Towards automated discovery of access control vulnerabilities

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

L.D.C. van der Poel
born in Vlaardingen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

DongIT B.V.
Schipholweg 103
Leiden, the Netherlands
https://www.dongit.nl/

Towards automated discovery of access control vulnerabilities

Author: L.D.C. van der Poel
Student id: 4958802

Abstract

This thesis is a research into developing a methodology and implementation of automated gray-box Broken Access Control Scanning (BACS) in web applications. Broken access controls take first place in the OWASP Top Ten Web Application Security Risks 2021. The need for this research comes from the observation that testing for broken access controls in web applications is labor-intensive, time-consuming, and error-prone. Therefore, security researchers require a modern methodology and toolset for exhaustively discovering access control vulnerabilities in web applications.

The posited hypothesis is that the contextual awareness required for access controls can be achieved by assuming that users are only authorized to perform actions accessible via the UI for that particular user. The methodology developed in this research consists of four phases: 1) A crawl phase where an application is crawled as multiple users. 2) A request selection phase, where potentially vulnerable requests are selected. 3) A request replay phase, where selected requests are replayed in the session context of another user. 4) A response comparison phase to identify whether an access control vulnerability has occurred. An implementation is provided and evaluated during web application penetration tests of DongIT. The results show that critical and structural access control issues can be identified when all four stages are completed. However, the intricacies of web applications often pose challenges for one or more of the four stages. From the results, it is concluded that the BACS methodology is a viable strategy and a valuable tool in the toolbelt of a security tester.

Thesis Committee:

Official supervisor:	Prof. Dr. M. Conti, Cybersecurity Research Group, TU Delft
Daily supervisor:	Dr. A. Zarras, Cybersecurity Research Group, TU Delft
Company supervisor:	Drs. W.S. van Dongen, DongIT
Committee member:	Dr. T. Durieux, Software Engineering Research Group, TU Delft

Preface

Dear reader,

This thesis is a research into tackling one of the most pervasive challenges in offensive web application security: Testing for access control vulnerabilities. The idea for this research sprang forth from my own work as a penetration tester. A glaring issue with testing for access controls is that it is time-intensive and challenging to automate. With time being a scarce resource, the lack of automated tooling leads to security researchers focusing on the most likely issues, resulting in missing critical access control issues.

From personal knowledge of the industry, I knew that no public automated non-whitebox tooling existed for testing access controls. Subsequent exploration of published academic research further confirmed that this remains an unexamined topic. When the Open Web Application Security Project (OWASP) Top Ten Web Application Security Risks came out later in 2021, they ranked Broken Access Control as the number one vulnerability. A better endorsement for the necessity of this research could not be given. I set out to find a more modern, automated, and standardized methodology for discovering broken access control.

For supporting and mentoring me during this arduous but rewarding process, I wish to thank three people. I wish to thank Apostolis Zarras, my daily supervisor, for his understanding and for giving me the time I needed. I wish to thank Wouter van Dongen, for sparring with me about web application hacking, and allowing me to be flexible in my work. And lastly, I wish to thank Tessa Slim for all her support and making the thesis-covid-lockdowns not just bearable, but memorable.

L.D.C. van der Poel
Delft, the Netherlands
September 11, 2022

Contents

Chapter 1

Introduction

Access control vulnerabilities have risen to first place in the Open Web Application Security Project (OWASP) Top 10 2021, kicking injection-related vulnerabilities from their throne [41]. Broken access control currently trumps all other web application vulnerability types in a combined metric of impact and likelihood. However, strikingly, while there is an abundance of research into black-box and gray-box detection of vulnerability types such as SQL injection [26] and cross-site scripting [20], the academic literature on automated discovery of broken access controls remains barren [15].

1.1 Problem statement and knowledge gap

People are required to create user accounts on many of the websites they visit. We constantly trust websites to harbor our personal information, trusting that websites strictly separate the contexts between different users. Creating this security boundary is a meticulous process. A single missing or misconfigured line of authorization-related code can be the difference between a secure environment and users taking over each others' accounts. A failure in access control can have a myriad of disastrous consequences, from application-wide cross-site scripting via broken access controls in a WordPress plugin [11], to remote code execution via a Webmin web portal [12], gaining administrative access to a web application of the U.S. Dept Of Defense [24], exfiltrating repository data from GitLab repositories [23], or taking over the chess.com account of World Chess Champion Magnus Carlson [10].

The perpetual increase in web application complexity and the number of web application vulnerabilities has been cited time and again [19, 39, 18]. Despite web application frameworks providing built-in methods for enforcing access controls [28, 16], access control vulnerabilities remain pervasive. This general trend evokes an urgent scientific need for research into the practical detection of access control vulnerabilities as part of the development process. It is from this industry-wide inability to consistently deal with broken access controls that the motivation of this research springs forth. Why is the problem of systematically detecting broken access controls so pervasive, and how can we tackle this problem?

Vulnerabilities can be identified and resolved during three stages of software devel-

opment: Pre-development, development, and post-development. Secure development processes should consider security at each of these stages [31]. Each stage is characterized by a set of advantages and disadvantages. Firstly, before development starts, the initial creation of vulnerabilities can be prevented. This can be achieved in several ways. Training developers in secure coding practices can help developers write secure code and recognize insecure coding behavior [33, 59]. In addition, research has been done into using more secure frameworks [55, 1] and safer languages [27]. The benefit of this approach is that it prevents vulnerabilities from occurring in the first place. Furthermore, both the scientific community and the industry have expressed the importance of integrating security into the design of software and hardware. However, security-by-design does come with challenges [21]. Security-by-design requires better-trained developers, more initial time investment, and more money up-front, and training needs to be routinely reinforced. Moreover, after investments in employees are made, developers can leave an organization.

A second method to reduce the number of vulnerabilities is via detection during the development phase. This can include automated static code analysis [37, 39], source code review [59], dynamically analyzing traffic [63, 13], and mining and creating access control rules [9, 8, 52]. The main benefit is that vulnerabilities are prevented from being pushed to a live production environment and exploited in the wild. Additionally, it saves the costly time of third-party security researchers in the post-development phase. However, it also increases development time and costs, as developers or another internal party need to configure, run, and process the results. These tools often require licenses and specialized knowledge to utilize fully, adding to the wide array of language and tool proficiencies that are already demanded of web developers.

