where they conducted a security analysis of the MirrorLink protocol. MirrorLink allows smartphones to run apps on the cars infotainment system. Their analysis showed weaknesses in the MirrorLink protocol which could, amongst other things, allow malicious smartphone apps to play unwanted music or interfere with navigation. Mandal et al. [28] showed that the similar system Android Auto have multiple problems that can be abused by third-party apps. For example, auto playing audio when launching an app or showing visual advertisements, both which are against Android Auto's quality policy. In our paper, we show similar attacks are possible on Android Automotive, however, without the requirement of the user's smartphone, since the malicious app runs on the infotainment system.

Intents, which is the main component in our exfiltration attack, are prob-

lematic for many reasons. Khadiranaikar et al. [23] highlighted some of these problems, including how malicious apps can both steal information and compromise other apps using intents. Our paper builds on these ideas to develop new exfiltration methods for the Android Automotive platform.

There is a large body of work on Android permissions [14, 38, 39]. As a representative example, a study on Android permissions by Porter Felt et al. [37] shows that many apps are using more permissions that they need, i.e. not adhering to the principle of least privilege. Other researches [6], also argue for the need of a more fine-grained model which can grant access to specific functions instead of full APIs or services. Extensions such as Apex [36] have also been developed in order to supply end users with a more fine-grained model, capable of granting permissions based on user-specified policies. While our focus is on the specifics of the in-vehicle setting, we argue that many apps get access to more data than necessary due to the coarse granularity of the permission model itself. For example, a weather app or Spotify app only needs low-precision location, such as city level.

7 CONCLUSIONS

To the best of our knowledge, we have presented the first study to analyze application-level security on the Android Automotive infotainment system. Unfortunately, our analysis shows that in-vehicle Android apps are currently as secure as regular phone apps. We argue it is insufficient because in-vehicle apps can affect road safety and to some extent user privacy.

Our study of the attack surface available to third-party apps include driver disturbance, availability, and privacy attacks, for which there is currently no protection mechanisms in Android Automotive.

Consequently, it is important for car manufacturers that third-party apps are limited in their abilities to cause a considerable distraction for the driver. Additionally, there are a number of vehicle specific APIs, such as access to current gear and engine RPM, that is a cause for concern when it comes to user privacy.

To address the vulnerabilities that lead to these attacks, we have suggested the countermeasures of robust and fine-grained permissions, API control, system support, and program analysis.

We have designed and developed AutoTame, a tool for detecting dangerous vehicle-specific API usage. We have demonstrated that in-vehicle code analysis can be performed using AndroBugs and QARK, to detect known vulnerabilities, AutoTame to detect vehicle specific vulnerabilities and FlowDroid, with the additional vehicle specific sources and sinks, to detect privacy

leaking apps.

We have evaluated the countermeasures with a Spotify app using an infrastructure of Volvo Cars.

Bibliography

[1] Apple. Apple carplay, 2014. http://www.apple.com/ios/carplay/.

[2] A. Armando, A. Merlo, M. Migliardi, and L. Verderame. Would you mind forking this process? a denial of service attack on android (and some countermeasures). In *IFIP*, 2012.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[4] M. Balliu, D. Schoepe, and A. Sabelfeld. We Are Family: Relating Information-Flow Trackers. In *European Symposium on Research in Computer Security*, 2017.

[5] S. Bratus, M. E. Locasto, B. Otto, R. Shapiro, S. W. Smith, and G. Weaver. Beyond selinux: the case for behavior-based policy and trust languages. 2011.

[6] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *USENIX Security Symposium*, 2013.

[7] Computest. Research paper: The connected car - ways to get unauthorized access and potential implications. Technical report, April 2018.

[8] Detroit Free Press. Gm tracked radio listening habits for 3 months: Here's why, 2018. https://eu.freep.com/story/money/cars/general-motors/2018/10/01/gm-radio-listening-habits-advertising/1424294002/.

[9] D. Dugal. List of potential improvements for cvss 3.1, 2018.

[10] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

[11] European Commission. Regulation (eu) 2016/679, 2016. `http://ec.europa.eu/justice/data-protection/reform/files/regulation_oj_en.pdf`.

[12] K. Fawaz and K. G. Shin. Location privacy protection for smartphone users. In *SIGSAC '14*, 2014.

[13] FIRST.Org Inc. Common vulnerability scoring system v3.0: User guide, 2018.

[14] M. Frank, B. Dong, Porter Felt, and D. Song. Mining permission request patterns from android and facebook applications. In *ICDM '12*. IEEE, 2012.

[15] A. Gampe. Radiotestfragment, 2018. `https://android.googlesource.com/platform/packages/services/Car/+/4d1e3469cb2f285e7a4a864bd48a4c5177e7c83f/tests/EmbeddedKitchenSinkApp/src/com/google/android/car/kitchensink/radio/RadioTestFragment.java`.

[16] X. Gao, B. Firner, S. Sugrim, V. Kaiser-Pendergrast, Y. Yang, and J. Lindqvist. Elastic pathing: Your speed is enough to track you. In *ubicomp 2014*, 2014.

[17] Google Inc. Android auto, 2014. `https://www.android.com/auto/`.

[18] Google Inc. Android, 2018. `https://www.android.com/`.

[19] Google Inc. Automotive, 2018. `https://source.android.com/devices/automotive/`.

[20] Google Inc. permission, 2018. `https://developer.android.com/guide/topics/manifest/permission-element.html`.

[21] Google Inc. Permissions overview, 2018. `https://developer.android.com/guide/topics/permissions/overview`.

[22] F. Ibrar, H. Saleem, S. Castle, and M. Z. Malik. A study of static analysis tools to detect vulnerabilities of branchless banking applications in developing countries. In *ICTD '17*, 2017.

[23] B. Khadiranaikar, P. Zavarsky, and Y. Malik. Improving android application security for intent based attacks. In *IEMCON 2017*, Oct 2017.

[24] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, 5 2010.

[25] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.

[26] Y.-C. Lin. Androbugs framework, 2018. `https://github.com/AndroBugs/AndroBugs_Framework`.

[27] LinkedIn Corporation. Qark, 2018. `https://github.com/linkedin/qark`.

[28] A. K. Mandal, A. Cortesi, P. Ferrara, F. Panarotto, and F. Spoto. Vulnerability analysis of android auto infotainment apps. In *CF '18*. ACM, 2018.

[29] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *ACSAC '12*, 2012.

[30] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy. A security analysis of an in-vehicle infotainment and app platform. In *WOOT*, 2016.

[31] Mercedes-Benz. Mercedes-benz user experience: Revolution in the cockpit, 2018. `https://www.mercedes-benz.com/en/mercedes-benz/innovation/mbux-mercedes-benz-user-experience-revolution-in-the-cockpit/`.

[32] K. Micinski, P. Phelps, and J. S. Foster. An empirical study of location truncation on android. *Weather*, 2:21, 2013.

[33] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015, 2015.

[34] MirrorLink. Mirrorlink, 2009. `https://mirrorlink.com/`.

[35] MITRE. CVE-2013-4787. Available from MITRE, CVE-ID CVE-2013-4787., 2013. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4787.

[36] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS '10*, 2010.

[37] Porter Felt, E. Chin, S. Hanna, D. Song, and D. A. Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security*, pages 627–638. ACM, 2011.

[38] Porter Felt, S. Egelman, M. Finifter, D. Akhawe, D. Wagner, et al. How to ask for permission. In *HotSec*, 2012.

[39] Porter Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[40] A. Porter Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. A. Wagner. Android permissions: user attention, comprehension, and behavior. In *SOUPS*, page 3. ACM, 2012.

[41] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying tls usage in android apps. In *CoNEXT '17*, 2017.

[42] RenaultâĂŞNissan Alliance. Renault-nissan-mitsubishi and google join forces on next-generation infotainment, 2018. `https://www.alliance-2022.com/news/renault-nissan-mitsubishi-and-google-join-forces-on-next-generation-infotainment/`.

[43] I. Reyes, P. Wiesekera, A. Razaghpanah, J. Reardon, N. Vallina-Rodriguez, S. Egelman, and C. Kreibich. " is our children's apps learning?" automatically detecting coppa violations. In *ConPro'17*, 2017.

[44] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1975.

[45] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS '11*, 2011.

[46] M. Singleton. Spotify is testing a driving mode feature, 2018. `https://www.theverge.com/2017/7/7/15937284/spotify-driving-mode-feature-testing`.

[47] Spotify. Privacy policy, 2018. `https://www.spotify.com/us/legal/privacy-policy/`.

[48] Tencent Keen Security Lab. New vehicle security research by keenlab: Experimental security assessment of bmw cars, 2018. https://keenlab.tencent.com/en/2018/05/22/New-CarHacking-Research-by-KeenLab-Experimental-Security-Assessment-of-BMW-Cars/.

[49] Volkswagen. 2018 passat press kit, 2018. https://media.vw.com/en-us/press-kits/2018-passat-press-kit.

[50] Volvo Car Group. Volvo cars to embed google assistant, google play store and google maps in next-generation infotainment system, 2018. https://www.media.volvocars.com/global/en-gb/media/pressreleases/228639/volvo-cars-to-embed-google-assistant-google-play-store-and-google-maps-in-next-generation-infotainme.

[51] C. Warren. Radio fm, 2018. https://play.google.com/store/apps/details?id=com.radio.fmradio&hl=en&reviewId=gp%3AAOqpTOFWacIVZQ-JHULA86lKu5ZYSNQdIjsM8e6Ph0aj2RWN2aVmoFJFfmJhC91yQEErw6Z0Re3I0LF6k1V_o_Y.

Appendix

Table 1.3: List of attacks and their severity score, based on CVSS v3.

Name	CVSS v3 Vector	Score
SoundBlast	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:L/A:L	4.4
Fork bomb	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H	5.9
Intent storm	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H	5.9
Permissionless speed	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3
Permissionless exfiltration	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3
Covert channel	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3

2

AutoNav: Evaluation and Automatization of Web Navigation Policies

Benjamin Eriksson, Andrei Sabelfeld

Web Conference (WWW) 2020

Abstract. Undesired navigation in browsers powers a significant class of attacks on web applications. In a move to mitigate risks associated with undesired navigation, the security community has proposed a standard that gives control to web pages to restrict navigation. The standard draft introduces a new `navigate-to` directive of the Content Security Policy (CSP). The directive is currently being implemented by mainstream browsers. This paper is a first evaluation of `navigate-to`, focusing on security, performance, and automatization of navigation policies. We present new vulnerabilities introduced by the directive into the web ecosystem, opening up for attacks such as probing to detect if users are logged in to other websites or have active shopping carts, bypassing third-party cookie blocking, exfiltrating secrets, as well as leaking browsing history. Unfortunately, the directive triggers vulnerabilities even in websites that do not use the directive in their policies. We identify both specification- and implementation-level vulnerabilities and propose countermeasures to mitigate both. To aid developers in configuring navigation policies, we develop and implement AutoNav[1], an automated black-box mechanism to infer navigation policies. AutoNav leverages the benefits of origin-wide policies in order to improve security without degrading performance. We evaluate the viability of `navigate-to` and AutoNav by an empirical study on Alexa's top 10,000 websites.

1 Introduction

As the power of the web platform grows, attackers increasingly target *client-side vulnerabilities* [3, 9, 12, 16, 18, 37, 39, 43, 50, 56, 57]. Exploiting these vulnerabilities is effective because clients manipulate highly sensitive information, like login credentials, banking, health, and location data, on behalf of the user.

1.1 Motivation

One of the bigger classes of client-side security vulnerabilities on today's web is *cross-site scripting (XSS)* [45]. An XSS vulnerability gives an attacker the power to execute JavaScript code on another website. This can be used to steal user credentials, change the behavior of the application or render the website unusable. A common approach to mitigate this problem is to let servers send extra security policies along with each HTTP response. The web browser will then enforce these policies, for example, by restricting which scripts to allow on the webpage. These security policies have been defined by the web security community as part of *Content Security Policy (CSP)* [63].

Navigation attacks The current CSP standard (level 2 [63]) does not address attacks via *navigation*. Attackers can thus freely redirect users to malicious or inappropriate websites. This type of attack can affect the confidentiality, integrity and availability of the attacked website. For confidentiality, an attacker with injection capabilities can inject the following script to leak the secret cookie.

```
<script>
window.location = "http://evil.com/?c="+document.cookie;
</script>

```

When the script is executed the user will be sent to `http://evil.com`, along with their cookies, potentially allowing the attacker to take over their account. In addition to only stealing the cookie, the attacker could launch a phishing attack by designing `http://evil.com` to look like the attacked

website. Here the user could be asked to supply more confidential information or be forced to download malicious software. The availability of the website is also compromised as every user visiting the page containing the injected script will be sent away. Note that while CSP can block scripts, an attacker could also force the user to perform a navigation by using meta tags as shown below. While not valid HTML, modern browsers will follow meta redirects in the HTML body.

```
<meta http-equiv="refresh"
      content="0;URL='http://evil.com/'" />
```

The `navigate-to` directive To mitigate these problems the World Wide Web Consortium (W3C) has drafted a standard for the new CSP directive `navigate-to` [61]. This directive has already been implemented in Chrome [36] and Firefox [28]. A common motivation for the directive is to increase the security on websites, as well as, give advertising platforms better control over navigations in ads [40]. We illustrate this in two example scenarios: HTML/JavaScript injection and malicious advertisement.

HTML/JavaScript injection Understanding the space of navigation links on a website can improve security thanks to `navigate-to`. By limiting the possible navigations, attackers will not be able to redirect users. A real-world example of where this policy would have helped is a vulnerability on `blockchain.info` [27]. Attackers were able to inject HTML and JavaScript into the search function on the page. This meant that a URL similar to `blockchain.info/?search=<code>`, which appears to point to `blockchain.info`, could redirect the user to another website. This is known as a reflective XSS vulnerability [48], as the code in the URL is reflected onto the page. Although `blockchain.info` used CSP to mitigate XSS, it was still possible to inject HTML code that forces a redirect. With the new directive, the following CSP policy can mitigate this type of attack. This policy blocks any navigation attempt to anything but `self`, i.e. `blockchain.info`.

```
navigate-to 'self'
```

Malicious advertisement Advertisement platform providers benefit from ensuring that users who click on their ads end up on the correct page. This is especially important if the pages where the ads are served are sensitive to inappropriate material, e.g. websites for kids, governments, or highly respected financial websites. Using the new directive, advertisers would be

able to block navigations leading to incorrect ads. The policy is required because even if the target site for the ad is correct when the ad is bought, the website can at a later stage be hacked or misconfigured. Google Ads could, for example, serve the following policy with an ad from `shoes.com`. This would only allow navigation to `https://shoes.com`, blocking both the HTTP version, as well as, possible deep-links to apps like `app://shoes.com`. The `unsafe-allow-redirects` keyword allows for any number of server-side redirections before reaching `shoes.com`.

```
navigate-to  https://shoes.com 'unsafe-allow-redirects'

```

1.2 Research questions

The standardization [35, 61] and implementation [28, 36] efforts for `navigate-to` are well underway. The time is critical to ask questions on the security, performance, and adoptability of the proposed directive, before its adoption starts on the web. (Our analysis at the time of the writing confirms that the landing pages of Alexa's top 10,000 domains are yet to contain `navigate-to` CSP headers). By pursuing these questions, our goal is to deepen understanding of navigation policies and their impact, contribute to the emergence of the new standard, and to utilize our findings for settling the ongoing discussions by the community [29].

Security While there seems to be much to gain from a navigation policy, what is the impact on the security of the entire web ecosystem? For a fully-fledged security evaluation, we seek to uncover both new vulnerabilities and amplifying effects of known vulnerabilities. Our methodology is thus to investigate possibilities of exploiting the directive by a comprehensive range of attackers defined in the security literature [39]: injection [5], gadget [6], web [2] and passive network [25] attackers. Even though these attackers share some capabilities, they each have unique abilities, e.g. reading network traffic or hosting websites, and as such require individual analysis. This brings us to the questions of security: *Does the new policy "break the web"? Does the new policy introduce security vulnerabilities? How can they be mitigated and by whom?*

Automatization Once the new directive is secured, how can we aid its adoption? CSP has been notoriously hard to adopt, introducing insecure policies or broken websites [56, 57]. To help developers use the new directive, and increase both usability and adoptability, we investigate the possibility

of automatically generating navigation policies. Hence, the question: *Can automatic mechanisms be used to help generate the new policy?*

Performance In contrast to CSP directives like `script-src`, intended to whitelist scripts that can be loaded by a webpage, the `navigate-to` directive will whitelist possible navigations. This results in already lengthy response headers becoming even larger, further increasing the overhead of security headers. This brings us to the question of performance: *What are efficient methods for delivering the new policy?*

1.3 Contributions

This paper is a first systematic evaluation of `navigate-to`. Our goal is to both initiate research on navigation security and to affect the emerging standards for navigation policies. We examine the security implications, efficiency, and the possibility of automatic generation of the new `navigate-to` policy.

Security The intricate connections between policies together with the growing complexity of the web results in new mechanisms becoming more challenging to incorporate into the ecosystem. This motivates the need to analyze multiple types of attackers, as well as, reexamining existing mechanisms in combination with new ones. We follow a methodology of examining the effects of `navigate-to` on a comprehensive range of attackers: injection [5], gadget [6], web [2] and passive network [25] attackers. By scrutinizing the full attack surface of the new directive, with respect to different types of attackers, we identify specification- and implementation-level vulnerabilities that can be exploited (Section 3). The vulnerabilities allow attackers to probe other websites to detect if users are logged in or have active shopping carts, bypass blocking mechanisms of third-party cookies, leak browsing history, and open up new methods for exfiltration. This demonstrates that the directive "breaks the web" in the sense of introducing vulnerabilities even in otherwise secure websites that do not use the directive in their policies. We present mitigations to security problems, both for web and policy developers (Section 4).