And thirdly, during the post-development phase, any remaining vulnerabilities can be discovered through third-party security research. Although there are several different setups, two notable types of security research are penetration tests and coordinated vulnerability disclosure programs. The first type consists of a typical service agreement between a security company and a development party with an agreed-upon rate [56, 14]. The second consists of a (semi-)open invitation for ethical hackers to find vulnerabilities in a coordinated vulnerability disclosure agreement, often for a monetary reward, goodies, or publicly recorded fame [62, 32]. These ethical hackers, or 'bug bounty hunters', are paid for the vulnerabilities they find, depending on the severity of their findings. Third-party security research is the last line of defense. The main security benefit is that it allows specialized security researchers to scrutinize an application. The main drawback, however, is the increased cost of third-party services and decreased familiarity with the application compared to in-house developers.

The setup of third-party security assessments comes in three types: white-box, gray-box, and black-box. In white-box research, the researching party is provided access to the source code and potentially to the server itself. In gray-box research, researchers are granted one or more sets of credentials, and their IP addresses may be specifically allowlisted from any intrusion detection or protection systems. In black-box research, researchers have no extra knowledge or privileges compared to an actual attacker.

A large body of research exists on preventing and testing access control vulnerabilities in white-box settings [37, 39, 59, 63, 9, 8, 52, 13]. However, preliminary analysis of the

available academic research revealed that very little research exists on gray and black-box access control vulnerability detection. There is, on the other hand, also a plethora of research on detection of other vulnerability types in gray and black-box settings, such as on SQL injections [26], cross-site scripting [20], prototype pollution [3], and black-box vulnerability discovery and vulnerability scanners in general [5, 18, 17]. Research also exists on the detection of logic flaws, which shares common ground with broken access controls [42, 30]. A 2016 meta-analysis corroborates these observations on the discovery and mitigation of SQL injection, cross-site scripting, and logic vulnerabilities [15]. In their research, Deepa and Thilagam found that the only research on gray or black-box access control vulnerability discovery is a 2010 research on parameter tampering [6]. This research uses a combination of client-side JavaScript analysis and intelligent parameter fuzzing to discover logic vulnerabilities. Although this can coincidentally reveal broken access controls, it does not focus on access control issues specifically. Most importantly, it does not consider different user roles when reasoning about access control issues.

A tentative explanation for the lack of research is that access control testing does not lend itself easily to automation. Automated vulnerability scanners often rely on specific application response behavior for identifying issues. For example, a typical telltale of a SQL injection vulnerability is a 500 Internal Server Error response code, SQL error messages, increased response time, or an out-of-bounds connection [25]. For reflected and stored cross-site scripting, an attacker can watch responses for reflected unencoded user input or input reflected inside dangerous locations [58]. For access controls, confirming their existence is relatively easy. Web applications typically respond with 401 Unauthorized, 403 Forbidden, or 404 Not Found status codes or some message inside the response body. However, determining that access controls are broken is challenging. When access controls fail, the application responds normally, indiscernible from the situation where the user is authorized. The main challenge for a broken access control scanning methodology is determining when a successful application response aligns with the intended access control policy. Here lies the crux of the problem: How can the intended access control policy be determined without access to the source code?

1.2 Research question and hypothesization

This thesis builds on the observations that access control vulnerabilities have become a more pressing issue in the last few years, and that insufficient academic exists on automated gray-box testing of access controls. These observations raise the following research questions, which are answered throughout this thesis:

- Why is the lack of automated broken access control scanning, if at all, a problem?
- What challenges does automated broken access control scanning pose?
- What is a viable methodology for automating broken access control scanning?

The research questions are answered throughout this thesis as follows: It is argued that the lack of automated broken access control scanning makes testing complex applications

with multiple user roles impractical. The difficulty of exhaustively testing access controls results in less total coverage of application functionality, leading to security testers missing access control vulnerabilities.

The primary challenge of automated broken access control scanning is identified as the lack of contextual awareness. The context of an application determines what actions users are authorized for, consciously designed by developers. Human testers can often intuit the intentions of those developers, but programs cannot. The hypothesis is posited that in order for automated gray-box access control vulnerability detection to be successful, an interpretable context must first be established to differentiate authorized from unauthorized functionality. The suggested method for establishing such a context is to assume that all functionality a user has access to via the user interface (UI) of a web application is the only functionality for which a user is authorized.

The aforementioned assumption forms the basis of the proposed broken access control scanning (BACS) methodology. The BACS methodology consists of four phases. Firstly, a web application should be crawled for all users that will be examined. Since agnostic web crawling is a research topic unto itself, the choice was made to allow security researchers to select their crawler of choice. A `Mitmdump`[36] HTTP traffic dump of the crawl can be used as input for the BACS pipeline. The second step is the selection phase, where the asymmetric difference of the HTTP traffic dumps is taken for each crawl pair. In simpler terms, only those requests are selected that are encountered via the UI of one user, but not via the UI of another. This selection effectively filters out only those requests that qualify as potential access control vulnerabilities. Thirdly, the selected requests are replayed in the session context of the other user. The responses to these replayed requests are stored along with the responses to the original requests. And fourthly, the responses of the original requests and the responses of the replayed requests are compared. In the implementation, the security researcher is presented with an interactive HTML page containing links to side-by-side views of the responses, as well as the differences in HTTP status codes, the difference ratio in response body contents, and additional information. The HTML page allows security researchers to visually compare request and response pairs, automatically highlighting differences.

The implementation was evaluated during ten web application security assessments for clients of DongIT. The evaluation methodology is qualitative rather than quantitative. Out of those web applications, it was only possible to fully complete all four steps of the BACS methodology in four cases. However, access control vulnerabilities of varying severity were found for each of those four cases. In some cases, this uncovered vulnerabilities not found during the initial manual testing of access controls. However, the implementation also missed vulnerabilities found manually by the security tester. In all cases where the detection of known vulnerabilities was missed, this resulted from incomplete web crawls or the inability to replay requests successfully. High-severity issues were found in two cases.

1.3 Scientific contributions

All in all, this paper makes the following scientific contributions:

1. The academic lack of research into automated gray-box discovery of access control vulnerabilities is addressed. The hypothesis is posited that automated gray-box discovery of access control vulnerabilities is challenging due to the lack of contextual awareness in most scanners.

2. A solution to the contextual awareness problem is proposed. Contextual awareness can be created by assuming that users are only authorized to access functionality that they can access via the UI.

3. The Broken Access Control Scanning (BACS) methodology is introduced to test the proposed contextual awareness hypothesis. An implementation of a broken access control scanner found high-severity vulnerabilities during professional security assessments.

4. The full implementation is planned to be published as free and open-source software (FOSS).

1.4 Thesis structure

Chapter 2 provides background information on authentication, authorization, sessions, access controls, and access control vulnerabilities. An overview of access control vulnerability types is provided. Of these types, a selection is identified that can be tested via the BACS methodology. Lastly, the reader is informed about how access control vulnerabilities are tested in practice. In addition, this section serves the purpose of identifying why the current manual methodology for testing access controls is problematic. In chapter 3, the four-stage Broken Access Control Scanning (BACS) methodology is posited. The implementations of each of these four stages – web crawling, request selection, request replay, and response comparison – are each discussed in separate sections. Chapter 4 presents the evaluation of the implementation, including the evaluation setup and the results. Chapter 5 provides a discussion of the evaluation, providing an interpretation of the results, the implication of the results, discovered limitations, and suggestions for future research. In chapter 6, an overview of related work is presented, being white-box access control vulnerability detection, black and gray-box vulnerability scanning of vulnerabilities in general, and web crawling. Chapter 7 concludes with a reiteration of the thesis in general and the most important findings. A bibliography of mixed scientific and non-scientific resources can be found at the end.

Chapter 2

Background

This chapter will provide background information on authentication, authorization, access control vulnerabilities, and testing for access control vulnerabilities. The first part of the first section discusses authentication and user sessions. Identifying users is a prerequisite for enforcing authorization schemes. The second part discusses the relationship between authorization and access controls. Most importantly, access control vulnerabilities occur when the theoretical authorization scheme does not match the implemented access controls. The second section provides a categorization of access control vulnerabilities into six types: 1) Unauthenticated functionality, 2) Identifier-based functionality, 3) Multistage functionality, 4) Static files, 5) Platform misconfiguration, and 6) Insecure access control methods. Section three shows how access control vulnerabilities are discovered. In addition, a demarcation is given of which access control vulnerability types are targeted via the Broken Access Control Scanning methodology. Afterward, an introduction is given on how access control vulnerabilities are tested for in-practice. Three examples of access control vulnerability testing are provided, accompanied by HTTP request-response pairs. The examples increase in complexity, already introducing some challenges that need to be overcome in the implementation. This section concludes with a reflection on the issues with manual access control testing. Due to the combinatorial explosion resulting from increases in number of functionalities and the number of user roles, exhaustive manual testing is infeasible. For this reason, we must look toward automated broken access control discovery.

2.1 Authentication, authorization, sessions, and access controls

This section discussed the relation between authentication, authorization, sessions, and access controls. Authentication is required to establish the identity of a particular user for the duration of a session. Determining the identifier of a session and any additional session-relevant information is required to replay requests in a different user context successfully. Authorization governs the actions particular identities or identity groups ought to have access to. Access controls are the actual implementations of these envisioned rules. When there is a discrepancy between the envisioned authorizations and the implemented access controls, an access control vulnerability occurs.

2.1.1 Authentication and user sessions

Authentication is concerned with verifying the identity of a user. In web applications, the authentication flow often consists of a server requesting users to provide a username (the identity) and a password (the verification). The authentication may require the user to confirm their identity via a second factor, such as via text message, phone call, or time-based one-time password (TOTP) code. After authentication is completed, the user will receive a session identifier from the application. Most commonly, the session identifier is a session cookie or an `Authorization` bearer header [38]. However, developers are free to implement custom authentication flows and session management schemes. The quirks of the authentication flow and session management system can therefore differ from application to application.

In addition, many session management systems use session-relevant information in each request. Session-relevant information typically consists of data sent with each request, where the data is tied to the session in some manner. The most common example is the cross-site request forgery (CSRF) token [4]. CSRF tokens are generated per session (and sometimes per page). These tokens are sent along with each (data-changing) request to prevent certain types of malicious cross-site requests. However, many other kinds of session-relevant data can be sent, such as ASP.NET ViewStates [35]. And again, developers are free to craft their own session-relevant data flows. These schemes can therefore differ from application to application.

2.1.2 Authorization, access controls, and broken access controls

Authorization is the specification of the access a user has to functionalities. Access controls are the implementations that enforce authorization rules. Access controls are often modeled on either Role-rased access control (RBAC) or attribute-based access control (ABAC), and sometimes relation-based access control (ReBAC).[1]. Role-based access controls authorize users based on their roles, such as ‘Administrator’ or ‘Guest’. Attribute-based access control authorizes users based on user attributes, functionality or object attributes, or environmental attributes. In effect, this allows developers to enforce more fine-grained access controls. Users can be authorized for specific actions instead of requiring the creation of an entirely new role. Relation-based access controls are paradigmatic for social networks. In this case, authorization is granted based on their relationships with other users, such as friends or followers. In practice, developers need not adhere strictly to these access control models but can choose to adopt a mix of them.

Horizontal access controls limit the actions between different users of the same type. For example, in most applications, users should be able to change their own passwords but not those of others. As another example, take an application for a news organization. This news organization employs ‘writer’ roles. Writers are only able to view, modify and delete their own draft work, but not those of others.

Context-dependent access controls limit a user’s actions in relation to a specific application state. Some actions can only be taken in specific orders or when certain conditions have

[1] For example, see the OWASP guidelines on access control implementations [40]

been met. In the case of the news organization, users of the 'reader' role are only allowed to read written submissions that have been published, not before. Similarly, writers may be prevented from modifying their submissions once published.

Access control vulnerabilities can span multiple access control types. In the news organization example, suppose an access control vulnerability allows a writer to modify the published work of not just herself but also of other writers. This constitutes a vulnerability in both the horizontal and context-dependent access controls. Naturally, not all examples allow for easy categorization. However, the three coarse categories of horizontal, vertical, and context-dependent are sufficient for grasping access control vulnerabilities.

2.2 Access control vulnerabilities

Access control vulnerabilities can come in several shapes and forms. The Web Application Hacker's Handbook[2] identifies six types of access control vulnerabilities: 1) Unauthenticated functionality, 2) Identifier-based functionality, 3) Multistage functionality, 4) Static files, 5) Platform misconfiguration, and 6) Insecure access control methods [57, 258-266].