Automatization Looking ahead when the proposed mitigations are in place, our goal is to aid in the adoption of `navigate-to`. We develop AutoNav, an automatic mechanism for navigation policy inference (Section 5). AutoNav

crawls websites and generates `navigate-to` policies. The goal of this mechanism is to simplify the deployment of the new directive by helping web developers and security engineers to find fitting policies for their websites. To further improve security, AutoNav can also generate origin-wide policies for the new origin policy delivery mechanism that is currently being drafted [59]. This improves security by applying the policy to the entire origin, covering pages that are easy to forget, like error pages. We implement and evaluate the mechanism by an empirical study (Section 6). In our experiments, we craw 100 pages per domain for 10,000 domains. Based on a subset of 80 pages, AutoNav generates a policy for the remaining 20 pages. For 42% of websites, AutoNav generated a policy which fully covered the 20 pages, and at 59% 19 out the 20 pages were covered. Further investigation into the category of websites shows that shopping websites and adult websites are the easiest to cover.

Performance To evaluate the performance impact of the policy we perform an empirical study (Section 6). Based on 10,000 crawled domains from Alexa's top 10,000, the policy will result in an overhead of 215 bytes for each HTTP response. We create simplification strategies to find a balance between security, performance and maintainability. These simplifications convert complicated policies with multiple subdomains to more manageable policies by using wildcards. For example, instead of including all language-specific subdomains from Wikipedia `navigate-to *.wikipedia.com` would be enough. Our simplification algorithm decreased the overhead by between 40% and 47%. Furthermore, we show that the use of an origin policy would result in an overhead of 1904 bytes in total, as opposed to per HTTP response. This is further decreased to 1004 bytes by using our simplification algorithm. A 900 byte reduction might not seem like much, but it can have a big impact on larger websites [21].

2 Background

Setting the background, we present the threat model in terms of relevant attackers. We describe CSP and how it relates to the origin policy. Finally, we explain navigation methods and how they are treated in the `navigate-to` directive.

2.1 Threat model

The main goal of the `navigate-to` directive is to give web developers control over where users can navigate from their website. The assets that need protecting include confidentiality, integrity and availability. Previous research has already shown how confidential information, such as cookies, can be exfiltrated using navigation [65]. While the new directive is a step in the right direction to address data exfiltration, Zalewski [65] points out that control over navigation is not necessarily enough. Attackers could, for example, inject HTML or JavaScript that change documents from private to public on a website like Dropbox. Forced navigation can also be used for phishing attacks by redirecting users to a similar-looking, but attacker-controlled, website.

Modern web browsers support many different methods for navigation, e.g. by clicking on a link, submitting a form, etc. These navigation methods, and the subset that the `navigate-to` directive is intended to apply to, are explained in Sections 2.4 and 2.5.

As mentioned above, we are interested in a comprehensive security evaluation of the impact of the directive on the entire web ecosystem. Hence, our threat model includes four types of attackers from the security literature [39]: injection, gadget, web and network attackers. In practice there is some overlap between the classes, for example, an attacker with *web attacker* capabilities will usually also have *injection attacker* capabilities. However, the best mitigation strategy might be different depending on which specific class we need to defend against. Therefore it is important to study each distinct class of attacker.

Injection attacker The *injection attacker* [5] is able to inject content into a website. A typical example is a user who can post content on a forum. If the user's post contains JavaScript then that code could be executed by other users on the site, in this scenario, with the goal to force a navigation.

Gadget attacker The *gadget attacker* [6] is similar but more powerful as they are allowed to host code, or gadgets, on other websites. A notable example is JQuery which is a JavaScript snippet that is used by many websites. Since JavaScript do not support any isolation, these gadgets run with the same capabilities as other scripts on the website. A malicious gadget could exfiltrate information from the website it is integrated to, modify content on pages or even navigate the user away from the website.

Web attacker The *web attacker* [2] is able to host and configure a full website. This is especially important for advertisers who want to ensure that the landing page does not redirect to anything other than what was specified in the ad.

Passive network attacker A *passive network attacker* [25] can listen in on all the traffic sent from and to a client but can not decrypt HTTPS. If the traffic is not encrypted, the attacker can read passwords and session cookies being sent to the server.

Note that `navigate-to` is not designed to handle network attacks. Yet we pay attention to network attackers in our effort to analyze the impact of the directive on the entire web ecosystem.

2.2 CSP

CSP is intended to mitigate cross-site scripting (XSS) and other code injection attacks. The current version of CSP, level 2, is supported by all major web browsers [26]. Level 3, which includes the new `navigate-to` directive, is being discussed and drafted [61].

CSP protects the users by specifying which resources and scripts are allowed on a page. The web server sends the CSP policies each time a user requests a page. These policies are then enforced by the browser to, among other things, block XSS. The policy below will only allow scripts to be loaded from the current origin, still blocking any injected inline scripts. In addition, the reporting header `Content-Security-Policy-Report-Only` [33] can be used to report policy violations without enforcing them. These reports are sent as POST requests to the server. They can also be detected using `SecurityPolicyViolationEvent` in JavaScript.

```
1 Content-Security-Policy: script-src 'self'
2
```

2.3 Origin policy

Today, CSP headers are sent with every HTTP(S) response, which is a concern for both safety and performance [50]. For security, it is easy to forget the policy on special pages, like error pages [59]. It also harms performance because servers need to repeat the same policy for each response, even if the policy should apply to all. To address this, specifications are being drafted [59], implemented [60], and evaluated [50] to enable *origin-wide policies*, known as *origin policies* [59] or *origin manifests* [50]. Using an origin

policy, the server only needs to include once which policies should apply to the whole origin.

2.4 Navigation

Navigations can be performed in many different ways by browsers, e.g. by clicking on a link, submitting a form or running JavaScript. Navigation methods can be split into two different categories, user-initiated or document-initiated. While navigation is defined in the Fetch [52] and HTML [4] standards, the exact methods available depend on the web browser implementation. We make an effort to summarize the most common methods in Table 2.4. The *Automatic* column shows if the navigation method can be performed automatically. This is true for all JavaScript function and, in case JavaScript is allowed, `<a>` and `<form>` tags. It is worth noting that while a web page cannot read a user's browsing history, it can initiate navigation to go back or forward in the browser history. There are many `.location` functions in JavaScript that can navigate, e.g. `window`, `document`, `parent`, etc. They all use the Location object defined in the HTML standard [4]. Some functions, like `window.navigate`, only works in Internet Explorer [11]. The last column specifies which methods `navgiate-to` affects.

Table 2.4: Navigation methods together with initiator and possibility to automatically navigate.

Method	Initiator	Automatic	Affected
`<a>` tag	Document	With JavaScript	✓
`<form>` tag	Document	With JavaScript	✓
`<meta>` tag	Document	Yes	✓
`<iframe>` tag [51]	Document	Yes	✓
`window.open` [53]	Document	Yes	✓
`*.location` [4]	Document	Yes	✓
`window.navigate` [11]	Document	Yes	✓
Typing the URL	User	No	
History buttons	User & Document	Yes	
Home button	User	No	

2.5 Navigate-to directive

The *navigate-to* directive gives developers the power to control the navigations a document can initiate. Document initiated navigations are discussed

in Section 2.4. This directive makes it harder for attackers to inject code to redirect users from legitimate websites. For example, if an attacker manages to inject links on `disney.com` then Disney's reputation is at stake if links lead to inappropriate websites. To tackle this, Disney could add the following to their CSP policy:

```
navigate-to *.disney.com *.thewaltdisneycompany.com

```

This would instruct the browser to only accept navigations to subdomains of `disney.com` and `thewaltdisneycompany.com`, and block all navigations to other websites. The standard also introduces the new keyword `unsafe-allow-redirects`, which allows any redirects as long as the final destination is allowed by the policy. It is deemed less safe since it does not have full control over all the sites in the redirect chain. However, it is still better than nothing in terms of limiting navigations.

The `navigate-to` directive is currently being standardized by W3C [61] and implemented in Chrome [36] and Firefox [28]. It is available in the current version of Chrome (version 77.0.3865), and other Chromium-based browsers like Edge and Brave, behind a flag that enables experimental features. It is also available in Firefox Nightly (Version 71.0a1) behind a flag [28].

3 Vulnerabilities

This section presents vulnerabilities and security concerns related to the `navigate-to` policy. These vulnerabilities are not navigation attacks, but rather vulnerabilities that become possible due to `navigate-to`. Except for the last vulnerability in Section 3.3.3, where we rather want to show that a small improvement to `navigate-to` can solve an existing problem. The policy introduces new methods for acquiring privacy-sensitive information, circumvention of security mechanism and data exfiltration. All the attacks described in this section have been tested in practice. While some of the vulnerabilities, like the data exfiltration, relies on the existence of other vulnerabilities, like content injection, the `navigate-to` adds a new layer to the attacks. This possibility of combining attacks shows the importance of reexamining existing ones when introducing new mechanisms.

3.1 Methodology

To systematically find vulnerabilities we distinguish vulnerabilities relating to the specification of the `navigate-to` directive and vulnerabilities related

to its implementation. For each category, we divide the investigation of vulnerabilities pertaining to confidentiality, integrity and availability, in accordance with the CIA triad. We draw on our threat model and examine vulnerabilities with respect to injection, web, gadget, and passive network attackers. Finally, we analyze how the directive can be used to circumvent modern countermeasures, such as third-party cookie blocking.

The presentation of the vulnerabilities is ordered by our estimate of their impact, from high to low. Table 2.5 lists the vulnerabilities we discover together with their corresponding attacker model. Interesting to note is that resource probing and Google Search profiling can be exploited to attack websites that themselves do not use the `navigate-to` directive. This results in previously security websites becoming insecure.

Table 2.5: The uncovered vulnerabilities together with corresponding attacker model.

Vulnerability / Attacker	Injection	Web	Gadget	Passive network
Resource probing		✓		
Google Search profiling		✓		
Third-party cookie bypass		✓		
History sniffing		✓		
Data exfiltration	✓		✓	
Ads leaking data				✓

3.2 Specification

The following vulnerabilities are present in the specification. This means that any browser following the specification correctly will be vulnerable.

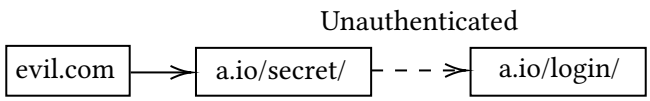

Figure 2.4: A user visiting `evil.com` will be navigated to `a.io/secret/`. If they are not logged in, they are further redirected to `a.io/log in/`.

3.2.1 Resource probing

In cases where web applications redirect based on sensitive resources, these resources could be probed. For example, probing for the existence of Dropbox

files. The probing attacks in this section are deterministic, as opposed to other attacks that rely on timings [55]. The attacks are also general and could potentially be used on any website, not solely on advertiser platforms such as the attack presented by Venkatadri et al. [54].

A malicious website, i.e. a web attacker, can navigate a user to `dropbox.com/preview/wallet.txt` to detect if a user has a file named `wallet.txt`. If no such file exists then the user is redirected to `dropbox.com/home/wallet.txt`, making it possible to craft a policy which blocks `/preview/` but not the redirection to `/home/`, like the following. Note here that we only use path-sensitivity to block `/preview/`. If we are redirected, then path-sensitivity is no longer available and we only have to allow `dropbox.com`. The main difference compared to previous work on CSP redirections [23] is that we only need path-sensitivity for the first request, not the redirects.

```
1 navigate-to 'unsafe-allow-redirects' https://www.dropbox.com/
     not_preview/;
2
```

By utilising invisible iframes multiple files can be checked in parallel, without the user being navigated away from the malicious website.

One specific application of resource probing that has been researched before is login detection. Previous methods [20, 32] relies on third-party cookies, which can be blocked by the user or by the proposed default SameSite policy [62]. Instead, note that a navigation to `facebook.com/settings/` will redirect the user to the login page, `facebook.com/login.php`, if they are not authenticated, similar to Figure 2.4. By allowing only one of these URLs in the policy, the attacker can differentiate between a successful navigation and a blocked one. This feature makes our method more powerful and general.

We have also found that on some E-commerce websites it is possible to detect if a customer has anything in their shopping cart. This is because navigating directly to the shopping cart or checkout page sometimes redirects the user depending on the content of the cart. PrestaShop, which is an E-commerce platform used on hundreds of thousands of websites [8], does exactly this. By visiting `example.com/en/order` a user will be redirected to `example.com/en/cart`, assuming `example.com` uses PrestaShop.

Some of the probing attacks can leak more data if they are done in an active fashion. The PrestaShop attack can be improved to, in theory, enumerate the full cart. This is due to a Cross-Site Request Forgery (CSRF) [47] vulnerability in PrestaShop, currently being disclosed, which allows an attacker to add and remove items. Using this method an attacker can repeatedly remove

items and then check if the cart is empty.

These are only a few examples we have found where redirects are based on sensitive data. We believe that many more such redirects currently exists on the web. Furthermore, navigations can bypass `lax` SameSite cookies, making the attack possible on sites where previous CSRF attacks were not possible.

3.2.2 Google Search profiling

Google Search relies on *personalized search* [19], meaning that the results of a search query are based on the users' previous interactions with Google. A recent study [24] shows that users are put into so-called "filter bubbles" by Google, resulting in varying result when searching for political terms such as "gun control" or "immigration". A web attacker can craft a malicious website which uses the `navigate-to` directive together with Google's *I'm feeling lucky* function to extract top results from visitors. This type of extraction attack is called cross-site search and has previously been successfully mounted against Gmail and other websites [17]. The main difference is that previous methods have relied on timing, whereas our method is fully deterministic. Castelluccia et al. [10] were also able to infer sensitive information about users based on Google Searches. However, their approach required network attacker capabilities and assumed the traffic was unencrypted, which is not the case anymore. Our attack can be mounted by anyone with the capability to set up a website.

The attacker can then use these top results from Google to infer these filter bubbles. Using the URL `https://www.google.com/search?q=QUERY&btnI`, Google will automatically redirect the user the top result for term "QUERY". Therefore the *I'm feeling lucky* function acts as an open redirector, which is something both OWASP [46] and Google [34] themselves warn about. It is well known that Google has this problem but so far they choose to accept the risk [1]. However, the `navigate-to` directive adds a new dimension to the problem as it enables attackers to infer data about users.

To exploit this the attacker can specify a report-only policy that only allows `google.com`, as shown below. The redirect will violate the policy and the browser will dutifully report which domain was in violation to the malicious website. The attacker can iteratively update the query to get more results. Assuming searching for "news" would return `news.com`, then the next query would be "news -site:news.com", which excludes `news.com` and perhaps returns `reports.com` instead. Another attack vector would be other search engines using this approach to directly copy personalized search results from Google, similar to what Bing did [41].

```
Content-Security-Policy-Report-Only: navigate-to 'unsafe-allow-
    redirects' google.com

```

3.2.3 Third-party cookie bypass

A cookie is a piece of data that websites can save locally on users' machines. [31] Depending on how the cookie is acquired, it will either be considered a first-party cookie or a third-party cookie. A navigation will result in first-party cookies while image request and similar results in third-party cookies.

Third-party cookies are useful for advertisers [14] as it allows them to use small tracking pixels [15] for tracking users. Modern browsers allow users to block third-party cookies or do it by default [42].

Previous work has demonstrated how Cookie Synchronization [7, 38] can be used by ad platforms to effectively break the same-origin policy. Privacy-aware users can mitigate this by blocking third-party cookies altogether. However, the `navigate-to` directive introduces a new method for advertisers to circumvent this by using navigations. As it requires control over the CSP headers, web attackers are the main threat. Figure 2.5 shows a user visiting `a.io`, then being forcibly navigated to `track.com` and acquiring a first-party cookie. Using the following policy, the redirection will be blocked, making the attack unnoticeable to the user.

```
navigate-to 'unsafe-allow-redirects'

```

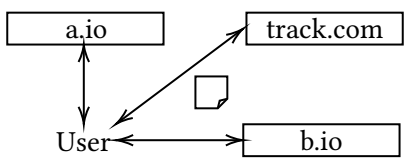

Figure 2.5: When a user visits `a.io` or `b.io`, they can force the user to obtain first-party cookies from `track.com`.

3.3 Implementation

The following vulnerabilities are due to implementation decisions. We focus on Chrome's [36] and Firefox's [28] implementations of `navigate-to`,

3.3.1 History sniffing

The `navigate-to` policy can, in some cases, be exploited by a web attacker with a malicious website to probe which websites a user has visited. The attack uses the fact that websites using HSTS force the browser to remember and upgrade insecure connections. Previous methods exploiting this have relied on timing attacks which are now mitigated [64].

Using `navigate-to`, a malicious website can make a POST request to another site which uses HSTS but is not preloaded. If the site redirects based on the POST data then the attacker might be able to detect if a user has visited the site before. This is possible because if the user has visited the site before it will result in an internal redirect (HTTP 307), which keeps the POST data. Otherwise, the server will redirect (HTTP 301/302), which drops the POST data. If the server specifically performs a 307 redirect then the attack will not work. By crafting a CSP that does not allow the redirect, the attacker can differentiate between the two cases, denoted X and Y in Figure 2.6. This is, for example, possible using the login function on the popular social media website VK.

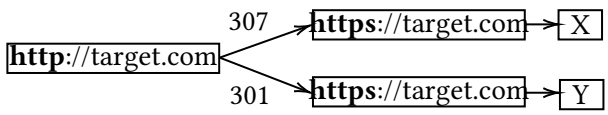

Figure 2.6: If `target.com` uses HSTS, and the user has visited the site before, then the browser will automatically upgrade the connection to HTTPS using a 307 redirect instead of a server side 301.

3.3.2 Data exfiltration and communication

Previous research has shown that data exfiltration is possible in the face of CSP [49]. The usage of forms and links to exfiltrate data has also been studied [65]. However, the `navigate-to` policy introduces an improved method for exfiltration, and two-way communication, based on JavaScript together with navigation. This works in Chrome, but not Firefox, as Chrome does not unload the page for `navigate-to` violations.

Consider a website using `connect-src 'none'` and `frame-src 'none'` to limit external loads as much as possible. The `connect-src` directive protects against some exfiltration methods including XHR, fetch and `<a ping>`, while `frame-src` will block exfiltration to iframes. Assume the website uses `unsafe-allow-redirects` followed by a list of allowed URLs. Note here that we show that *unsafe* has implications beyond the scope of restricting

navigation. An attacker capable of injecting JavaScript, i.e. either an injection or gadget attacker, can now use `window.location`, as shown in the listing below, to exfiltrate arbitrary data. Each navigation request will exfiltrate data, then be blocked by the policy, as the attacker can choose a website outside the CSP whitelist. Furthermore, by adding a `SecurityPolicyViolation` event listener the attacker can inspect the blocked URI in the violation. To send a response, `evil.com` would redirect the request to a subdomain like `<msg>.evil.com`.

```
function exfiltrate (data) {
  window.location = "http://evil.com/?d=" + data;
}

```

The main difference between not using `navigate-to` and using the policy described is that by blocking the navigations, the control is returned to the attacker, allowing for further stealth exfiltration and communication.

3.3.3 Ads leaking data

We have found that ads served over HTTPS can still leak the final landing page to a passive network attacker if an ad in the redirection chain is unencrypted. While network-level eavesdropping is outside of CSP's threat model, the `navigate-to` directive presents a great opportunity to fix this problem. The problem stems from the fact that when a user clicks on an ad they can be channeled through multiple tracking websites. Listing 2.1 shows a chain where the user is redirected to three different websites before the landing page. We performed a small empirical study using the same dataset as in Section 6. We extracted all iframes and compared their source URL to a list of known advertisement platforms, e.g. DoubleClick. If the URL matched we followed it and recorded the redirects. This resulted in 24650 unique ads, of which 26.7% have a website between the advertisement platform and the landing page. This highlights the need for advertisement platforms to consider potential redirects from tracking websites and further motivates the need for the `navigate-to` directive.

```
https://www.googleadservices.com/...
http://www.kqzyfj.com/...
http://cj.dotomi.com/...
http://www.emjcd.com/...
https://<landing page>/...

```

Listing 2.1: Example of an ad chain containing three different unencrypted domains between the encrypted ad platform and landing page.

As can be seen in Listing 2.1, both the first and last websites use HTTPS but there exist sites between that are unencrypted. This is very hard for a user to detect as both the ad and the landing page seems secure. The problem with having HTTP in the chain is that an eavesdropper can follow the request and find the landing page. Our empirical experiments show 10.6% of the ads follow this pattern. As ads become more personal this becomes a privacy concern. Advertisements related to economic status or specific diseases might be leaked without the user's knowledge.

4 Countermeasures

This section presents countermeasures to the vulnerabilities in 3. The countermeasures cover the specification, mitigations for web developers, as well as, implementation improvements in web browsers. Similarly to the vulnerabilities in Section 3 we distinguish specification- and implementation-level countermeasures.

4.1 Specification

4.1.1 Resource probing

Previous login detection methods have forced web developers to rewrite their applications to avoid special types of redirections. As mentioned in [13], Google added an extra regex check to make sure the redirection did not lead to resources that could be loaded cross-origin, e.g. "jpg", "js" and "ico".

The `navigate-to` policy circumvents this by being able to block and report different paths in the URL, i.e. it is possible to block `example.com/settings/` and allow `example.com/login/`. In this case, if `/settings/` redirects to `/login/` for unauthenticated users, then the CSP report log can be inspected to discern between authenticated and unauthenticated users.

To fix this, path precision could be removed from the policy. If an origin as a whole can not be trusted, it seems to add little security to trust certain paths on the origin. Since these vulnerabilities affect websites that do not use `navigate-to`, we also present countermeasures web developers can implement. We recommend avoiding redirection based on secrets. Instead, by showing an error page or rendering the login form on the same page the website is guaranteed to not leak any data, as there will be no redirections. If redirection is necessary, encoding paths in GET parameters, e.g. from `example.com/files/` to `example.com/?path=/files/`, also mitigates the problem.

4.1.2 Google Search profiling

For vulnerabilities like Google Search profiling, as presented in Section 3.2.2, the key countermeasure is to avoid open redirects [46]. One possible way for Google to accomplish this without removing the *I'm feeling lucky* function is to use a CSRF token [47].

4.1.3 Third-party cookie bypass

The navigation path through the redirection chain can depend on the user's cookies. For this reason, it is not possible to block cookies while checking if the navigation is allowed. Instead, we suggest that cookies attained during the check are temporarily sandboxed and then removed if the navigation is blocked.

4.2 Implementation

4.2.1 History sniffing

Privacy problems related to HTTP Strict Transport Security (HSTS) [22] has been researched before [44]. However, they focused on tracking mechanisms similar to cookies but harder to remove.

The solution is to ensure that an attacker can not differentiate between the paths in Figure 2.6. Again, it becomes the web developers responsibility to either use an internal redirect or not redirect on post data.

4.2.2 Data exfiltration

What makes this attack extra powerful is its ability to regain execution control after the navigation fails. It is not specified what should happen when the `navigate-to` policy blocks a navigation attempt. Currently, Chrome seems to simulate a 204 response [58], resulting in the continuation of the script, and the possibility to exfiltrate more data. Firefox, on the other hand, uses a full-page error that unloads the original document. By using this strategy the script will stop executing, blocking further exfiltration. The attack can also be mitigated by avoiding `unsafe-allow-redirects`, as this will block the exfiltration during the pre-navigation check.

4.2.3 Ads leaking data

The `navigate-to` directive could block redirect chains which contain HTTP websites. Currently, the policy `navigate-to https:` allows navigation to any website using HTTPS. However, combined with

`unsafe-allow-redirects` HTTP is allowed in the chain, as long as the landing page is HTTPS. One solution is to add a value `unsafe-allow-https-redirects` which would only allow redirection by HTTPS. A more general solution is to split the policy into `navigate-to` and `navigate-by`, where the latter would apply as long as the request is redirected. When no redirect is received, the landing page is checked against the `navigate-to` policy. By using this method, the following policy would allow any HTTPS redirections which lead to `https://example.com`.

```
navigate-to https://example.com
navigate-by https:

```

5 AutoNav

We present AutoNav, an automatic mechanism to aid web developers in inferring policies for their websites. The mechanism crawls the website and creates a map of where pages can navigate. This mapping is used to generate and simplify the policies. AutoNav can generate both per-page policies, where each page on a website gets its own policy, and origin-wide policies [59].

5.1 Inference

We use a key-value map from the crawler to infer the policies. The page is used as a key, and a list of all possible navigations from the page is used as a value. Listing 2.2 shows an example.

```
{
  "example.com/a.html": [facebook.com, google.com],
  "example.com/b.html": [twitter.com, google.com]
}

```

Listing 2.2: Example of a key-value map generate from crawling two pages on `example.com`

Using the key-value map, AutoNav can generate separate policies for each page on the website. This is shown in Listing 2.3. AutoNav can also generate an origin-wide policy based on the union of all the URLs, as shown in Listing 2.4. These policies are then simplified, using the method described in Section 5.2, to reduce the size and improve maintainability.

```
{
  "a.html": "navigate-to facebook.com google.com",
```

```
  "b.html": "navigate-to twitter.com google.com"
}

```

Listing 2.3: Per-page policies generated from Listing 2.2.

```
{
  "*": "navigate-to facebook.com twitter.com google.com"
}

```

Listing 2.4: Origin-wide policy generated from Listing 2.2.

5.2 Policy generation

The navigation policy is a whitelist of URLs that the user is allowed to navigate to. In the most secure setting, the policy should contain the full URLs to each allowed target. While secure, this creates big and hard to maintain lists of URLs requiring much bandwidth. Take Wikipedia for example, their policy could consist of all subdomains like `en.wikipedia.org`, `es.wikipedia.org`, etc. for each language. A more compact policy is `*.wikipedia.org`. This simplification results in both less data being transmitted and a more maintainable policy, however, it does decrease security as it also allows `evil.wikipedia.org`.

AutoNav supplies developers with best-effort policies that aim to help them harden their websites. Using our parameterized simplification algorithm, developers get a slider style method for finding a trade-off between maintainability, performance and security. The simplification algorithm looks for evidence that all subdomains are trusted. The two sources used are the number of URLs that point to the subdomains (denoted t_1) and the number of subdomains that are pointed to (denoted t_2). The motivation for t_2 is that even if multiple links are found to `a.example.com` it does not imply that `b.example.com` should be allowed. Similarly, t_1 is motivated by the notion that the more URLs that point to `*.example.com`, the more it can be trusted. Figure 2.7 shows the tree representation of 10 URLs pointing to `example.com` and its subdomains. u_i in the figure represents one URL, e.g. u_7 points to a resource on `test.b.www.example.com`. Furthermore, the figure also includes tuples of the threshold values (t_1, t_2). Figure 2.8 shows the tree after simplification using a threshold of $(2, 2)$. Using this method the policy will only contain 3 entries instead of 7 entries.

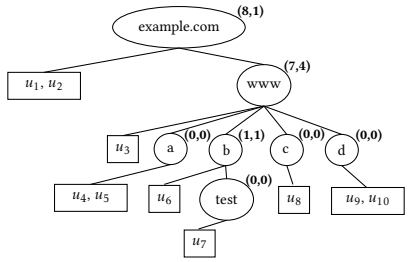

Figure 2.7: Tree representation of 10 URLs collected from `example.com` and its subdomains. The tuples corresponds to the (t_1, t_2) thresholds.

Figure 2.9 shows the result from crawling five pages on `ebay.com` and generating a policy. The crawler was only supplied with the start page and then found the other four using the crawling algorithm from Section 5.3. The five pages crawled are shown in the middle of the figure in grey with integer labels. The arrows from these nodes indicate that a possible navigation was found between two nodes. The colors correspond to which part of the policy covers the navigation. As shown, `*.ebay.com` covers a lot of the subdomains, thus they all share the same color. Using the figure, an origin policy could be generated by taking the union of all the colors.

This method of generating policies guarantees that the functionality of the website will remain intact. This is because, if a domain is in the list of possible navigations, then it will be included in the policy. Similar to other policies, the generated policy would need to be recalculated if the website was updated to include new possible navigations. For security, the method guarantees that if a domain is not in the list, then it will not be added to the policy. However, subdomains of domains in the list can be added to the policy.

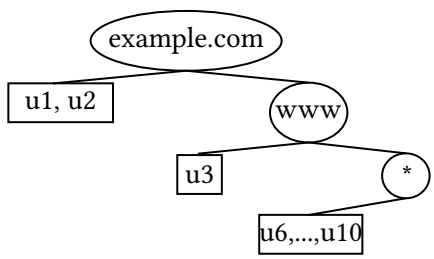

Figure 2.8: Result of applying the simplification algorithm, using a treshold of $(2, 2)$, to the tree in Figure 2.7. Resulting in the following policy, `navigate-to example.com www.example.com *.www.example.com`.

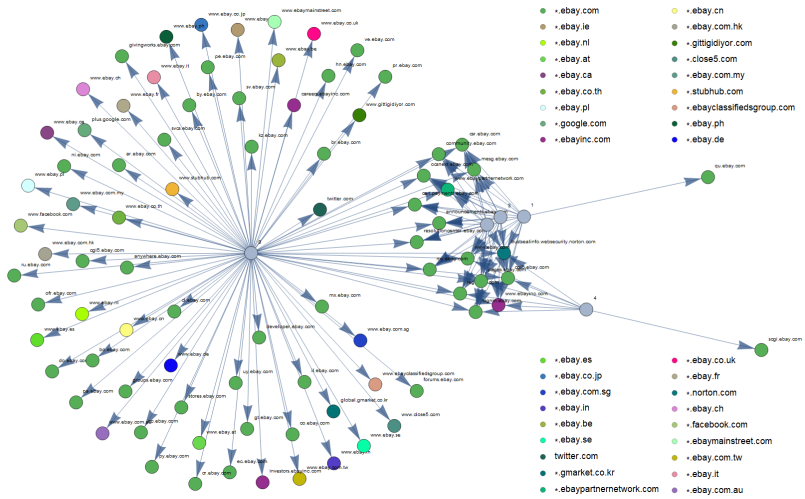

Figure 2.9: Generated policies for `ebay.com`. The nodes with outwards pointing arrows are the five pages that we crawled. All the other nodes correspond to a possible navigation. The color indicates which part of the policy covers the navigation.

5.3 Crawling

Our implementation of AutoNav uses selenium with a Chrome instance to crawl the pages on a website. By only supplying AutoNav with the first URL it will automatically collect and crawl new URLs that it finds. When a URL from the same website is found it is added to a set of unvisited URLs, from which the next URL is picked. For each page on a domain, all the JavaScript is executed, then the URLs from links and forms are saved. When the crawling session is over, the inference method described in Section 5.1 is used to generate the policy.

5.4 Limitations

We did not take special care to crawl behind the login. However, it is trivial for a site owner to add a session cookie to the crawler. The more pages AutoNav can crawl the more the policy will cover. Crawling too few pages will result in an incomplete yet secure policy. The policy is secure because AutoNav will never add a domain to the policy that has not been seen.

We use static links to infer the policies, which will miss possible redirections. While not a security concern, we would produce more precise policies if each link was followed dynamically and the redirections recorded.

User-agent sniffing is a common problem for crawling studies. Since the AutoNav is designed for developers we think they can manually add entries like `languages.mysite.org` and use the AutoNav to detect everything else.

6 Empirical Study

This section presents an empirical study to evaluate the performance impact of the new directive, as well as, how different delivery methods and simplifications can reduce the impact. Next, we evaluate AutoNav in how well automatically generated policies based on a subset of the website cover the full website.

To test how the new `navigate-to` policy will function on common websites we utilize AutoNav in a crawling experiment. For calculating the performance impact in Section 6.1, we use Alexa's top 10,000 websites. For evaluating AutoNav itself we use Alexa's top 14,000, ensuring we have 10,000 domains which all have more than 100 pages each.

6.1 Policy tradeoffs

This section presents the performance tradeoffs between per-page and origin-wide policies together with the delivery methods of HTTP headers and origin policy.

The costs in Table 2.6 are based on a user visiting n pages on a website, thus the cost of HTTP headers need aggregation over all pages, i.e. $\sum_{i \leq n}$. The cost of sending a single CSP policy depends on the number of URLs it contains. We defined the cost of the policy based on the set of URLs, i.e. $|U_i|$, U_i being the set of URLs on page i. Further, we can define a set of all URLs as the union of the sets of URLs on each page as $\bigcup_{j \leq n} U_j$, with corresponding

Empirical performance Based on the 10,000 crawled domains, a per-page policy, without any simplifications, would increase the header size with 215 bytes, per response. A more maintainable origin-wide policy results in a size increase to 1904 bytes. This cost can be decreased by using the origin policy for delivery, in which case the user only downloads the policy once. Note, as shown in Table 2.6, that an origin policy outperforms a per-page policy after only 9 responses. While per-page policies might seem better, they are difficult to use since they require knowledge about the content on each page. As such, some website, e.g. Facebook, use origin-wide policies, motivating the need for an origin policy delivery method.

In addition to the comparison between per-page and origin policy, we also evaluated the cost benefits of using our policy simplification algorithm. Using maximum simplifications, i.e. $t_1 = 1, t_2 = 1$, the average size of the origin wide policy decreases from 1904 to 1004 bytes, a decrease of 47%. Similarly, the per-page policy decreases from 215 bytes to 129 bytes, which is a 40% decrease. For some websites, the benefit of simplification is much greater. In particular, this is the case when websites allow navigation to numerous subdomains. For example, `spravker.ru` would require a 20438 byte origin policy without simplification, but only 61 bytes after simplification. The big difference stems from the fact that `spravker.ru` have 954 subdomains.

Table 2.6: Empirical costs for different policy models.

	HTTP	**Origin Policy**
Per-page	$\sum_{i \leq n} 215$	-
Origin-wide	$\sum_{i \leq n} 1904$	1904

We also performed a more in-depth analysis of three websites, `ebay.com`, `wikipedia.org` and `stackexchange.com`, to see how the threshold affect performance. Fixing t_1 to 0, we only focus on the number of subdomains when deciding if wildcards should be used. Figure 2.10 shows these domains as solid lines, together with the corresponding costs for their origin policies. As can be noted, after the t_2 threshold reaches 280 subdomains Wikipedia can no longer use the wildcard and the policy quickly increases in size. By increasing t_1 to 1000, more URLs are required before simplifications can take place. As can be seen in the dashed lines in Figure 2.10, the crawled data from Wikipedia did not contain enough URLs to the same domain for a simplification. This would be the desired behavior if Wikipedia required high assurance before introducing wildcards.