2.2.1 Unauthenticated functionality

Access control vulnerabilities can be coarsely split into six types. Firstly, some functionality is simply left unprotected. Websites may unknowingly expose sensitive endpoints or functionalities to unauthenticated users. For example, developers may unknowingly expose a `/phpinfo` endpoint [44], an administrative dashboard, or vulnerable development functionalities inside a `/vendor/` directory. In some cases, no references to the functionality are present in the application. A `/phpinfo` page could be a remnant of the development phase, forgotten to be removed for the production phase. In other cases, functionality is present for authenticated users in the application. However, missing access control checks allow unauthenticated users to execute this endpoint. Unauthenticated attackers can often find vulnerable endpoints through brute-forcing application endpoints, parameter names, and parameter values. In some cases, functionality is leaked through public or leaked source code or client-side JavaScript code.

2.2.2 Identifier-based functionality

The second class of access control vulnerabilities is the case where access controls rely on a specific identifier to be sent by the client. For example, an endpoint `/download?id=123` ↪ that allows downloading files would likely download the 123th uploaded file. A simple decrementation would download the 122 previously uploaded files. If these are files that the current user should not have access to, this constitutes an access control vulnerability.

Sometimes identifiers can be more difficult to guess. A common identifier type is the universally unique identifier (UUID) [51], such as `7f17f961-143e-4212-b280-6f9a0ae24591`. UUIDs are designed to be generated as identifiers with minimal risk of collision. Although

[2] A staple work in web application penetration testing

UUIDs are practically impossible to guess, they are no substitute for actual access controls. Firstly, according to the relevant RFC, UUIDs are not promised to be cryptographically secure [51]. Furthermore, UUIDs do not have the same security standards as session identifiers. They often do not expire, cannot be refreshed, and are often sent as part of the URL (and are therefore cached, logged, sent via referrer headers, accessible via cross-site scripting, et cetera). In addition, developers can implement functionality that leaks UUIDs, such as through overly verbose API responses. For these reasons, identifier-based functionality should employ access controls even when using hard-to-guess identifiers.

2.2.3 Multistage functionality

Thirdly, vulnerabilities can arise when functionality spreads over multiple stages. This largely coincides with context-dependent access controls. Think, for example, of shopping carts in webshops. When purchasing an item online, it is crucial to enforce the correct order of steps. If a user could first pay and add items to their shopping cart afterwards, a webshop would quickly go out of business. These kinds of vulnerabilities are sometimes referred to as logic flaws [42]. Due to their abstract nature, they vary greatly from application to application.

2.2.4 Static files

Fourthly, developers may leave sensitive static files unprotected. This relates partly to unauthenticated functionality. However, in this case, a file is requested directly from the filesystem. This often bypasses any routing and web application logic in place. The severity of the exposed static files varies. In some cases, the impact is relatively mild, such as `composer.json` ↪ files disclosing PHP dependencies and versions [7]. In other cases, source code backups such as `/backup.tar.gz` can expose the application source code and potentially application secrets.

Furthermore, even intentionally stored files are not always properly access controlled. An application that generates and emails PDF invoices may offer those files to the user as `/invoices/49b27d8b3f12e25ad685b992ac4b4546.pdf`, where the filename is an MD5 hash of a UNIX timestamp. Although this may look secure to the naked eye, without proper access control, it is, in fact, easily enumerable.

2.2.5 Platform misconfigurations

Fifthly, misconfiguration at the web server or application platform layers can lead to access control bypasses. For example, an `.htaccess` file can prohibit direct file access from `/uploads` ↪ `/` [2]. But if the rule is not applied recursively, or a more permissive `.htaccess` file exists in a lower directory, then files in directories under `/uploads` may still be directly accessible.

In other cases, platform-level configurations can prohibit users from sending specific request methods to certain endpoints. For example, an endpoint for setting user roles may be correctly configured to block POST requests but may allow PUT and DELETE requests, unbeknownst to the developer. On a network level, a reverse proxy or firewall may only allow requests from allowlisted IP addresses to sensitive endpoints, such as development

or administrative functionality. However, in some cases, user-controllable headers, such as `X-Real-Ip: 127.0.0.1`, can be used to bypass these restrictions.

2.2.6 Insecure access control methods

Sixthly and lastly, applications may use insecure access control methods. They may enforce access control based on data that is under control of the attacker. A website may set a cookie with the value `YWRtaW49ZmFsc2U=`, which is the base64 encoding of `admin=false`. An attacker can modify this value, re-encode the data, and submit it to the server. Similar access controls based on client-controlled data, such as parameters, cookies, headers, and even location, can often be controlled by the user.

2.3 Testing for access control vulnerabilities

Different access control vulnerability types require different detection methods. This research focuses on a specific technique for detecting access control issues. Therefore, only a subsection of the aforementioned vulnerability types can be discovered using this technique. This section first describes which vulnerabilities can be found using the BACS methodology. Secondly, some examples are given of how these access controls are typically manually tested. This section also introduces some of the more apparent challenges in successfully replaying requests in another user's session context. Lastly, an argument is provided for why manual testing for access controls is problematic, introducing the need for automated broken access control scanning.

2.3.1 Access control vulnerabilities targeted by the BACS methodology

The primary assumption in the BACS methodology is that functionality can be determined to be authorized or unauthorized depending on whether a user can reach this functionality via the user interface (UI). Therefore, vulnerabilities can only be found for endpoints, files, and parameters that can be accessed through the UI by at least one user. This requirement entails that certain vulnerability types or subgroups of vulnerability types cannot be found with the BACS methodology. Most importantly, the BACS methodology will not find access control issues that can only be found through brute-force attacks. For example, old backup files such as `/backup.tar.gz` that can only be found through brute-forcing file paths will not be detected. The same applies to guessing unprotected functionality, API endpoints, and `action` values. As a result, some vulnerabilities within type one (unauthenticated functionality) access control vulnerabilities will not be found. Other vulnerabilities within this type can still be found as long as the functionality missing access controls is present for at least one user.

Similarly, static files missing access controls can be discovered as long as a reference to those files exists in the UI. A `backup.tar.gz` file will likely not be detected. However, a file such as `/uploads/invoice-bob-3.pdf` can still be access control tested if referenced in the UI. The BACS methodology can also find type two (identifier-based) access control vulnerabilities. If a crawl is performed using two users, and user Bob can download a file with

`/download?id=123`, but user Alice can only download a file with `/download?id=124`, then during the replay phase, a session authenticated with Alice can try to retrieve the file `/download?id=123`. If the same content is returned, this signifies that Alice can download Bob's files. The vulnerability types of platform misconfigurations and insecure access control methods are vast. This vulnerability type is not tested for explicitly by the BACS methodology, but may still be detected in some cases. For example, the original request headers are sent during the replay phase. If a custom header is sent during the crawl of Bob with `X-Auth: Bob`, and this request is replayed in the session of Alice, it may be discovered that the `X-Auth` header functions as an insecure access control method.