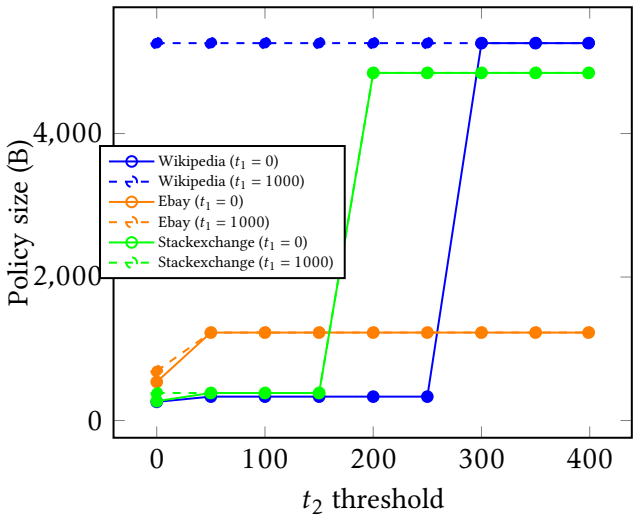

Figure 2.10: Cost of origin policy for different domains and simplification thresholds. The y-axis shows policy size in bytes and the x-axis shows the t_2 threshold. The legend shows the t_1 threshold

6.2 Coverage

While full coverage may be desirable, the goal of AutoNav is to help even if the coverage is not complete, by providing a useful baseline policy for developers to build on.

Our coverage was generated similarly to the method used in CSPAutoGen [37]. We generate the policy based on a training set of 80% of the pages on a domain and then test how well they match the other 20%. We define U as the set of URLs in the training set. For the n pages in the validation set, we check if URLs on the page are covered, i.e. $p_i \subseteq U$, where p_i is the set of all the URLs on page p_i. Finally the coverage of a website is calculated as: $c = \frac{|\{p_i : p_i \subseteq U\}|}{n}$

Using this formula, c is calculated for all the websites that were crawled. In total 42% of all the websites were fully covered and for 59% of the websites 95% or more were covered. Note that these results come from only crawling 100 pages, deeper crawls can greatly increase this coverage.

7 Related work

Automatic methods for generating CSP policies have been studied before [12, 16, 18, 37]. deDacota by Doupe et al. [12] performs static analysis of ASP.NET

code in order to separate JavaScript code from data. After the JavaScript has been separated into files, a CSP policy was generated for the file. AutoCSP by Fazzini et al. [16] takes a similar approach by analysing server-side code, PHP in this case. However, AutoCSP uses dynamic taint tracking instead of static analysis, allowing it to create policies for inline JavaScript events and CSS code. While both AutoCSP and deDacota were successful, they required access to the source code of the application. In contrast, AutoNav uses a black-box approach which removes the need for the source code. Furthermore, the aforementioned methods focus on JavaScript and CSS, while our focus is on navigation and URLs. In addition, static analysis of source code will miss many URLs since modern web applications, like WordPress, store content in the database and not in the code.

In addition, research has been done on generating policies without access to the source code. Golubovic's autoCSP [18] method utilizes a reverse proxy and the report function in CSP to run an application in *learning mode*. In this mode, the tool externalizes inline code and generates policies for the scripts that should be allowed. A drawback is that autoCSP requires manual navigation through the application to ensure all scripts are triggered. While this works well for scripts, it becomes challenging when all possible links need to be navigated. A similar approach based on the report function in CSP was utilized by King's Firefox extension Laboratory [30]. Laboratory is impressive as it enables users to record and generate CSP policies in real-time while visiting a website. Starting with a strict policy, it gradually weakens it as violation reports are received. While this method could be extended to include navigations, it would require the user to initiate all possible navigations on each page. Instead of relying on the reporting functionality, our method uses a combination of static and dynamic analysis to record the navigations a document can initiate. By doing this we avoid the problem of having to initiate all navigations to generate a report. We also improve on the manual aspect of traversing a website by implementing an automatic crawler, as suggested by Golubovic, in future works.

CSPAutoGen Pan et al. [37] is also intended to automatize CSP generation. CSPAutoGen uses a crawler to analyze websites and try to infer which scripts should be allowed. Similar scripts are also generalized into abstract syntax trees, based on how many similar scripts are found. Once a policy has been inferred, CSPAutoGen functions as a proxy between the client and the server. This enables CSPAutoGen to rewrite requests and responses in real-time, without needing any CSP configurations on the website. This is a great feature when a server needs to be secured without any direct modification. While a similar approach could be used for URLs and navigation, our goal is

to generate CSP policies that can be used by the server directly.

In addition to policy generation, we benefit from origin-wide policies [59]. Similarly to the work on evaluating general origin-wide policies by Van Acker et al. [50], our results also indicate that an origin-wide policy provides additional security without degrading performance.

8 Conclusion

Security We have performed a security analysis of the emerging CSP directive `navigate-to`. Our findings show that the current specification and implementations introduce new vulnerabilities. The vulnerabilities include methods for resource probing, login detection, circumventing blockage of third-party cookies, as well as, history enumeration. To mitigate these problems we propose countermeasures to both the specification and implementation of the directive. We demonstrate that the directive triggers vulnerabilities even in websites that do not use the directive in their policies. Thus, we also propose countermeasures web developers can make to their applications in order to mitigate the possibilities of being exploited.

Automatization We have evaluated the possibility of automatically generating policies to help developers adopt the policy, we created AutoNav. AutoNav uses a black-box approach to crawl websites and generate CSP policies that can be directly applied to the website. Our results show that in total 42% of all the websites were fully covered and for 59% of the websites 95% or more were covered. We further simplify the process by identifying categories of websites which the policy better fits. Our research shows that shopping and adult websites are best covered. These websites have a high incentive to keep the users on their site, with the exception of linking to sponsors or partners, which AutoNav's policies cover.

Performance To analyze the performance of `navigate-to` we have conducted an empirical study of Alexa's top 10,000 websites. For each website, we have crawled 100 pages and based on theses generated policies. We show that on average this directive would increase the header size by 215 bytes per request. However, using our simplification algorithm we produce more maintainable policies which were also 40% smaller on average. Our results indicate that using an origin policy would require a one time cost of 1904 bytes, or 1004 using simplifications, as opposed to 215 bytes per request. Thus we show that the performance hit from the increased security can be efficiently mitigated by adopting an origin policy with suitable simplifications.

Coordinated disclosure We are in the process of disclosing the discovered vulnerabilities to the affected vendors, including Google where both Chrome's implementation of `navigate-to` directive and the Google Search website are affected. Based on our recommendations Firefox chose to harden their implementation against exfiltration attacks, as explained in Section 4.2.2.

Acknowledgements Thanks are due to Mike West, Christoph Kerschbaumer, and Daniel Hausknecht for helpful discussions on the topic of `navigate-to`. This work was partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

Bibliography

[1] F. Aboukhadijeh. Is google an open redirector?, 2011.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF)*, pages 290–304. IEEE, 2010.

[3] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 377–392, 2018.

[4] Apple, Google, Mozilla, Microsoft. Html living standard, 2019.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88. ACM, 2008.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*, 52(6):83–91, 2009.

[7] M. A. Bashir, S. Arshad, W. Robertson, and C. Wilson. Tracing information flows between ad exchanges using retargeted ads. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 481–496, 2016.

[8] BuiltWith Pty Ltd. Prestashop usage statistics, 2018.

[9] S. Calzavara, A. Rabitti, and M. Bugliesi. Semantics-based analysis of content security policy deployment. *ACM Trans. Web*, 12(2), Jan. 2018.

[10] C. Castelluccia, E. De Cristofaro, and D. Perito. Private information disclosure from web searches. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 38–55. Springer, 2010.

[11] Dottoro. navigate method (window), 2019.

[12] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna. dedacota: toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1205–1216, New York, NY, USA, 2013. ACM.

[13] A. Elsobky. Novel techniques for user deanonymization attacks, 2016.

[14] S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1388–1401. ACM, 2016.

[15] Facebook Inc. Use facebook pixel, 2018.

[16] M. Fazzini, P. Saxena, and A. Orso. Autocsp: Automatically retrofitting csp to web applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 336–346, May 2015.

[17] N. Gelernter and A. Herzberg. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1394–1405. ACM, 2015.

[18] N. Golubovic. autocsp - csp-injecting reverse http proxy, 2013.

[19] Google Inc. Personalized search for everyone, 2009.

[20] G. G. Gulyás, D. F. Somé, N. Bielova, and C. Castelluccia. To extend or not to extend: on the uniqueness of browser extensions and web logins. *CoRR*, abs/1808.07359, 2018.

[21] S. Helme. Optimising twitter's csp header, Jan 2018.

[22] J. Hodges, C. Jackson, and A. Barth. Http strict transport security (hsts). RFC 6797, RFC Editor, November 2012.

[23] E. Homakov. Using content-security-policy for evil, Jan 2014.

[24] D. Inc. Measuring the "filter bubble": How google is influencing what you click, 2018.

[25] C. Jackson and A. Barth. Forcehttps: protecting high-security web sites from network attacks. In *Proceedings of the 17th international conference on World Wide Web*, pages 525–534. ACM, 2008.

[26] J. Karahalis. Content security policy (csp), 2018.

[27] K. Karlsson. 179426 reflected xss on blockchain.info, 2017.

[28] C. Kerschbaumer. 1529068 - implement csp 'navigate-to' directive, February 2018.

[29] A. King. Allow navigation to only whitelisted urls via navigate-to 125, 2016.

[30] A. King. april/laboratory, 2018.

[31] D. Kristol and L. Montulli. Http state management mechanism. RFC 2965, RFC Editor, October 2000.

[32] R. Linus. Your social media fingerprint, 2017.

[33] J. Medley. Content-security-policy-report-only, 2018.

[34] J. Morrison. Open redirect urls: Is your site being abused?, 2009.

[35] A. Paicu. CSP 'navigate-to' directive: Consensus & Standardization, 2018.

[36] A. Paicu. Implement the 'navigation-to' directive, 2018.

[37] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 653–665, New York, NY, USA, 2016. ACM.

[38] P. Papadopoulos, N. Kourtellis, and E. Markatos. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *The World Wide Web Conference*, pages 1432–1442. ACM, 2019.

[39] P. D. Ryck, L. Desmet, F. Piessens, and M. Johns. *Primer on Client-Side Web Security*. Springer, 2014.

[40] G. B. Security. Communication with google's blink security team, November 2018.

[41] R. Singel. Google catches bing copying; microsoft says 'so what?, February 2011.

[42] N. Statt. Advertisers are furious with apple for new tracking restrictions in safari 11, 2017.

[43] M. Steffens, C. Rossow, M. Johns, and B. Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *NDSS*, 2019.

[44] M. Stockley. Anatomy of a browser dilemma - how hsts supercookies make you choose between privacy or security, 2015.

[45] The OWASP Foundation. Owasp top 10 - 2017, 2017.

[46] The OWASP Foundation. Unvalidated redirects and forwards cheat sheet, 2017.

[47] The OWASP Foundation. Cross-site request forgery (csrf), 2018.

[48] The OWASP Foundation. Cross-site scripting (xss), 2018.

[49] S. Van Acker, D. Hausknecht, and A. Sabelfeld. Data exfiltration in the face of csp. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 853–864. ACM, 2016.

[50] S. Van Acker, D. Hausknecht, and A. Sabelfeld. Raising the bar: Evaluating origin-wide security manifests. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 342–354, 2018.

[51] A. van Kesteren. Fetch standard, February 2018.

[52] A. van Kesteren. Fetch living standard, 2019.

[53] M. Vasigh. Window.open(), 2018.

[54] G. Venkatadri, A. Andreou, Y. Liu, A. Mislove, K. P. Gummadi, P. Loiseau, and O. Goga. Privacy risks with facebook's pii-based targeting: Auditing a data broker's advertising interface. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 89–107, May 2018.

[55] T. Watanabe, E. Shioji, M. Akiyama, K. Sasaoka, T. Yagi, and T. Mori. User blocking considered harmful? an attacker-controllable side channel to identify social accounts. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 323–337. IEEE, 2018.

[56] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1376–1387. ACM, 2016.

[57] M. Weissbacher, T. Lauinger, and W. Robertson. Why is csp failing? trends and challenges in csp adoption. In *International Workshop on Recent Advances in Intrusion Detection*, pages 212–233. Springer, 2014.

[58] M. West. Allow navigation to only whitelisted urls via navigate-to 125, 2016.

[59] M. West. Origin policy, 2017.

[60] M. West. Origin policy, 2017.

[61] M. West. Content security policy level 3, 2018.

[62] M. West. Incrementally better cookies, May 2019.

[63] M. West, A. Barth, and D. Veditz. Content security policy level 2, 2016.

[64] Yan (bcrypt). @bcrypt -paper2- advanced browser fingerprinting - toorcon 2015, November 2015.

[65] M. Zalewski. Postcards from the post-xss world, 2011.

3

Black Widow: black-box Data-driven Web Scanning

Benjamin Eriksson, Giancarlo Pellegrino, Andrei Sabelfeld

IEEE Symposium on Security and Privacy (S&P) 2021

Abstract. Modern web applications are an integral part of our digital lives. As we put more trust in web applications, the need for security increases. At the same time, detecting vulnerabilities in web applications has become increasingly hard, due to the complexity, dynamism, and reliance on third-party components. Blackbox vulnerability scanning is especially challenging because (i) for deep penetration of web applications scanners need to exercise such browsing behavior as user interaction and asynchrony, and (ii) for detection of nontrivial injection attacks, such as stored cross-site scripting (XSS), scanners need to discover inter-page data dependencies.

This paper illuminates key challenges for crawling and scanning the modern web. Based on these challenges we identify three core pillars for deep crawling and scanning: navigation modeling, traversing, and tracking inter-state dependencies. While prior efforts are largely limited to the separate pillars, we suggest an approach that leverages all three. We develop Black Widow, a blackbox data-driven approach to web crawling and scanning. We demonstrate the effectiveness of the crawling by code coverage improvements ranging from 63% to 280% compared to other crawlers across all applications. Further, we demonstrate the effectiveness of the web vulnerability scanning by featuring no false positives and finding more cross-site scripting vulnerabilities than previous methods. In older applications, used in previous research, we find vulnerabilities that the other methods miss. We also find new vulnerabilities in production software, including HotCRP, osCommerce, PrestaShop and WordPress.

1 Introduction

Ensuring the security of web applications is of paramount importance for our modern society. The dynamic nature of web applications, together with a plethora of different languages and frameworks, makes it particularly challenging for existing approaches to provide sufficient coverage of the existing threats. Even the web's main players, Google and Facebook, are prone to vulnerabilities, regularly discovered by security researchers. In 2019 alone, Google's bug bounty paid $6.5 million [16] and Facebook $2.2 million [12], both continuing the ever-increasing trend. *Cross-Site Scripting (XSS)* attacks, injecting malicious scripts in vulnerable web pages, represent the lion's share of web insecurities. Despite mitigations by the current security practices, XSS remains a prevalent class of attacks on the web [38]. Google rewards millions of dollars for XSS vulnerability reports yearly [21], and XSS is presently the most rewarded bug on both HackerOne [20] and Bugcrowd [5]. This motivates the focus of this paper on detecting vulnerabilities in web applications, with particular emphasis on XSS.

Blackbox web scanning: When such artifacts as the source code, models describing the application behaviors, and code annotations are available, the tester can use whitebox techniques that look for vulnerable code patterns in the code or vulnerable behaviors in the models. Unfortunately, these artifacts are often unavailable in practice, rendering whitebox approaches ineffective in such cases.

The focus of this work is on *blackbox* vulnerability detection. In contrast to whitebox approaches, blackbox detection techniques rely on no prior knowledge about the behaviors of web applications. This is the standard for security penetration testing, which is a common method for finding security vulnerabilities [31]. Instead, they acquire such knowledge by interacting with running instances of web applications with *crawlers*. Crawlers are a crucial component of blackbox scanners that explore the attack surface of web applications by visiting webpages to discover URLs, HTML form fields, and other input fields. If a crawler fails to cover the attack surface sufficiently, then vulnerabilities may remain undetected, leaving web applications ex-

posed to attacks.

Unfortunately, having crawlers able to discover in-depth behaviors of web applications is not sufficient to detect vulnerabilities. The detection of vulnerabilities often requires the generation of tests that can interact with the web application in non-trivial ways. For example, the detection of stored cross-site scripting vulnerabilities (stored XSS), a notoriously hard class of vulnerabilities [38], requires the ability to reason about the subtle dependencies between the control and data flows of web application to identify the page with input fields to inject the malicious XSS payload, and then the page that will reflect the injected payload.

Challenges: Over the past decade, the research community has proposed different approaches to increase the coverage of the attack surface of web applications. As JavaScript has rendered webpages dynamic and more complex, new ideas were proposed to incorporate these dynamic behaviors to ensure a correct exploration of the page behaviors (jÄk [30]) and the asynchronous HTTP requests (CrawlJAX [4, 26]). Similarly, other approaches proposed to tackle the complexity of the server-side program by reverse engineering (LigRE [10] and KameleonFuzz [11]) or inferring the state (Enemy of the State [8]) of the server, and then using the learned model to drive a crawler.