For all subsequent sections and chapters, 'access control vulnerabilities' refer to the kind of vulnerabilities targeted by the BACS methodology unless specified otherwise.

2.3.2 Manual testing

When testing for access control vulnerabilities, the goal is to reperform an action from one user context within the context of another user. When a user authenticates to a web application, a session is created for that user. In effect, a session is a temporary and particular instance of a user context. Therefore, access controls can be tested by reperforming an action from one user session in the user session of another user. Three examples will be provided to show access control tests are performed in practice, with increasing complexity. The users Alice and Bob are used to mark the different user sessions.

Case one: simple requests with simple sessions.

First of all, when a session is identified via a static session identifier without any other authorization mechanisms or application states, testing access control issues is very straightforward. A security tester can replace the original session identifier with a session identifier from another user, resend the response, and observe the differences in the response. For example, in Listing 2.1 the user Alice is an administrator and updates the role of the user Charlie to `admin`. This request is met with a `200 OK` response in Listing 2.2. In Listing 2.3, the request is replayed with the session cookie of Bob. This time, the server responds with a `403 Forbidden` response.

```
1 PUT /management/users/updateRole HTTP/1.1
2 Host: example.com
3 Content-Type: application/x-www-form-urlencoded
4 Cookie: session_id=2U4miTiZK-ALICE; remember_me=true
5
6 username=charlie&role=admin
```

Listing 2.1: HTTP request to update the role of a user (performed by Alice).

```
1 HTTP/1.1 200 OK
2 Connection: close
```

Listing 2.2: Response. Request authorized.

```
PUT /management/users/updateRole HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Cookie: session_id=h4Sy8iuV-BOB; remember_me=true

username=charlie&role=admin
```

Listing 2.3: HTTP request to update the role of a user (performed by Bob).

```
HTTP/1.1 403 Forbidden
Connection: close
```

Listing 2.4: Response. Request forbidden.

From the responses, it is obvious that Bob is not authorized to send the same request as Alice. Even in this simple example, two challenges can already be identified. Firstly, how is the session token identified? Although a human can intuitively guess that cookies named `session_id`, `JSESSIONID`, `PHPSESSID` and `asp.net_sessionid` are likely session identifiers, it is not straightforward to program a computer to do so. Some web frameworks, such as Laravel, provide a session cookie name that contains the web application's name, such as `actester_session` [29]. Frameworks can, of course, create any kind of cookie parameter name, including a string of random characters.

A further complicating factor, for both machines and humans, is the case in which multiple session identifiers or session-relevant values are provided. These may be for different parts of the application, or they may be non-functional development remnants. Correctly replaying requests is one of the challenges that will be addressed in the chapter on system design (section 3.3).

A second challenge is to determine the differences between responses. In the examples above, the two responses are easily distinguished. However, web applications often return `200 OK` HTTP status codes for both authorized and non-authorized requests, instead of returning `403 Forbidden` explicitly. A message informing the user that the action was forbidden may be included in the HTML response, or it may not be included at all. Additionally, responses need not always be static. If Alice sends the same request multiple times, the content of the responses can differ. Although the contents typically only differ slightly, such as a different server timestamp or a different CSRF token, the information informing a user that the request was blocked can also consist of only a few characters. Determining the difference between authorized and non-authorized responses is not a trivial task. Like replaying requests, comparing responses is one of the challenges that is addressed in the chapter on system design (section 3.4).

Case 2: CSRF tokens and other session-relevant parameters.

Session-relevant data, such as cross-site request forgery (CSRF) tokens, complicate the ability to replay requests correctly. Web applications block requests with incorrect or missing CSRF tokens if implemented correctly.

CSRF token handling can be implemented in multiple ways. Firstly, the easiest case is when one CSRF token is generated for all forms per user session. The second and more

difficult scenario is when a CSRF token is generated for each form. For example, making a PUT request to `/management/users/updateRole` first requires making a GET request (Listing 2.5) to the page that hosts the original form. Often, this is a request to the same endpoint. The CSRF token can then be extracted from the response (Listing 2.6). The token can then be used in added to the request at the correct parameter (Listing 2.7). In this case, Bob can only test access controls for requests where the original form is on a page that Bob has access to. If, on the other hand, Bob does not have access to such a page, Bob cannot obtain a valid CSRF token for the request. As a result, the CSRF token inadvertently blocks access control issues. CSRF tokens are not designed to protect against access control violations (except for cross-site request forgery) and should, therefore, not be seen as a proper access control.

There may be an arbitrary amount of custom parameters, headers, and cookies tied to the session token. Some of these are known and documented, such as the earlier stated 'ViewStates'. Others are entirely custom and sporadic. Testing for broken access controls requires the tester to be mindful of these values.

```
GET /management/users/updateRole HTTP/1.1
Host: example.com
Cookie: session_id=2U4miTiZK-ALICE; remember_me=true
```

Listing 2.5: HTTP request to page where user roles can be updated. (performed by Alice).

```
HTTP/1.1 200 OK
Connection: close

Content-Length: 1234

<!DOCTYPE html>

<html lang="en-GB">
    <head>
    <meta> name="csrf_token" content="qxBPTHLc-EntdvOsil7oeHeYORLCwwJoqg3k">
    <title>Update user roles</title>
    </head>
    <body>
        ...REDACTED...
    </body>
</html>
```

Listing 2.6: Response containing a CSRF token.

```
PUT /management/users/updateRole HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Cookie: session_id=2U4miTiZK-ALICE; remember_me=true

username=charlie&role=admin&_csrf_token=qxBPTHLc-EntdvOsil7oeHeYORLCwwJoqg3k
```

Listing 2.7: HTTP request to update the role of a user but with a CSRF token (performed by Alice).

```
HTTP/1.1 200 OK
Connection: close
```

Listing 2.8: Response. Request authorized.

Case 3: State-dependent requests.

Modern web applications are dynamic. Web applications can give different responses to the same request and the same responses to different requests. Take, for example, a note-taking application. A note-taking application may only allow the creation of notes with unique names. For example, if Alice creates a new note called 'Shopping list', the first request (Listing 2.9) may succeed (Listing 2.10), but the second request (Listing 2.11) may fail (Listing 2.12). Similarly, a request to delete an entity often only succeeds the first time but not the second.

Some applications give different responses to the same request. For example, the note-taking application may allow users to browse through their notes via subsequent GET requests to `/note/next`. Although the endpoint is the same, the response is different each time.

```
POST /notes/new HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Cookie: session_id=2U4miTiZK-ALICE; remember_me=true

name=Shopping+list&content=<p><ul><li>butter</li><li>cheese</li><li>eggs</li></ul></p>
```

Listing 2.9: HTTP request to add a new note (performed by Alice).

```
HTTP/1.1 201 Created
Connection: close
```

Listing 2.10: Response. Request authorized.

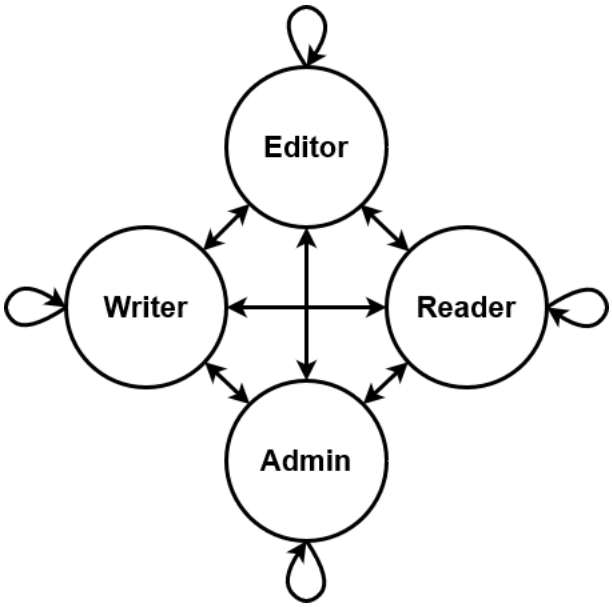
```
POST /notes/new HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Cookie: session_id=2U4miTiZK-ALICE; remember_me=true

name=Shopping+list&content=<p><ul><li>butter</li><li>cheese</li><li>eggs</li></ul></p>
```

Listing 2.11: Repeating the same request to add a new note (performed by Alice).

```
HTTP/1.1 409 Conflict
Connection: close
```

Listing 2.12: Response. Request authorized.

Figure 2.1: Access control graph 'Publishing Company' application, no unauthenticated user, $\frac{4*(4-1)}{2}+4=10$ vertices.



2.3.3 Issues with manual access control testing

The primary issue with manual access control testing is that it cannot properly scale. As an application becomes more complex, more functionality and user roles are introduced. Mature and complex web applications have hundreds or thousands of unique actions. Furthermore, to test vertical access controls, each functionality that each role has access to (assuming a role-based access control scheme) must be checked against each other role. In addition, to find horizontal access control issues, each functionality for each role must be checked between two users of the same role. These relations can be represented as a undirected complete graph with self-loops, where $node1 \rightarrow node2$ represents vertical access controls, and $node1 \rightarrow node1$ horizontal access controls. For example, see the access control graph of 'Publishing Company' web application in Figure 2.1. Taking N as the number of roles (nodes), there are in total $\frac{N*(N-1)}{2}+N$ different relations (edges) to check. For example, if there are only four roles, a total number of $\frac{4*(4-1)}{2}+4=10$ relations must be checked. This does not yet include the unauthenticated user, which has a one-way relationship with all other roles. In practice, the total number of testable relations can be reduced due to the transitive nature of some roles. Most prominently, all actions an unauthenticated user is authorized for, a regular authenticated user is typically also authorized for. Similarly, all actions a regular authenticated user is authorized for, an administrator is typically also authorized for.

The total search space truly explodes when examining attribute-based access control models. The complexity of the access control model increases exponentially with the number of attributes. The total number of possible access control combinations is the power set

of all access control attributes. For example, with access control attributes x, y, and z, users can have any of the eight subsets in the set $\{\varnothing, \{x\}, \{y\}, \{z\}, \{x,y\}, \{x,z\}, \{y,z\}, \{x,y,z\}\}$ The size of a power set is 2^n. Strictly speaking, each subset can be considered a unique role. Each combination of attributes impacts the functioning of the web application. Therefore, exhaustively testing the relations between all 2^n 'roles' becomes impractical, even with automation. Twenty attribute-based access control rules becomes $2^20 = 1048576$ roles, which is a staggering total of $\frac{1048576*(1048576-1)}{2} + 1048576 = 5.50 * 10^{11}$ relations.

Nevertheless, even in a favorable scenario where effectively only four relations need to be examined for five users (assuming fully transitive authorizations), performing exhaustive access control checks for all functionality endpoints can become laborious and time-intensive.

Chapter 3

System design

This chapter discusses the broken access control scanning methodology and its implementation. The Broken Access Control Scanning (BACS) workflow consists of four parts. First is the crawling phase. At least two web crawls are performed as different users. The web crawl can be performed using any third-party web crawler. The traces ('flow') of the request-response pairs ('exchanges') are captured and stored. The second phase is the request selection phase. The requests of two flows are compared, and the asymmetric difference is taken. During the third phase, the selected requests are replayed in the session context of the other user. The new AC-tested exchanges are stored jointly with the original exchanges. Lastly, during the fourth phase, the responses of the original and AC-tested exchanges are compared. The tester is informed about potential access control vulnerabilities. She is provided with an HTML interface to inspect the results interactively. The following sections discuss the implementation details of each of these four steps in more detail.

3.1 Web crawling

In order to capture exchange flows, the security tester is required to configure their desired web crawler to be able to authenticate with credentials. Arguably, the crawling phase is the most critical step. Only requests that can be reached during the crawl can actually be tested. Ultimately, any crawl-based vulnerability scanner is limited by the code coverage the web crawler can achieve [17]. However, due to the variety and complexity of web applications, crawling arbitrary web applications is challenging [34, 43]. There are three primary considerations when choosing or developing a web application crawler. Firstly, which authentication methods are supported by the crawler? Secondly, how well does the crawler handle various HTML elements and JavaScript? Thirdly, how does the crawler handle edge cases, such as volatile data? During this research, the choice was made to use Burp Suite Professional [46], which is designed explicitly in an application-agnostic manner. Regardless, even such a state-of-the-art crawler encounters limitations.