Unfortunately, despite the recent efforts, existing approaches do not offer sufficient coverage of the attack surface. To tackle this challenge, we start from two observations. First, while prior work provided solutions to individual challenges, leveraging their carefully designed combination has the potential to significantly improve the state of the art of modern web application scanning. Second, existing solutions focus mostly on handling control flows of web applications, falling short of taking into account intertwined dependencies between control and data flows. Consider, for example, the dependency between a page to add new users and the page to show existing users, where the former changes the state of the latter. Being able to extract and use such an inter-page dependency will allow scanners to explore new behaviors and detect more sophisticated XSS vulnerabilities.

Contributions: This paper presents Black Widow, a novel blackbox web application scanning technique that identifies and builds on three pillars: navigation modeling, traversing, and tracking inter-state dependencies.

Given a URL, our scanner creates a navigation model of the web application with a novel JavaScript dynamic analysis-based crawler able to explore both the static structure of webpages, i.e., anchors, forms, and frames, as well as discover and fire JavaScript events such as mouse clicks. Also, our scanner further annotates the model to capture the sequence of steps required

to reach a given page, enabling the crawler to retrace its steps. When visiting a webpage, our scanner enriches our model with data flow information using a black-box, end-to-end, dynamic taint tracking technique. Here, our scanner identifies input fields, i.e., taint source, and then probe them with unique strings, i.e., taint values. Later, the scanner checks when the strings re-surface in the HTML document, i.e., sinks. Tracking these taints allows us to understand the dependencies between different pages.

We implement our approach as a scanner on top of a modern browser with a state-of-the-art JavaScript engine. To empirically evaluate it, both in terms of coverage and vulnerability detection, we test it on two sets of web applications and compare the results with other scanners. The first set of web applications are older well-known applications that have been used for vulnerability testing before, e.g. WackoPicko and SCARF. The second set contains new production applications such as CMS platforms including WordPress and E-commerce platforms including PrestaShop and osCommerce. From this, we see that our approach improves code coverage by between 63% and 280% compared to other scanners across all applications. Across all web applications, our approach improves code coverage by between 6% and 62%, compared to the *sum* of all other scanners. In addition, our approach finds more XSS vulnerabilities in older applications, i.e. phpBB, SCARF, Vanilla and WackoPicko, that have been used in previous research. Finally, we also find multiple new vulnerabilities across production software including HotCRP, osCommerce, PrestaShop and WordPress.

Finally, while most scanners produce false positives, Black Widow is free of false positives on the tested applications thanks to its dynamic verification of code injections.

In summary, the paper offers the following contributions.

- We identify unsolved challenges for scanners in modern web applications and present them in Section 2.
- We present our novel approaches for finding XSS vulnerabilities using inter-state dependency analysis and crawling complex workflows in Section 3.
- We implement and share the source code of Black Widow[1]
- We perform a comparative evaluation of Black Widow on 10 popular web applications against 7 web application scanners.
- We present our evaluation in Section 4 showing that our approach finds 25 vulnerabilities, of which 6 are previously unknown in HotCRP, osCommerce, PrestaShop and WordPress. Additionally, we find more vulnerabil-

[1] Our implementation is available online on `https://www.cse.chalmers.se/research/group/security/black-widow/`

ities in older applications compared to other scanners. We also improve code coverage on average by 23%.
- We analyze the results and explain the important features required by web scanners in Section 5.

2 Challenges

Existing web application scanners suffer from a number of shortcomings affecting their ability to cope with the complexity of modern web applications [3, 9]. We observe that state-of-the-art scanners tend to focus on separate challenges to improve their effectiveness. For example, jÄk focuses on JavaScript events, Enemy of the State on application states, LigRE on reverse engineering and CrawlJAX on network requests. However, to successfully scan applications our insight is that these challenges must be solved simultaneously. This section focuses on these shortcomings and extracts the key challenges to achieve high code coverage and effective vulnerability detection.

High code coverage is crucial for finding any type of vulnerability as the scanner must be able to reach the code to test it. For vulnerability detection, we focus on stored XSS as it is known to be difficult to detect and a category of vulnerabilities poorly covered by existing scanners [3, 9]. Here the server stores and uses at a later time untrusted inputs in server operations, without doing proper validation of the inputs or sanitization of output.

A web application scanner tasked with the detection of subtle vulnerabilities like stored XSS faces three major challenges. First, the scanner needs to model the various states forming a web application, the connections and dependencies between states (Section 2.1). Second, the identification of these dependencies requires the scanner to be able to traverse the complex workflows in applications (Section 2.2). Finally, the scanner needs to track subtle dependencies between states of the web application (Section 2.3).

2.1 Navigation Modeling

Modern web applications are dynamic applications with an abundance of JavaScript code, client-side events and server-side statefulness. Modeling the scanner's interaction with both server-side and client-side code is complicated and challenging. Network requests can change the state of the server while clicking a button can result in changes to the DOM, which in turn generates new links or fields. These orthogonal problems must all be handled by the scanner to achieve high coverage and improved detection rate

of vulnerabilities. Consider the flow in an example web application in Figure 3.11. The scanner must be able to model links, forms, events and the interaction between them. Additionally, to enable workflow traversal, it must also model the path taken through the application. Finally, the model must support inter-state dependencies as shown by the dashed line in the figure.

The state-of-the-art consists of different approaches to navigation modeling. Enemy of the State uses a state machine and a directed graph to infer the server-side state. However, the navigation model lacks information about client-side events. In contrast, jÄk used a graph with lists inside nodes, to represent JavaScript events. CrawlJAX moved the focus to model JavaScript network requests. While these two model client-side, they miss other important navigation methods such as form submissions.

A navigation model should allow the scanner to efficiently and exhaustively scan a web application. Without correct modeling, the scanner will miss important resources or spend too much time revisiting the same or similar resources. To achieve this, the model must cover a multitude of methods for interaction with the application, including GET and POST requests, JavaScript events, HTML form and iframes.

In addition, the model should be able to accommodate dependencies. Client-side navigations, such as clicking a button, might depend on previous events. For example, the user might have to hover the menu before being able to click the button. Similarly, installation wizards can require a set of forms to be submitted in sequence.

With a solution to the modeling challenge, the next challenge is how the scanner should use this model, i.e. how should it traverse the model.

2.2 Traversing

To improve code coverage and vulnerability detection, the crawler component of the scanner must be able to traverse the application. In particular, the challenge of reproducing workflows is crucial for both coverage and vulnerability detection. The challenges of handling complex workflows include deciding in which order actions should be performed and when to perform possibly state-changing actions, e.g. submitting forms. Also, the workflows must be modeled at a higher level than network requests as simply replaying requests can result in incorrect parameter values, especially for context-dependent value such as a comment ID. In Figure 3.11, we can observe a workflow requiring a combination of normal link navigation, form submission and event interaction. Also, note that the forms can contain security nonces to protect against CSRF attacks. A side effect of this is that the scanner can not replay the request and just change the payload, but has to reload

the page and resubmit the form.

The current state-of-the-art focuses largely on navigation and exploration but misses out on global workflows. Both CrawlJAX and jÄk focused on exploring client-side events. By exploring the events in a depth-first fashion, jÄk can find sequences of events that could be exploited. However, these sequences do not extend across multiple pages, which will miss out on flows. Enemy of the State takes the opposite approach and ignores traversing client-side events and instead focuses on traversing server-side states. To traverse, they use a combination of picking links from the previous response and a heuristic method to traverse edges that are the least likely to result in a state change, e.g. by avoiding form submission until necessary. To change state they sometimes need to replay the request from the start. Replaying requests may not be sufficient as a form used to post comments might contain a submission ID or view-state information that changes for each request. Due to the challenge of reproducing these flows, their approach assumes the power to reset the full application when needed, preventing the approach from being used on live applications.

We note that no scanner handles combinations of events and classic page navigations. Both jÄk and CrawlJAX traverse with a focus on client-side state while Enemy of the State focus on links and forms for interaction. Simply combining the two approaches of jÄk and Enemy of the State is not trivial as their approaches are tailored to their goals. Enemy of the State uses links on pages to determine state changes, which are not necessarily generated by events.

Keeping the scanner authenticated is also a challenge. Some scanners require user-supplied patterns to detect authentication [28, 34, 36]. jÄk authenticates once and then assumes the state is kept, while CrawlJAX ignores it altogether. Enemy of the State can re-authenticate if they correctly detect the state change when logging out. Once again it is hard to find consensus on how to handle authentication.

In addition to coverage, traversing is important for the fuzzing part of the scanner. Simply exporting all requests to a standalone fuzzer is problematic as it results in loss of context. As such, the scanner must place the application in an appropriate state before fuzzing. Here some scanners take the rather extreme approach of trying to reset the entire web application before fuzzing each parameter [8, 10, 11]. jÄk creates a special attacker module that loads a URL and then executes the necessary events. This shows that in order to fuzz the application in a correct setting, without requiring a full restart of the application, the scanner must be able to traverse and attack both server-side and client-side components.

Solving both modeling and traversing should enable the scanner to crawl the application with improved coverage, allowing it to find more parameters to test. The final challenge, particularly with respect to stored XSS, is mapping the dependencies between different states in the application.

2.3 Inter-state Dependencies

It is evident that agreeing on a model that fits both client-side and server-side is hard, yet important. In addition, neither of the previous approaches are capable of modeling inter-state dependencies or general workflows. While Enemy of the State model states, they miss the complex workflows and the inter-state dependencies. The model jÄk uses can detect workflows on pages but fails to scale for the full application.

A key challenge faced by scanners is how to accurately and precisely model how user inputs affect web applications. As an example, consider the web application workflow in Figure 3.11 capturing an administrator registering a new user. In this workflow, the administrator starts from the index page (i.e., `index.php`) and navigates to the login page (i.e., `login.php`). Then, the administrator submits the password and lands on the administrator dashboard (i.e., `admin.php`). From the dashboard, the administrator reaches the user management page (i.e., `admin.php#users`), and submits the form to register a new user. Then, the web application stores the new user data in the database, and, as a result of that, the data of the new user is shown when visiting the page of existing users (i.e., `view_users.php`). Such a workflow shows two intricate dependencies between two states of the web application: First, an action of `admin.php#users` can cause a transition of `view_users.php`, and second, the form data submitted to `admin.php#users` is reflected in the new state of `admin.php#users`.

To detect if the input fields of the form data are vulnerable to, e.g., cross-site scripting (XSS), a scanner needs to inject payloads in the form of `admin.php#users` and then reach `view_users.php` to verify whether the injection was successful. Unfortunately, existing web scanners are not aware of these inter-state dependencies, and after injecting payloads, they can hardly identify the page where and whether the injection is reflected.

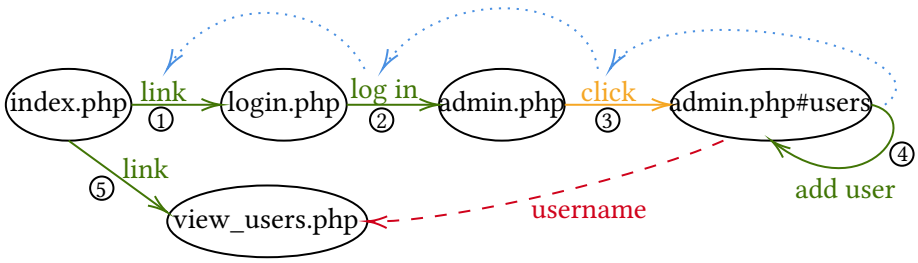

Figure 3.11: Example of a web application where anyone can see the list of users and the admin can add new users. The dashed red line represents the inter-state dependency. Green lines are HTML5 and orange symbolises JavaScript. The dotted blue lines between edges would be added by our scanner to track its path. The sequence numbers shown the necessary order to find the inter-state dependency.

3 Approach

Motivated by the challenges in Section 2, this section presents our approach to web application scanning. The three key ingredients of our approach are edge-driven navigation with path-augmentation, complex workflow traversal, and fine-grained inter-state dependency tracking. We explain how we connect these three parts in Algorithm 1. In addition to the three main pillars, we also include a section about the dynamic XSS detection used in Black Widow and motivate why false positives are improbable.

Algorithm 1 takes a single target URL as an input. We start by creating an empty node, allowing us to create an initial edge between the empty node and the node containing the input URL. The main loop picks an unvisited edge from the navigation graph and then traverses it, executing the necessary workflows as shown in Algorithm 2. In Algorithm 2, we use the fact that each edge knows the previous edge. The `isSafe` function in Algorithm 2 checks if the type of action, e.g. JavaScript event or form submission, is *safe*. We consider a type to be *safe* if it is a GET request, more about this in Section 3.2. Once the safe edge is found we navigate the chain of actions. Following this navigation, the scanner is ready to parse the page. First, we inspect the page for inter-state dependency tokens and add the necessary dependency edges, as shown in Algorithm 3. Each `token` will contain a taint value, explained more in Section 3.3, a source edge and a sink edge. If a source and sink are found, our scanner will fuzz the source and check the sink. Afterward, we extract any new possible navigation resources and add them to the graph.

Next, we fuzz any possible parameters in the edge and then inject a taint token. The order is important as we want the token to overwrite any stored fuzzing value. Finally, the edge is marked as visited and the loop repeats.

The goal of this combination is to improve both vulnerability detection and code coverage. The three parts of the approach support each other to achieve this. A strong model that handles different navigation methods and supports augmentation with path and dependency information will enable a richer interaction with the application. Based on the model we can build a strong crawler component that can handle complex workflow which combines requests and client-side events. Finally, by tracking inter-state dependencies we can improve detection of stored vulnerabilities.

```
   Data: Target url
 1 Global: tokens // Used in Algorithm 3
 2 Graph navigation; // Augmented navigation graph
 3 navigation.addNode(empty);
 4 navigation.addNode(url);
 5 navigation.addEdge(empty, url);
 6 while unvisited edge e in navigation do
 7 |   traverse(e); // See Algorithm 2
 8 |   inspectTokens(e, navigation); // See Algorithm 3
 9 |   resources = extract({urls, forms, events, iframes});
10 |   for resource in resources do
11 |   |   navigation.addNode(resource) navigation.addEdge(e.targetNode,
   |   |   resource)
12 |   end
13 |   attack(e);
14 |   injectTokens(e);
15 |   mark e as visited;
16 end
```

Algorithm 1: Scanner algorithm

```
 1 Function traverse(e: edge)
 2 |   workflow = []; // List of edges
 3 |   currentEdge = e;
 4 |   while prevEdge = currentEdge.previous do
 5 |   |   workflow.prepend(currentEdge);
 6 |   |   if isSafe(currentEdge.type) then
 7 |   |   |   break;
 8 |   |   end
 9 |   |   currentEdge = prevEdge
10 |   end
11 |   navigate(workflow);
12 end
```

Algorithm 2: Traversal algorithm

```
1  Function inspectTokens(e: edge, g: graph)
2      for token in tokens do
3          if pageSource(e) contains token.value then
4              token.sink = e;
5              g.dependency(token.source, token.sink);
6              attack(token.source, token.sink);
7          end
8      end
9  end
10 Function injectTokens(e: edge)
11     for parameter in e do
12         token.value = generateToken();
13         token.source = e;
14         tokens.append(token);
15         inject token in parameter;
16     end
17 end
```

Algorithm 3: Inter-state dependency algorithms

3.1 Navigation Modeling

Our approach is model-based in the sense that it creates, maintains, and uses a model of the web application to drive the exploration and detection of vulnerabilities. Our model covers both server-side and client-side aspects of the application. The model tracks server-side inter-state dependencies and workflows. In addition, it directly captures elements of the client-side program of the web application, i.e., HTML and the state of the JavaScript program.

Model Construction Our model is created and updated at run-time while scanning the web application. Starting from an initial URL, our scanner retrieves the first webpage and the referenced resources. While executing the loaded JavaScript, it extracts the registered JavaScript events and adds them to our model. Firing an event may result in changing the internal state of the JavaScript program, or retrieving a new page. Our model captures all these aspects and it keeps track of the sequence of fired events when revisiting the web application, e.g., for the detection of vulnerabilities.

Accordingly, we represent web applications with a labeled directed graph, where each node is a state of the client-side program and edges are the action (e.g., click) to move from one state to another one. The state of our model contains both the state of the page, i.e., the URL of the page, and the state of the JavaScript program, i.e., the JavaScript event that triggered the

execution. Then, we use labeled edges for state transitions. Our model supports four types of actions, i.e., GET requests, form submission, iframes and JavaScript events. While form submissions normally result in GET or POST requests, we need a higher-level model for the traversing method explained in Section 3.2. We consider iframes as actions because we need to model the inter-document communication between the iframe and the parent, e.g firing an event in the parent might affect the iframe. By simply considering the iframe source as a separate URL, scanners will miss this interaction. Finally, we annotate each edge with the previous edge visited when crawling the web application, as shown in Figure 3.11. Such an annotation will allow the crawler to reconstruct the *paths* within the web application, useful information for achieving deeper crawling and when visiting the web application for testing.

Extraction of Actions The correct creation of the model requires the ability to extract the set of possible actions from a web page. Our approach uses dynamic analysis approach, where we load a page and execute it in a modified browser environment, and then we observe the execution of the page, monitoring for calls to browser APIs to register JavaScript events and modification of the DOM tree to insert tags such as forms and anchors.