Ideally, a web crawler automatically performs the authentication process when given a set of credentials. However, authentication mechanisms of web applications come in many different forms. Most web applications provide simple username-password authentication,

but other authentication schemes exist. A modern and widely used authentication scheme is Single Sign-On (SSO). SSO has different implementations, such as Security Assertion Markup Language (SAML), Auth0, or OpenID. In addition, authentication features such as CAPTCHA and multi-factor authentication may be in place. These again can come in various forms, such as TOTP (Time-based, One-Time Password), text-based, or email-based.

Furthermore, sessions can be tracked in multiple ways, most commonly via session cookies or Authorization bearer headers. However, even these come in different forms, as was also shown in chapter 2. Some applications send cookies back with each request, which are expected to be returned with subsequent requests. Authorization bearers can be retrieved from the browser's local storage through JavaScript. Besides the wide variety of standard authentication and session implementation, one can encounter non-standard methods. For example, a website may require a custom header to be sent. To allow security testers to deal with unforeseen situations, a web crawler ought to support extensive user configuration and user-created plugins. A more coarse solution would be to place a proxy (such as `mitmproxy`) in-between the site and the crawler to allow for on-the-fly request modification.

Secondly, modern web applications make heavy use of JavaScript. Modern websites have a thousand ways to perform various actions. These websites are not designed to be interacted with in an automated manner. For this reason, modern crawlers have shifted to using browser engines, such as Selenium, for navigating websites [54]. Selenium-based crawlers are no panacea. For example, single-page applications (SPAs) heavily use asynchronous JavaScript requests. Properly crawling such applications has proven to be difficult [45]. One known limitation for the Burp Crawler, for example, is that the Recorded Login Sequences of the Burp Scanner can not handle `<iframe>`'s [48]. With continuously emerging technologies, crawlers need to be continuously improved.

Thirdly, common edge cases, such as volatile data or huge amounts of highly similar pages, can pose challenges for crawlers. Taking the Burp Crawler as an example, a maximum crawl depth can be set.[1] This can be based on the literal path depth, such as `/media/cinematography/drama/pulpfiction`. It can also be based on the number of sequential actions a crawler can take from its starting point, such as `/book/page?=1`, `/book/page?=2`, et cetera. Similarly, the breadth of a crawl can be configured. If an online readable book contains a thousand pages, it may not be necessary to visit every page.

Ideally, a crawler can cover the entire application with one or multiple crawls. However, the amount of content available can pose time limitations. Fine-tuning allows a security tester to cover as many actual functionalities as possible within a limited amount of time. Volatile or dynamic behavior is challenging to handle correctly. Web applications can give different responses to the same request and the same responses to different requests, as shown in chapter 2.[2] However, it is not always feasible to verify whether a crawl successfully dealt with all these edge cases.

Even with most challenges above somewhat dealt with, inherent issues remain. Careful crawling is inherently slow. It can hardly be parallelized. The order of the users with whom

[1] For an overview of the Burp Scanner configuration options, see [47].

[2] The Burp Crawler makes some attempts to deal with volatile data. See [46].

a web application is crawled can influence the results. However, if two crawls were to be started at once, they are likely to influence each other or lead to infinite loops. Therefore, crawls are best performed linearly. Another issue is the potentially dangerous nature of web crawls. On production systems, web crawlers may accidentally interact with sensitive or administrative functionality. This can have dire consequences, such as accidentally deleting all users or making personally identifiable information publicly accessible. One should, therefore, preferably test on testing (or 'acceptance') environments. Even then, deleting critical application data can make the acceptance environment unusable. Running a crawl therefore always requires prior knowledge prior knowledge of the application at hand.

Regardless, when a crawler is chosen, the generated exchanges must be extracted. Although crawlers often have some ways of exporting results, there is no standardized format for exchange flows. In order to remain crawler-independent, it is proposed to use Mitmproxy[3] for capturing network traffic. Mitmproxy is a well-known Python-based free and open-source proxy software that allows the dumping and manipulation of requests and responses. Specifically, `mitmdump` can be used as a forward proxy for the crawler, thereby automatically dumping the request-response flows. For example, the following command can be used: `$ mitmdump --proxy-port 8090 --write crawl_1.dump`. The crawler then needs to be configured to use this port as its proxy.

Proxy configuration is often straightforward. Many Linux-based applications work with the `HTTP_PROXY` and `HTTPS_PROXY` environment variables, including Python programs. For example, the following Bash commands allow a security tester to proxy their crawler to a listening `mitmdump` instance:

```
$ export HTTP_PROXY="http://localhost:8090"
$ export HTTPS_PROXY="http://localhost:8090"
$ python crawler.py
```

Regarding proxy configurations for web application crawlers, the Python crawling module 'Scrapy' provides dedicated Proxy options via middleware [53]. Within Burp Suite, upstream proxies can be configured per project [50].

3.2 Request selection

In order to test for access control issues, candidate requests need to be selected. A human tester would know from experience, context, and observation which web application functionalities may be of particular interest. Again taking the example application for the news organization, a tester can rationalize what actions the 'writer', 'reader', and 'editor' roles should be allowed to perform. She can verify whether access controls are enforced by crafting an appropriate request. The limitation of manual tests, however, is that they do not scale well. Ideally, a tester exhaustively covers the entire application flow. Just like a SQL injection vulnerability can occur in every input location (arguments, parameter names, paths, body, cookies, headers, et cetera.), access control vulnerabilities can occur in every

[3] Man-in-the-Middle Proxy [36].

unique code path. In order to test as much functionality as possible, a vulnerability scanner should maximize its total path coverage. However, without a preliminary selection phase, the number of actions that need to be investigated quickly becomes unwieldy.

The selection handler class `RequestSelector` (3.2) can be used to select promising requests from two Mitmproxy flow dumps. The class takes in a tuple of two `Crawl` (Listing 3.1) objects. Optionally, it can take in a set of regexes for domain allowlisting, domain denylisting, path blocklisting, volatile parameter blocklisting, and status code denylisting. Additionally, an option can be passed to ignore any cached selection results. When running an instantiated `RequestSelector` object with the method `run`, a check is performed for a cached selection object. If the cached selection does not exist or caching is disabled, the two mitmproxy dumps from the `Crawl` objects are read normally.

```python
@dataclass(kw_only=True, slots=True)
class Crawl:
    """A Crawl dataclass
    Should be initialized with a username, password, link to a login page, and
    a filepath to a Mitmproxy dump. A flow object can be attached at a later point.
    """

    mitm_dump: str
    username: str
    login_page: str = field(repr=False)
    password: str = field(default_factory=str, repr=False)
    flow: list[Exchange] = field(init=False, default_factory=list, repr=False)
```