Event Registration Hooking Before loading a page we inject JavaScript which allows us to wrap functions such as `addEventListener` and detect DOM objects with event handlers. We accomplish this by leveraging the JavaScript libraries developed for the jÄk scanner [30]. While lightweight and easy to use, in-browser instrumentation is relatively fragile. A more robust approach could be directly modifying the JavaScript engine or source-to-source compile the code for better analysis.

DOM Modification To detect updates to the page we rescan the page whenever we execute an event. This allows us to detect dynamically added items.

Infinite Crawls When visiting a webpage, crawlers can enter in an infinite loop where they can perform the same operation endlessly. Consider the problem of crawling an online calendar. When a crawler clicks on the *View next week* button, the new page may have a different URL and content. The new page will container again the button *View next week*, triggering an infinite loop. An effective strategy to avoid infinite crawls is to define (i) a set of heuristics that determine when two pages or two actions are similar, and (ii) a hard limit to the maximum number of "similar" actions performed by the crawler. In our approach, we define two pages to be similar if they share the same URL except for the query string and the fragments. For example,

`https://example.domain/path/?x=1` and `https://example.domain/path/?x=2` are similar whereas `https://example.domain/?x=1` and `https://example.domain/path/?x=2` are different. The hard limit is a configuration parameter of our approach.

3.2 Traversal

To traverse the navigation model we pick unvisited edges from the graph in the order they were added, akin to breadth-first search. This allows the scanner to gain an overview of the application before diving into specific components. The edges are weighted with a positive bias towards form submission, which enables this type of deep-dive when forms are detected.

To handle the challenge of session management, we pay extra attention to forms containing password fields, as this symbolizes an opportunity to authenticate. Not only does this enable the scanner to re-authenticate but it also helps when the application generates a login form due to incorrect session tokens. Another benefit is a more robust approach to complicated login flows, such as double login to reach the administrator page—we observed such workflow in phpBB, one of the web applications that we evaluated.

The main challenge to overcome is that areas of a web application might require the user to complete a specific sequence of actions. This could, for example, be to review a comment after submitting it or submit a sequence of forms in a configuration wizard. It is also common for client-side code to require chaining, e.g. hover a menu before seeing all the links or click a button to dynamically generate a new form.

We devise a mechanism to handle navigation dependencies by modeling the workflows in the application. Whenever we need to follow an edge in the navigation graph, we first check if the previous edge is considered *safe*. Here we define safe to be an edge which represents a GET request, similar to the HTTP RFC [14]. If the edge is safe, we execute it immediately. Otherwise, we recursively inspect the previous edge until a safe edge is found, as shown in Algorithm 2. Note that the first edge added to the navigation graph is always a GET request, which ensures a base case. Once the safe edge is found, we execute the full workflow of edges leading up to the desired edge. Although the RFC defines GET requests to be idempotent, developers can still implement state-changing functions on GET requests. Therefore, considering GET requests as *safe* is a performance trade-off. This could be deactivated by a parameter in Black Widow, causing the scanner to traverse back to the beginning.

Using Figure 3.11 as an example if the crawler needed to submit a form on `admin.php#users` then it would first have to load `login.php` and then

submit that form, followed by executing a JavaScript event to dynamically add the user form.

We chose to only chain actions to the previous GET request, as they are deemed safe. Chaining from the start is possible, but it would be slow in practice.

3.3 Inter-state Dependencies

One of the innovative aspects of our approach is to identify and map the ways user inputs are connected to the states of a web application. We achieve that by using a dynamic, end-to-end taint tracking while visiting the web application. Whenever our scanner identifies an input field, i.e., a *source*, it will submit a unique token. After that, the scanner will look for the token when visiting other webpages, i.e., *sinks*.

Tokens To map source and sinks, we use string tokens. We designed tokens to avoid triggering filtering functions or data validation checks. At the same time, we need tokens with a sufficiently high entropy to not be mistaken for other strings in the application. Accordingly, we generate tokens as pseudo-random strings of eight lowercase characters e.g. `frcvwwzm`. This is what `generateToken()` does in Algorithm 3. This could potentially be improved by making the tokens context-sensitive, e.g. by generating numeric tokens or emails. However, if the input is validated to only accept numbers, for example, then XSS is not possible.

Sources and Sinks The point in the application where the token is injected defines the *source*. More specifically, the source is defined as a tuple containing the edge in the navigation graph and the exact parameter where the token was injected. The resource in the web application where the token reappears defines the *sink*. All the sinks matching a certain source will be added to a set which in turn is connected to the source. Similar to the sources, each sink is technically an edge since they carry more context than a resource node. Since each source can be connected to multiple sinks, the scanner needs to check each sink for vulnerabilities whenever a payload is injected into a source.

In our example in Figure 3.11, we have one source and one connected sink. The source is the *username* parameter in the form on the management page and the sink is the view users page. If more parameters, e.g. email or signature, were also reflected then these would create new dependency edges in the graph.

3.4 Dynamic XSS detection

After a payload has been sent, the scanner must be able to detect if the payload code is executed. Black Widow uses a fine-grained dynamic detection mechanism, making false positives very improbable. We achieve this by injecting our JavaScript function `xss(ID)` on every page. This function adds `ID` to an array that our scanner can read. Every payload generated by Black Widow will try to call this function with a random ID, e.g. `<script>xss(71942203)</script>` Finally, by inspecting the array we can detect exactly which payloads resulted in code execution.

For this to result in a false positive, the web application would have to actively listen for a payload, extract the ID, and then run our injected `xss(ID)` function with a correct ID.

4 Evaluation

In this section, we present the evaluation of our approach and the results from our experiments. In the next section, we perform an in-depth analysis of the factors behind the results.

To evaluate the effectiveness of our approach we implement it in our scanner Black Widow and compare it with 7 other scanners on a set of 10 different web applications. We want to compare both the crawling capabilities and vulnerability detection capabilities of the scanners. We present the implementation details in Section 4.1. The details of the experimental setup are presented in Section 4.2. To measure the crawling capabilities of the scanners we record the code coverage on each of application. The code coverage results are presented in Section 4.3. For the vulnerability detection capabilities, we collect the reports from each scanner. We present both the reported vulnerabilities and the manually verified ones in Section 4.4.

4.1 Implementation

Our prototype implementation follows the approach presented above in Section 3. It exercises full dynamic execution capabilities to handle such dynamic features of modern applications like AJAX and dynamic code execution, e.g. `eval`. To achieve this we use Python and Selenium to control a mainstream web browser (Chrome). This gives us access to a state-of-the-art JavaScript engine. In addition, by using a mainstream browser we can be more certain that the web application is rendered as intended.

We leverage the JavaScript libraries developed for the jÄk scanner [30]. These libraries are executed before loading the page. This allows us to wrap

Table 3.7: Lines of code (LoC) executed on the server. Each column represents the comparison between Black Widow and another crawler. The cells contain three numbers: unique LoC covered by Black Widow ($A \setminus B$), LoC covered by both crawlers ($A \cap B$) and unique LoC covered by the other crawler ($B \setminus A$). The numbers in bold highlight which crawler has the best coverage.

Crawler	Arachni			Enemy			jÄk			Skipfish			w3af			Wget			ZAP		
	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$
Drupal	**35 146**	22 870	757	6 365	51 651	**20 519**	**25 198**	32 818	5 846	**29 873**	28 143	937	**32 213**	25 803	725	**32 981**	25 035	498	**15 610**	42 406	2 591
HotCRP	**2 416**	16 076	948	**16 573**	1 919	0	**6 771**	11 721	271	**11 295**	7 197	31	**3 217**	15 275	768	**16 345**	2 147	3	**16 001**	2 491	24
Joomla	**14 573**	29 263	1 390	**33 335**	10 501	621	**24 728**	19 108	1 079	**33 254**	10 582	328	**12 533**	31 303	1 255	**33 975**	9 861	576	**7 655**	36 181	1 659
osCommerce	**3 919**	6 722	172	**9 626**	1 015	15	**4 171**	6 470	507	**4 964**	5 677	110	**5 601**	5 040	661	**6 070**	4 571	103	**6 722**	3 919	209
phpBB	**2 822**	5 178	492	**2 963**	5 037	337	**3 150**	4 850	348	**4 643**	3 357	72	**4 312**	3 688	79	**4 431**	3 569	21	**4 247**	3 753	65
PrestaShop	**105 974**	75 924	65 650	**157 095**	24 803	3 332	**155 579**	26 319	58	**138 732**	43 166	1 018	**156 513**	25 385	3 053	**148 868**	33 030	118	**141 032**	40 866	110
SCARF	**189**	433	12	**270**	352	5	**342**	280	2	**464**	158	5	**404**	218	6	**520**	102	2	**340**	282	2
Vanilla	**5 381**	9 908	491	**6 032**	9 257	185	**3 122**	12 167	536	**8 285**	7 004	577	**8 202**	7 087	171	**8 976**	6 313	18	**8 396**	6 893	145
WackoPicko	**202**	566	2	**58**	710	9	**463**	305	0	**274**	494	14	**111**	657	9	**495**	273	0	**379**	389	2
WordPress	**8 871**	45 345	1 615	**35 092**	19 124	256	**18 572**	35 644	579	**7 307**	46 909	5 114	**26 785**	27 431	640	**37 073**	17 143	73	**25 732**	28 484	781

functions such as `addEventListener` and detect DOM objects with event handlers.

4.2 Experimental Setup

In this section, we present the configuration and methodology of our experiments.

Code Coverage To evaluate the coverage of the scanners we chose to compare the lines of code that were executed on the server during the session. This is different from previous studies [8, 30], which relied on requested URLs to determine coverage. While comparing URLs is easier, as it does not require the web server to run in debug mode, deriving coverage from it becomes harder. URLs can contain random parameter data, like CSRF tokens, that are updated throughout the scan. In this case, the parameter data has a low impact on the true coverage. Conversely, the difference in coverage between `main.php?page=news` and `main.php?page=login` can be large. By focusing on the execution of lines of code we get a more precise understanding of the coverage.

Calculating the total number of lines of code accurately in an application is a difficult task. This is especially the case in languages like PHP where code can be dynamically generated server-side. Even if possible, it would not give a good measure for comparison as much of the code could be unreachable. This is typically the case for applications that have installation code, which

is not used after completing it.

Instead of analyzing the fraction of code executed in the web application, we compare the number of lines of code executed by the scanners. This gives a relative measure of performance between the scanners. It also allows us to determine exactly which lines are found by multiple scanners and which lines are uniquely executed.

To evaluate the code coverage we used the Xdebug [33] module in PHP. This module returns detailed data on the lines of code that are executed in the application. Each request to the application results in a separate list of lines of code executed for the specific request.

Vulnerabilities In addition to code coverage, we also evaluate how good the scanners are at finding vulnerabilities. This includes how many vulnerabilities they can find and how many false positives they generate. While there are many vulnerability types, our study focuses on both reflected and stored XSS.

To evaluate the vulnerability detection capabilities of the scanners, we collect and process all the vulnerabilities they report. First, we manually analyze if the vulnerabilities can be reproduced or if they should be considered false positives. Second, we cluster similar vulnerability reports into a set of unique vulnerabilities to make a fair comparison between the different reporting mechanisms in the scanners. We do this because some applications, e.g. SCARF, can generate an infinite number of vulnerabilities by dynamically adding new input fields. These should be clustered together. Classifying the uniqueness of vulnerabilities is no easy task. What we aim to achieve is a clustering in which each injection corresponds to a unique line of code on the server. That is, if a form has multiple fields that are all stored using the same SQL query then all these should count as one injection. The rationale is that it would only require the developer to change one line in the server code. Similarly, for reflected injections, we cluster parameters of the same request together. We manually inspect the web application source code for each reported true-positive vulnerability to determine if they should be clustered.

Scanners We compare our scanner Black Widow with both Wget [27] for code coverage reference and 6 state-of-the-art open-source web vulnerability scanners from both academia and the web security community: Arachni [36], Enemy of the State [8], jÄk [30], Skipfish [42], w3af [34] and ZAP [28]. We use Enemy of the State and jÄk as they are state-of-the-art academic black-box scanners. Skipfish, Wget and w3af are included as they serve as good

benchmarks when comparing with previous studies [8, 30]. Arachni and ZAP are both modern open-source scanners that have been used in more recent studies [19]. Including a pure crawler with JavaScript capabilities, such as CrawlJAX [26], could serve as a good coverage reference. However, in this paper we focus on coverage compared to other vulnerability scanners. We still include Wget for comparison with previous studies. While it would be interesting to compare our results with commercial scanners, e.g. Burp Scanner [32], the closed source nature of these tools would make any type of feature attribute hard.

We configure the scanners with the correct credentials for the web application. When this is not possible we change the default credentials of the application to match the scanner's default values. Since the scanners have different capabilities, we try to configure them with as similar configurations as possible. This entails activating crawling components, both static and dynamic, and all detection of all types of XSS vulnerabilities.

Comparing the time performance between scanners is non-trivial to do fairly as they are written in different languages and some are sequential while others run in parallel. Also, we need to run some older ones in VMs for compatibility reasons. To avoid infinite scans, we limit each scanner to run for a maximum of eight hours.

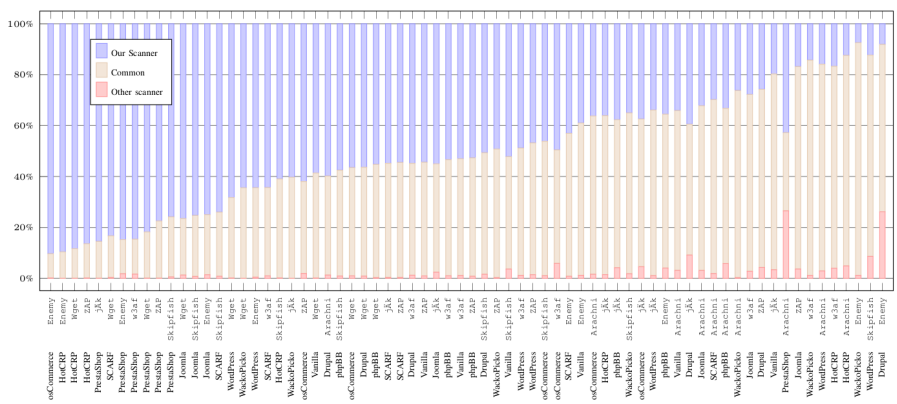

Figure 3.12: Each bar compares our scanner to one other scanner on a web application. The bars show three fractions: unique lines we find, lines both find and lines uniquely found by the other scanner.

Web Applications To ensure that the scanners can handle different types of web applications we test them on 10 different applications. The appli-

cations range from reference applications that have been used in previous studies to newer production-grade applications. Each application runs in a VM that we can reset between runs to improve consistency.

We divide the applications into two different sets. Reference applications with known vulnerabilities: phpBB (2.0.23), SCARF (2007), Vanilla (2.0.17.10) and WackoPicko (2018); and modern production-grade applications: Drupal (8.6.15), HotCRP (2.102), Joomla (3.9.6), osCommerce (2.3.4.1), PrestaShop (1.7.5.1) and WordPress (5.1).

4.3 Code Coverage Results

This section presents the code coverage in each web application by all of the crawlers. Table 3.7 shows the number of unique lines of code that were executed on the server. Black Widow has the highest coverage on 9 out of the 10 web applications.

Using Wget as a baseline Table 3.7 illustrates that Black Widow increases the coverage by almost 500% in SCARF. Similarly with modern production software, like PrestaShop, we can see an increase of 256% in coverage compared to Wget. Even when comparing to state-of-the-art crawlers like jÄk and Enemy of the State we have more than 100% increase on SCARF and 320% on modern applications like PrestaShop. There is, however, a case where Enemy of the State has the highest coverage on Drupal. This case is discussed in more detail in Section 5.1.

While it would be beneficial to know how far we are from perfect coverage, we avoid calculating a ground truth on the total number of lines of code for the applications as it is difficult to do in a meaningful way. Simply aggregating the number of lines in the source code will misrepresent dynamic code, e.g. `eval`, and count dead code, e.g. installation code.

Table 3.8: Unique lines our scanner finds ($A \setminus U$) compared to the union of all other scanners (U).

Application	Our scanner $A \setminus U$	Other scanners U	Improvement $\lvert A \setminus U \rvert / \lvert U \rvert$
Drupal	4 378	80 213	+5.5%
HotCRP	1 597	18 326	+8.7%
Joomla	5 134	42 443	+12.1%
osCommerce	2 624	9 216	+28.5%
phpBB	2 743	5 877	+46.7%
PrestaShop	95 139	153 452	+62.0%
SCARF	176	464	+37.9%
Vanilla	2 626	14 234	+18.4%
WackoPicko	50	742	+6.7%
WordPress	3 591	58 131	+6.2%

Table 3.9: Comparison of unique lines of code found by our scanner ($A \setminus B$) and the other scanners ($B \setminus A$). Improvement is new lines found by our scanner divided by the other's total.

Crawler	Our scanner $A \setminus B$	Other scanners $B \setminus A$	Other's total B	Improvement $\lvert A \setminus B \rvert / \lvert B \rvert$
Arachni	179 477	71 489	283 664	+63.3%
Enemy	267 372	25 268	149 548	+178.8%
jÄk	242 066	9 216	158 802	+152.4%
Skipfish	239 064	8 206	160 794	+148.7%
w3af	249 881	7 328	149 099	+167.6%
Wget	289 698	1 405	103 359	+280.3%
ZAP	226 088	5 560	171 124	+132.1%

We also compare Black Widow to the combined efforts of the other scanners to better understand how we improve the state-of-the-art. Table 3.8 has three columns containing the number of lines of code that Black Widow finds which none of the others find, the combined coverage of the others and finally our improvement in coverage. In large applications, like PrestaShop, Black Widow was able to find 53 266 lines of code that none of the others

found. For smaller applications, like phpBB, we see an improvement of up to 46.7% compared to the current state-of-the-art.

To get a better understanding of which parts of the application the scanners are exploring, we further compare the overlap in the lines of code between the scanners. In Table 3.9 we present the number of unique lines of code Black Widow find compared to another crawler. The improvement is calculated as the number of unique lines we find divided by the *total* number of lines the other crawlers find.

We plot the comparison for all scanners on all platforms in Figure 3.12. In this figure, each bar represents the fraction of lines of code attributed to each crawler. At the bottom is the fraction found only by the other crawlers, in the middle the lines found by both and on top are the results found by Black Widow. The bars are sorted by the difference of unique lines found by Black Widow and the other crawlers. Black Widow finds the highest number of unique lines of code in all cases except the rightmost, in which Enemy of the State performed better on Drupal. The exact number can be found in Table 3.7.

Table 3.10: Number of reported XSS injections by the scanners and the classification of the injection as either reflected or stored.

Crawler	Arachni		Enemy		jÄk		Skipfish		w3af		Widow		ZAP	
Type	R	S	R	S	R	S	R	S	R	S	R	S	R	S
Drupal	-	-	-	-	-	-	-	-	-	-	-	-	-	-
HotCRP	-	-	-	-	-	-	-	-	-	-	1	-	-	-
Joomla	-	-	8	-	-	-	-	-	-	-	-	-	-	-
osCommerce	-	-	-	-	-	-	-	-	-	-	1	1	9	-
phpBB	-	-	-	-	-	-	-	-	-	-	-	32	-	-
PrestaShop	-	-	-	-	-	-	-	-	-	-	2	-	-	-
SCARF	31	-	-	-	-	-	-	-	1	-	3	5	-	-
Vanilla	2	-	-	-	-	-	-	-	-	-	1	2	-	-
WackoPicko	3	1	2	1	13	-	1	1	1	-	3	2	-	-
WordPress	-	-	-	-	-	-	-	-	-	-	1	1	-	-

Table 3.11: Number of unique and correct XSS injections by the scanners and the classification of the injection as either reflected or stored.

Crawler	Arachni		Enemy		jÄk		Skipfish		w3af		Widow		ZAP	
Type	R	S	R	S	R	S	R	S	R	S	R	S	R	S
Drupal	-	-	-	-	-	-	-	-	-	-	-	-	-	-
HotCRP	-	-	-	-	-	-	-	-	-	-	1	-	-	-
Joomla	-	-	-	-	-	-	-	-	-	-	-	-	-	-
osCommerce	-	-	-	-	-	-	-	-	-	-	1	1	-	-
phpBB	-	-	-	-	-	-	-	-	-	-	-	3	-	-
PrestaShop	-	-	-	-	-	-	-	-	-	-	1	-	-	-
SCARF	3	-	-	-	-	-	-	-	1	-	3	5	-	-
Vanilla	-	-	-	-	-	-	-	-	-	-	1	2	-	-
WackoPicko	3	1	2	1	1	-	1	-	1	-	3	2	-	-
WordPress	-	-	-	-	-	-	-	-	-	-	1	1	-	-

4.4 Code Injection Results

This section presents the results from the vulnerabilities the different scanners find. To be consistent with the terminology used in previous works [8, 30], we define an XSS vulnerability to be any injected JavaScript code that results in execution. While accepting JavaScript from users is risky in general, some applications, like Wordpress, have features which require executing user supplied JavaScript. In Section 5.7 we discuss the impact and exploitability of the vulnerabilities our scanner finds.

In Table 3.10 we list all the XSS vulnerabilities found by the scanners on all the applications. The table contains the number of self-reported vulnerabilities. After removing the false positives and clustering similar injections, as explained in Section 4.2, we get the new results in Table 3.11. The results from Table 3.11 show that Black Widow outperforms the other scanners on both the reference applications and the modern applications. In total, Black Widow finds 25 unique vulnerabilities, which is more than 3 times as many as the second-best scanner. Of these 25, 6 are previously unknown vulnerabilities in modern applications. We consider the remaining 19 vulnerabilities to be known for the following reasons. First, all WackoPicko vulnerabilities are implanted by the authors and they are all known by design. Second, SCARF has been researched thoroughly and vulnerabilities may be already known. We conservatively assumed the eight vulnerabilities to be known. Third, the vulnerabilities on phpBB and Vanilla were fixed in their newest versions.

It is important to note we did not miss any vulnerability that the others

found. However, there were cases where both Black Widow and other scanners found the same vulnerability but by injecting different parameters. We explore these cases more in Section 5.6. Furthermore, Black Widow is the only scanner that finds vulnerabilities in the modern web applications.

4.5 Takeaways

We have shown that our scanner can outperform the other scanners in terms of both code coverage and vulnerability detection. Figure 3.12 and Table 3.7 show that we outperform the other scanners in 69 out of 70 cases. Additionally, Table 3.8 and Table 3.9 show we improve code coverage by between 63% and 280% compared to the other scanners and by between 6% and 62%, compared to the *sum* of all other scanners. We also improve vulnerability detection, as can be seen in Table 3.10 and Table 3.11. Not only do we match the other scanners but we also find new vulnerabilities in production applications.

In the next section, we will analyze these results closer and conclude which features allowed us to improve coverage and vulnerability detection. We also discuss what other scanners did better than us.

5 Analysis of Results

The results from the previous section show that the code coverage of our scanner outperforms the other ones. Furthermore, we find more code injections and in particular more stored XSS. In this section, we analyze the factors which led to our advantage. We also analyze where and why the other scanners performed worse.

Since we have access to the executed lines of code we can closely analyze the path of the scanner through the application. We utilize this to analyze when the scanners miss injectable parameters, what values they submit, when they fail to access parts of the application and how they handle sessions.

We start by presenting interesting cases from the code coverage evaluation in Section 5.1, followed by an analysis of the reported vulnerabilities from all scanners in Section 5.2. In Section 5.3, we discuss the injections our scanner finds and compare it with what the others find. In Section 5.4, we perform two case studies of vulnerabilities that only our scanner finds and which requires both workflow traversal and dependency analysis. Finally, in Section 5.5, we extract the crucial features for finding injections based on all vulnerabilities that were found.

5.1 Coverage Analysis

As presented in Section 4.3, Black Widow improved code coverage, compared to the aggregated result of all the other scanners, ranged from 5.5% on Drupal to 62% on PrestaShop. Comparing the code coverage to each scanner, Black Widow's improvement ranged from 63.3% against Arachni to 280% against Wget. In this section, we analyze the factors pertaining to code coverage by inspecting the performance of the different scanners. To better understand our performance we divide the analysis into two categories. We look at both cases where we have low coverage compared to the other scanners and cases where we have high relative coverage.

Low coverage As shown in Figure 3.12, Enemy of the State is the only scanner that outperforms Black Widow and this is specifically on Drupal. Enemy of the State high coverage on Drupal is because it keeps the authenticated session state by avoiding logging out. The reason Black Widow lost the state too early was two-fold. First, we use a heuristic algorithm, as explained in Section 3.2 to select the next edge and unfortunately the logout edge was picked early. Second, due to the structure of Drupal, our scanner did not manage to re-authenticate. In particular, this is because, in contrast to many other applications, Drupal does not present the user with a login form when they try to perform an unauthorized operation. To isolate the reason for the lower code coverage, we temporarily blacklist the Drupal logout function in our scanner. This resulted in our scanner producing similar coverage to Enemy of the State, ensuring the factor behind the discrepancy is session handling.

Skipfish performs very well on WordPress, which seems surprising since it is a modern application that makes heavy use of JavaScript. However, WordPress degrades gracefully without JavaScript, allowing scanners to find multiple pages without using JavaScript. Focusing on static pages can generate a large coverage but, as is evident from the detected vulnerabilities, does not imply high vulnerability detection.

High coverage Enemy of the State also performs worse against Black Widow on osCommerce and HotCRP. This is because Enemy of the State is seemingly entering an infinite loop, using 100% CPU without generating any requests. This could be due to an implementation error or because the state inference becomes too complicated in these applications.

Although Black Widow performs well against Wget, Wget still finds some unique lines, which can seem surprising as it has previously been used as a reference tool [8, 30]. Based on the traces and source code, we see that most

of the unique lines of code Wget finds are due to state differences, e.g. visiting the same page Black Widow finds but while being unauthenticated.

5.2 False positives and Clustering

To better understand the reason behind the false positives, and be transparent about our clustering, we analyze the vulnerabilities reported in Table 3.10. For each scanner with false positives, we reflect on the reasons behind the incorrect classification and what improvements are required. We do not include w3af in the list as it did not produce any false positives or required any clustering.

a) Arachni reports two reflected XSS vulnerabilities in Vanilla. The injection point was a Cloudflare cookie used on the online support forum for the Vanilla web application. The cookie is never used in the application and we were unable to reproduce the injection. In addition, Arachni finds 31 XSS injections on SCARF. Many of these are incorrect because Arachni reuses payloads. For example, by injecting into the title of the page, all successive injection will be label as vulnerable.

b) Enemy of the State claims the discovery of 8 reflected XSS vulnerabilities on Joomla. However, after manual analysis, none of these result in code execution. The problem is that Enemy of the State interprets the reflected payload as an executed payload. It injects, `eval(print "[random]")`, into a search field and then detects that `"[random]"` is reflected. It incorrectly assumes this is because `eval` and `print` were executed. For this reason, we consider Enemy of the State to find 0 vulnerabilities on Joomla.

c) jÄk reports 13 vulnerabilities on WackoPicko. These 13 reports were different payloads used to attack the search parameter. After applying our clustering method, we consider jÄk to find one unique vulnerability.

d) Black Widow finds 32 stored vulnerabilities on phpBB. Most of these parameters are from the configuration panel and are all used in the same database query. Therefore, only 3 can be considered unique. Two parameters on PrestaShop are used in the same request, thus only one is considered unique. Black Widow did not produce any false positives thanks to our dynamic detection method explained in section 3.4

e) Skipfish claims the detection of a stored XSS in WackoPicko in the image data parameter when uploading an image. However, the injected JavaScript could not be executed. Interesting to note is that Skipfish was able to inject JavaScript into the guestbook but was not able to detect it.

f) ZAP claims to find 9 reflected XSS injection on osCommerce. They are all variations of injecting `javascript:alert(1)` into the parameter of

a link. Since it was just part of a parameter and not a full URL, the JavaScript code will never execute. Thus, all 9 injections were false positives.

5.3 What We Find

In this section, we present the XSS injections our scanner finds in the different applications. We also extract the important features which made it possible to find them.

HotCRP: Reflected XSS in bulk user upload The admin can upload a file with users to add them in bulk. The name of the file is then reflected on the upload page. To find this, the scanner must be able to follow a complex workflow that makes heavy use of JavaScript, as well as handle file parameters. It is worth noting that the filename is escaped on other pages in HotCRP but missed in this case.

osCommerce; Stored and reflected XSS Admins can change the tax classes in osCommerce and two parameters are not correctly filtered, resulting in stored XSS vulnerabilities. The main challenge to find this vulnerability was to find the injection point as this required us to interact with a navigation bar that made heavy use of JavaScript.

We also found three vulnerable parameters on the review page. These parameters were part of a form and their types were *radio* and *hidden*. This highlights that we still inject all parameters, even if they are not intended to be changed.

phpBB; Multiple Stored XSS in admin backend Admins can change multiple different application settings on the configuration page, such as flooding interval for posts and max avatar file size. On a separate page, they can also change the rank of the admin to a custom title. In total, this results in 32 vulnerable parameters that can be clustered to 3 unique ones. These require inter-state dependency analysis to solve. Once a setting is changed, the admin is met with a "Successful update" message, which does not reflect the injection. Thus, the dependency must be found to allow for successful fuzzing.

PrestaShop; Reflected XSS in admin dashboard The admin dashboard allows the admin to specify a date range for showing statistics. Two parameters in this form are not correctly filtered and result in a reflected XSS.

Finding these requires a combination of modeling JavaScript events and handling workflows. To find this form the scanner must first click on a button on the dashboard.

SCARF; Stored XSS in comments There are many vulnerabilities in SCARF, most are quite easy to find. Instead of mentioning all, we focus on one that requires complex workflows, inter-state dependencies and was only found by us. The message field in the comment section of conference papers is vulnerable. What makes it hard to find is the traversing and needed before posting the comment and the inter-state dependency analysis needed to find the reflection. The scanner must first create a user, then create a conference, after which it can upload a paper that can be commented on.

Vanilla; Stored and reflected XSS The language tag for the RSS feed is vulnerable and only reflected in the feed. Note that the feed is served as HTML, allowing JavaScript to execute. There is also a stored vulnerability in the comment section which can be executed by saving a comment as a draft and then viewing it. Both of these require inter-state dependency analysis to find the connecting between language settings and RSS feeds, as well as posting comments and viewing drafts.

Black Widow also found a reflected XSS title parameter in the configuration panel that was vulnerable. Finding this mainly required and modeling JavaScript and forms.

WackoPicko; Multi-step stored XSS We found all the known XSS vulnerabilities [7], except the one requiring flash as we consider it out-of-scope. We also found a non-listed XSS vulnerability in the reflection of a SQL error. Most notably we were able to detect the multi-step XSS vulnerability that no other scanner could. This was thanks to both inter-state dependency tracking and handling workflows. We discuss this in more detail in the case study in Section 5.4.1.

WordPress; Stored and reflected XSS The admin can search for nearby events using the admin dashboard. The problem is that the search query is reflected, through AJAX, for the text-to-speech functionality. Finding this requires modeling of both JavaScript events, network requests and forms.

Our scanner also found that by posting comments from the admin panel JavaScript is allowed to run on posts. For this, the scanner must handle the workflows needed to post the comments and the inter-state dependency analysis needed to later find the comment on a post.

5.4 Case Studies

In this section, we present two in-depth case studies of vulnerabilities that highlights how and why our approach finds vulnerabilities the other scanners do not. We base our analysis on server-side traces, containing the executed lines of code, generated from the scanner sessions. By manually analyzing the source code of an application we can determine the exact lines of code that need to be executed for an injection to be successful.

The cases we use are the comment section in WackoPicko and the configuration panel in phpBB. As we have shown, Black Widow can find vulnerabilities in more complex modern web applications. Nevertheless, these cases allow us to limit the number of factors when comparing our approach with the other scanners. Since WackoPicko and phpBB have been used in previous studies [8, 30] they also serve as a level playing field for all scanners.

5.4.1 Comments on WackoPicko

WackoPicko has a previously unsolved multistep XSS vulnerability that no other scanner has been able to find. The difficultly of finding and exploiting is the need for correctly reproducing a specific workflow. After submitting a comment via a form the user needs to review the comment. While reviewing, the user can choose to either delete the comment or add it. If, however, the user decided to visit another page, before adding or deleting, then the review form will be removed and the user will have to resubmit the comment before reviewing it again. Thus, the steps that must be taken are: Find an image to comment on (`view.php#50`, i.e. line 50 in `view.php`), Post a comment (`preview_comment.php#54`), Accept the comment while reviewing (`view.php#53`) In Table 3.12 we note that two scanners are able to find the input but not exploit it.

Both Enemy of the State and Arachni managed to post a comment but neither could exploit the vulnerability. Enemy of the State was able to post a comment containing an empty string but the fuzzing was unsuccessful. Arguably, Arachni made it a bit further since it was able to inject an XSS payload. However, the payload was not detected and reported. Enemy of the State's shortcoming is that it fuzzes the forms independently while Arachni's shortcoming is that it forgets it's own injection.

jÄk and ZAP had problems finding the first step, i.e. viewing the pictures, because the login form breaks the HTML standard by putting a form inside a table [41]. We avoid this by using a modern browser to parse the web page. This allows Black Widow to view the web page as the developer intended, assuming they tested it in a modern browser

Both w3af and Skipfish were able to find the pictures but not able to post the comment. w3af because it could not model the `textarea` in the form. Skipfish, on the other hand, does not have this problem. We believe that Skipfish logged out after seeing the picture but before posting the comment. The data shows that Skipfish does not try to log in multiple times. In comparison, we correctly handle the `textarea` allowing us to post comments. At the same time, we also try to log in multiple times if presented with a login form. This mitigates losing the session forever at an early stage.

To solve this challenge Black Widow needs to combine the modeling of form elements, handle workflows and use inter-state dependency analysis to correctly inject and detect the vulnerability.

Table 3.12: Steps to recreate the vulnerability in WackoPicko. The columns contain the file name and line of code for each step.

Crawler	view.php#50	preview_comment.php#54	view.php#53	Exploit
Arachni	✓	✓	✓	
Enemy	✓	✓	✓	
jÄk				
Skipfish	✓			
w3af	✓			
Widow	✓	✓	✓	✓
ZAP				

5.4.2 Configuration on phpBB

The configuration panel on phpBB has multiple code injection possibilities. To find these the crawler must overcome two challenges. First, to reach the admin panel requires two logins, the first to authenticate as a user and then again, with the same credentials, to authenticate as an administrator. Second, the injected parameter is not reflected on the same page. To detect this injection inter-state dependency analysis is required. The steps needed to find the vulnerability is, log in as admin (`admin/index.php#28`), find the vulnerable form (`admin_board.php#34`), successfully update the database (`admin_board.php#74`) find the reflection (`admin_board.php#34`).

As shown in Table 3.13, none of the other scanners managed to access the configuration panel. This is because phpBB requires a double login. Arachni, jÄk, Skipfish, w3afand ZAP all require user-supplied credentials together with parameters before running. Based on the traces they do not try these credentials on the admin login form, only the first login form. Enemy

of the State, on the other hand, tries the standard username and password `scanner1`. This was enough to log in but it did not manage to log in as an admin.

Our scanner solves the double login by being consistent with the values we submit. This allows us to both authenticate as a user and then also as an admin when presented with the login prompt. After submitting the form in configuration panel with our taint tokens and later revisiting it, we detect the inter-state dependency and can fuzz the source and sink.

Table 3.13: Steps to recreate the vulnerability in phpBB. The columns contain the file name and line of code for each step.

Crawler	admin/ index.php #28	admin_ board.php #34	admin_ board.php #74	admin_ board.php #34	Exploit
Arachni					
Enemy					
jÄk					
Skipfish					
w3af					
Widow	✓	✓	✓	✓	✓
ZAP					

5.5 Features Attribution

In this section, we identify and attribute the key features that contributed to finding the vulnerabilities in the web applications.

In particular, we try to determine the impact of our modeling, traversing and inter-state dependency analysis techniques. Below are the definitions we use in Table 3.14.

Modeling Modeling is considered to contribute if a combination of HTML forms and JavaScript events were used to find the code injection.

Traversal Workflow traversal contributes if the point of injection depends on a previous state. This could, for example, be a form submission, a click of a button or some other DOM interaction.

Inter-state dependency A code injection is defined to need inter-state dependency analysis if the point of reflection is different from the point of injection.

Table 3.14 shows the 25 unique code injections from the evaluation. Of these, modeling contributed to 4, workflow traversal contributed to 9, and inter-state dependency analysis contributed to 13. In total, at least one of them was a contributor in 16 unique injections. The remaining 9 were usually simpler. Four of them were from WackoPicko where the results of injection were directly reflected. SCARF had 3 directly reflected injections and osCommerce had 2. It is clear, especially for unique vulnerabilities, that modeling, workflow traversal and inter-state dependency analysis plays an important role in detecting stored XSS vulnerabilities.

Table 3.14: For each of the vulnerabilities we note contributing features, i.e. modeling, workflow reproduction or inter-state dependency (ISD) analysis. We also present if they were uniquely detected by Black Widow.

Id	Application	Description	Model	Workflow	ISD	Unique
1	HotCRP	User upload	✓	✓		✓
2	osCommerce	Review rating				✓
3	osCommerce	Tax class				✓
4	phpBB	Admin ranks			✓	✓
5	phpBB	Configuration			✓	✓
6	phpBB	Site name			✓	✓
7	PrestaShop	Date	✓	✓		✓
8	SCARF	Add session		✓	✓	✓
9	SCARF	Comment		✓	✓	✓
10	SCARF	Conference name				
11	SCARF	Edit paper		✓	✓	✓
12	SCARF	Edit session				
13	SCARF	Delete comment		✓	✓	✓
14	SCARF	General options				
15	SCARF	User options			✓	
16	Vanilla	Comment draft			✓	✓
17	Vanilla	Locale			✓	✓
18	Vanilla	Title banner	✓			✓
19	WackoPicko	Comment				
20	WackoPicko	Multi-step		✓	✓	✓
21	WackoPicko	Picture				
22	WackoPicko	Search				
23	WackoPicko	SQL error				
24	WordPress	Comment		✓	✓	✓
25	WordPress	Nearby event	✓	✓	✓	✓

5.6 Missed by Us

Out of the 25 unique injections found by all scanners, we also find all 25. There was, however, an instance where Arachni found a vulnerability by injecting a different parameter than we did. This does not constitute a unique vulnerability due to our clustering, which we explain in Section 4.4. On

SCARF, input elements can be dynamically generated by adding more users. The input names will simply be `1_name`, `2_name`, etc. Arachni managed to add multiple users by randomizing email addresses. Since our crawler is focused on consistency, we do not generate valid random email addresses and could therefore not add more than one user.

The drawback, as we have discussed is that is it easier to lose the state if too much randomness is used. A possible solution to this could be to keep two sets of default values and always test both when possible. There is still the risk that using multiple users can result in mixing up the state between them. It would also introduce a performance penalty as multiple submissions for each form would be required.

The w3af scanner was able to find a reflected version of a vulnerable parameter that we considered to be stored. In this particular case on SCARF, it was possible to get a direct reflection by submitting the same *password* and *retype password* in the user settings. This is what w3af did. Our scanner injected unique values into each field, resulting in an error without reflection, however, the fields were still stored. Inter-state dependency analysis was used to detect these stored values when revisiting the user settings.

Further possible improvements include updating our method for determining safe requests and more robust function hooking. A machine learning approach, such as Mitch [6], could be used to determine if a request can be considered safe. The function hooking could be done by modifying the JavaScript engine instead of instrumenting JavaScript code.

5.7 Vulnerability Exploitability

For the six new vulnerabilities, we further investigate the impact and exploitability. While all of these vulnerabilities were found using an admin account in the web application, the attacker does not necessarily need to be an admin. In fact, XSS payloads executed as the admin gives a higher impact as the JavaScript runs with admin privileges. What the attacker needs to do is usually to convince the admin to click on a link or visit a malicious website, i.e. the attacker does not require any admin privileges. Although, there might be an XSS vulnerability in the code, i.e. user input being reflected, there are orthogonal mitigations such as CSRF tokens and CSP that can decrease the exploitability.

To exploit the HotCRP vulnerability the attacker would have to guess a CSRF token, which is considered difficult. Similarly, PrestaShop has a persistent secret in the URL which would have to be known by the attacker. One of the WordPress vulnerabilities was a self-XSS, meaning the admin would need to be convinced to, in this case, input our payload string, while the other

one required a CSRF token. Finally, osCommerce required no CSRF tokens making it both high impact and easy to exploit.

5.8 Coordinated Disclosure

We have reported the vulnerabilities to the affected vendors, following the best practices of coordinated disclosure [15]. Specifically, we reported a total of six vulnerabilities to HotCRP, osCommerce, PrestaShop and WordPress.

So far our reports have resulted in HotCRP patching their vulnerability [24]. A parallel disclosure for the same vulnerability was reported to PrestaShop and is now tracked as CVE-2020-5271 [1]. Due to the difficulty of exploitation, WordPress did not consider them vulnerabilities. However, the *nearby event* vulnerability is fixed in the latest version. We have not received any confirmation from osCommerce yet.

6 Related Work

This section discusses related work. Automatic vulnerability scanning has been a popular topic due to its complexity and practical usefulness. This paper focuses on blackbox scanning, which requires no access to the application's source code or any other input from developers. We have evaluated our approach with respect to both community-developed open-source tools [28, 34, 36] and academic blackbox scanners [8, 30]. There are also earlier works on vulnerability detection and scanning [2, 10, 11, 17, 23, 35]. While we focus on blackbox testing, there is also progress on whitebox security testing [13, 18, 22, 25, 39].

As previous evaluations [3, 9, 29, 37, 40] show, detecting stored XSS is hard. A common notion is that it is not the exact payload that is the problem for scanners but rather crawling deep enough to find the injections, as well as, model the application to find the reflections. Similar to our findings, Parvez et al. [29] note that while some scanners were able to post comments to pictures in WackoPicko, something which requires multiple actions in sequence, none of them was able to inject a payload.

We now discuss work that addresses server-side state, client-side state, and tracking data dependencies.

Server-side state Enemy of the State [8] focuses on inferring the state of the server by using a heuristic method to compare how requests result in different links on pages. Black Widow instead takes the approach of analyzing the navigation methods to infer some state information. For example, if the

previous edge in the navigation graph was a form submission then we would have to resubmit this form before continuing. This allows us to execute sequences of actions without fully inferring the server-side state.

One reason many of the other scanners pay little attention to server-side state is to prioritize performance from concurrent requests. Skipfish [42] is noteworthy for its high performance in terms of requests per second. One method they use to achieve this is making concurrent requests. Concurrent requests can be useful in a stateless environment since the requests will not interfere with each other. ZAP [28], w3af [34] and Arachni [36] take the same approach as Skipfish and use concurrent requests in favor of better state control. Since our traversing method relies on executing a sequence of possibly state-changing action we need to ensure that no other state-changing requests are sent concurrently. For this reason, our approach only performs actions in serial.

Client-side state jÄk considers client-side events to improve exploration. The support for events is however limited, leaving out such events as form submission. While other scanners like Enemy of the State, w3af, and ZAP execute JavaScript, they do not model the events. This limits their ability to explore the client-side state. As modern applications make heavy use of JavaScript, Black Widow offers fully-fledged support of client-side events. In contrast to jÄk, Black Widow models client-side events like any other navigation method. This means that we do not have to execute the events in any particular order which allows us to chain them with other navigations such as form submissions.

Tracking data dependencies Tracking payloads is an important part of detecting stored XSS vulnerabilities. Some scanners, including Arachni, use a session-based ID in each payload. Since the ID is based on the session this can lead to false positives as payloads are reused for different parameters. jÄk and Enemy of the State use unique IDs for their payload but forgets them on new pages. w3af uses unique payloads and remembers them across pages. ZAP uses a combination in which a unique ID is sent together with a generic payload but in separate requests. This works if both the ID and payload are stored on a page. In addition to using unique IDs for all our payloads, Black Widow incorporates the inter-state dependencies in the application to ensure that we can fuzz the correct input and output *across* different pages.

LigRE [10], and its successor KameleonFuzz [11] use a blackbox approach to reverse engineering the application and apply a genetic algorithm to modify the payloads. While they also use tainting inside the payloads to track

them, we use plaintext tokens to avoid filters destroying the taints. While Black Widow works on live applications, KameleonFuzz requires the ability to reset the application. Unfortunately, neither LigRE nor KameleonFuzz are open-source, which has hindered us from their experimental evaluation.

7 Conclusion

We have put a spotlight on key challenges for crawling and scanning the modern web. Based on these challenges, we have identified three core pillars for deep crawling and scanning: navigation modeling, traversing, and tracking inter-state dependencies. We have presented Black Widow, a novel approach to blackbox web application scanning that leverages these pillars by developing and combining augmented navigation graphs, workflow traversal, and inter-state data dependency analysis. To evaluate our approach, we have implemented it and tested it on 10 different web applications and against 7 other web application scanners. Our approach results in code coverage improvements ranging from 63% to 280% compared to other scanners across all tested applications. Across all tested web applications, our approach improved code coverage by between 6% and 62%, compared to the *sum* of all other scanners. When deployed to scan for cross-site scripting vulnerabilities, our approach has featured no false positives while uncovering more vulnerabilities than the other scanners, both in the reference applications, i.e. phpBB, SCARF, Vanilla and WackoPicko, and in production software, including HotCRP, osCommerce, PrestaShop and WordPress.

Acknowledgment

We would like to thank Sebastian Lekies for inspiring discussions on the challenges of web scanning. We would also like to thank Nick Nikiforakis and the reviewers for their valuable feedback. This work was partially supported by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

Bibliography

[1] CVE-2020-5271. Available from MITRE, CVE-ID CVE-2020-5271., Apr. 20 2020.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE, 2008.

[3] 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. IEEE, 2010.

[4] C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 81–90. ACM, 2009.

[5] Bugcrowd. The State of Crowdsourced Security in 2019. `https://www.bugcrowd.com/`, 2020.

[6] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei. Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 528–543. IEEE, 2019.

[7] A. Doupé. Wackopicko, 2018.

[8] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security Symposium 12*, pages 523–538, 2012.

[9] A. Doupé, M. Cova, and G. Vigna. Why johnny canâĂŹt pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131. Springer, 2010.

[10] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz. Ligre: Reverse-engineering of control and data flow models for black-box xss detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 252–261. IEEE, 2013.

[11] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 37–48, 2014.

[12] Facebook. A Look Back at 2019 Bug Bounty Highlights. `https://www.facebook.com/notes/facebook-bug-bounty/a-look-back-at-2019-bug-bounty-highlights/3231769013503969/`, 2020.

[13] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, volume 58, 2010.

[14] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, RFC Editor, June 2014.

[15] Google. Project zero: Vulnerability disclosure faq, 2019.

[16] Google. Vulnerability Reward Program: 2019 Year in Review. `https://security.googleblog.com/2020/01/vulnerability-reward-program-2019-year.html`, 2020.

[17] W. G. Halfond, S. R. Choudhary, and A. Orso. Penetration testing with improved input vector identification. In *2009 International Conference on Software Testing Verification and Validation*, pages 346–355. IEEE, 2009.

[18] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52, 2004.

[19] S. Idrissi, N. Berbiche, F. Guerouate, and M. Shibi. Performance evaluation of web application security scanners for prevention and protection

against vulnerabilities. *International Journal of Applied Engineering Research*, 12(21):11068–11076, 2017.

[20] InfoSecurity. XSS is Most Rewarding Bug Bounty as CSRF is Revived. https://www.infosecurity-magazine.com/news/xss-bug-bounty-csrf-1-1-1-1/, 2019.

[21] S. Innovation. Google Awards $1.2 Million in Bounties Just for XSS Bugs. https://blog.securityinnovation.com/google-awards-1.2-million-in-bounties-just-for-xss-bugs, 2016.

[22] N. Jovanovic, C. Kruegel, and E. Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[23] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 247–256, 2006.

[24] E. Kohler. Correct missing quoting reported by Benjamin Eriksson at Chalmers. https://github.com/kohler/hotcrp/commit/81b7ffee2c5bd465c82acf139cc064daacca845c, 2020.

[25] X. Li, W. Yan, and Y. Xue. Sentinel: securing database from logic flaws in web applications. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 25–36, 2012.

[26] A. Mesbah, E. Bozdag, and A. Van Deursen. Crawling ajax by inferring user interface state changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134. IEEE, 2008.

[27] H. NikÅąiÄĞ. Wget - gnnu project, 2019.

[28] OWASP. Owasp zed attack proxy (zap), 2020.

[29] 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. IEEE, 2015.

[30] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *International Symposium on Recent Advances in Intrusion Detection*, pages 295–316. Springer, 2015.

[31] A. Petukhov and D. Kozlov. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. *Computing Systems Lab, Department of Computer Science, Moscow State University*, pages 1–120, 2008.

[32] PortSwigger. Burp Scanner - PortSwigger. `https://portswigger.net/burp/documentation/scanner`, 2020.

[33] D. Rethans. Xdebug - debugger ad profiler tool for php, 2019.

[34] A. Riancho. w3af - open source web application security scanner, 2007.

[35] T. S. Rocha and E. Souto. Etssdetector: A tool to automatically detect cross-site scripting vulnerabilities. In *2014 IEEE 13th International Symposium on Network Computing and Applications*, pages 306–309, Aug 2014.

[36] Sarosys LLC. Framework - arachni - web application security scanner framework, 2019.

[37] L. Suto. Analyzing the accuracy and time costs of web application security scanners. *San Francisco, February*, 2010.

[38] The OWASP Foundation. Owasp top 10 - 2017, 2017. `https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf`.

[39] A. Vernotte, F. Dadeau, F. Lebeau, B. Legeard, F. Peureux, and F. Piat. Efficient detection of multi-step cross-site scripting vulnerabilities. In A. Prakash and R. Shyamasundar, editors, *Information Systems Security*, pages 358–377, Cham, 2014. Springer International Publishing.

[40] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 566–571. IEEE, 2009.

[41] WHATWG. Html standard, 2019.

[42] M. Zalewski. Skipfish, 2015.

Appendix

3..1 Scanner configuration

3..1.1 Arachni

The following command was used to run Arachni.

```
arachni [url] --check=xss* --browser-cluster-pool-size=1 --plugin?
    autologin:url=[loginUrl],parameters="[userField]=[username]&[
    passField]=[password]",check="[logout string]}
```

3..1.2 Black Widow

The following command was used to run Black Widow.

```
python3 crawl.py [url]
```

3..1.3 Enemy of the State

First we changed the username and password in the web application to *scanner1* then we ran the following command.

```
jython crawler2.py [url]
```

3..1.4 jÄk

We updated the `example.py` file with the URL and user data.

```
url = [url]
user = User("[sessionName]", 0, url, login_data = {"[userField]": "[
    username]", "[passField]": "[password]"}, session="ABC")
```

3..1.5 Skipfish

The following command was used to run Skipfish.

```
skipfish -uv -o [output]
         --auth-form [loginUrl]
         --auth-user-field [userField]
         --auth-pass-field [passField]
         --auth-user [username]
         --auth-pass [password]
         --auth-verify-url [verifyUrl]
         [url]
```

3..1.6 w3af

For w3af we used the following settings, *generic* and *xss* for the audit plugin, *web_spider* for crawl plugin and *generic* (with all credentials) for the auth plugin.

3..1.7 Wget

The following command was used to run Wget.

```
wget -rp -w 0 waitretry=0 -nd --delete-after --execute robots=off [url
    ]
```

3..1.8 ZAP

For ZAP we used the *automated scan* with both *traditional spider* and *ajax spider*. In the *Scan Progress* window we deactivated everything that was not XSS. Similar to Enemy of the State, we changed the credentials in the web application to the scanner's default, i.e. *ZAP*.