Listing 3.1: Crawl dataclass.

```python
class SelectionHandler:
    """Selects requests to test for access control vulnerabilities"""

    def __init__(
        self,
        crawls: tuple[Crawl, Crawl],
        allowlist: Optional[set[str]] = None,  # Regexes
        denylist: Optional[set[str]] = None,  # Regexes
        path_denylist: Optional[set[str]] = None,  # Regexes
        volatile_params: Optional[set[str]] = None,  # Non regex
        status_code_denylist: Optional[set[int]] = None,  # Non-regex
        forbidden_extensions: Optional[set[str]] = None,
        ignore_cache: bool = False,
    ):
        if len(crawls) != 2:
            raise ValueError("Expected exactly two Crawl objects.")

        self.config = {
            "allowlist": allowlist if allowlist else set(),  # Set of Regexes
            "denylist": denylist if denylist else set(),  # Set of Regexes
            "path_denylist": path_denylist if path_denylist else set(),
            "status_code_denylist": status_code_denylist
            if status_code_denylist
            else set(),
            "forbidden_extensions": forbidden_extensions
```

```
            else set({
                ".js",
                "min.js",
                ".css",
                ".min.css",
                ".ttf",
                ".woff",
                ".woff2",
                ".jpeg",
                ".jpg",
                ".gif",
                ".png",
                ".svg",
            }), # Default forbidden extensions.
            "volatile_params": volatile_params
            if volatile_params
            else set(),  # Set of non-regexes.
        }
        self.crawls = crawls
        filenames = [crawl.mitm_dump for crawl in self.crawls]
        self.checksum = self.checksum_files(filenames)
        self.cache_file = CACHE_PATH + self.checksum + ".pickle"
        self.ignore_cache = ignore_cache
```

Listing 3.2: RequestSelector initialization method.

For each Exchange object (Listing 3.3) several checks are performed to verify that the requests are in scope. Not all domains that are encountered during a crawl are of importance. For example, a website may load JavaScript assets from third parties. Allowlisting and denylisting regexes can be supplied for this purpose. Not all filetypes are of interest either. For example, `.css` or `.woff` files are typically publicly accessible files. In addition, some paths are known to be benign beforehand, such as a public `/documentation/` directory. Similarly, some status codes are not of interest. Especially 404 codes are likely not of any relevance. Filtering these kinds of requests helps to reduce the number of requests to be retested. Lastly, a security tester may have identified volatile parameters already. For example, some applications send a request identifier for logging purposes with each request, such as `_=1234`. By providing these volatile parameters, requests with the paths `/book?action=read&_=1234` and `/book?action=read&_=1235` will be correctly identified as identical requests.

```
class Exchange(dict):
    """ Request-Response exchange. """

    def __init__(
        self,
        request: Request,
        response: Response,
        volatile_params: dict = {},
    ):
        self.request: Request = request
        self.response: Response = response
        self.ac_request = None
        self.ac_response = None
```

```
        self.category: Category = Category.UNCATEGORIZED
        self.volatile_params = volatile_params
...
```

Listing 3.3: Exchange initialization method.

After all in-scope exchanges are loaded for the two crawls, the selection process begins. The 'left crawl' is the user we want to test access control vulnerabilities for. For example, if we have a crawl for an administrative user and an unauthorized user, the 'left crawl' would be for the administrative user. We then find the relative complement of the 'right flow in the left flow'. The relative complement is synonymous with the set-theoretic difference of the 'left crawl' and the 'right flow'. In other words, we find all requests that exist in the 'left flow' (administrator) but not in the 'right flow' (unauthenticated). (Listing 3.4) The resulting exchanges are interesting endpoints to check for access control vulnerabilities because the user from the 'right flow' did not have access to these endpoints through the user interface. Therefore, it is hypothesized that only those requests that occurred in one crawl, but not in the other, are at risk of access control issues.

```
@staticmethod
def _relative_complement(
    flow_left: list[Exchange],
    flow_right: list[Exchange],
    volatile_params: set[str],
):
    """
    Takes two flows. Returns the relative complement of 'flow_right' in 'flow_left'.
    ↪ In other
    words, returns all requests that occur in the left flow (crawl),
    but not in the right flow (crawl).
    """

    selection = []

    for exchange in flow_left:
        exchange.category = SelectionHandler._categorize(
            exchange.request, flow_right, volatile_params
        )

        if exchange.category in [
            Category.DIFFERENT,
            Category.DIFF_KEYS,
            Category.DIFF_PARAMS,
        ]:  # If somewhat unique, add request
            selection.append(exchange)

    return selection
```

Listing 3.4: Method retrieving the relative complement of two flows.

In order to find functionalities that were reached in the 'left crawl' but not in the 'right crawl', each request in the 'left crawl' is compared to each request in the 'right crawl'. During the comparison, each request from the 'left crawl' is categorized according to four

degrees of ‘uniqueness’ (3.5). Comparing the combinations of the hosts, methods, and paths of a request is the most obvious and easiest way to differentiate between requests. Clearly, the following requests are unique:

- `GET https://ac-publishing/books/1`
- `DELETE https://ac-publishing/books/1`
- `GET https://ac-publishing/books/2`
- `GET https://competitor-publishing/books/1`

We would not expect identical functionality from any of these requests. This initial differentiation will be called ‘category 1 uniqueness’. Category 1 uniqueness is too coarse, however. According to this categorization, the following requests are identical, which they they likely are not:

- `GET https://ac-publishing/books?preview=1`
- `GET https://ac-publishing/books?loan=1`

On the other hand, the following requests *are* likely identical, despite having different queries:

- `GET https://ac-publishing/books?trackingId=9`
- `GET https://ac-publishing/books`

We cannot know beforehand whether a parameter name contributes to unique functionality for every request. Therefore, we should keep track separately of this second type, called ‘category 2 uniqueness’. Knowing what type of uniqueness a category is allows both the tester and the implementation to interpret the results more accurately.

Lastly, some requests only vary in their parameter values, but are clearly different functionalities:

- `GET https://ac-publishing/books?action=buy`
- `GET https://ac-publishing/books?action=return`

On the other hand, some requests vary in parameter values but do probably follow the same code path:

- `GET https://ac-publishing/books?cacheBuster=2`
- `GET https://ac-publishing/books?cacheBuster=3`

This category is referred to ‘category 3 uniqueness’. Requests which fall not in any of these categories have a host, method, path, and query combination that is already seen in another request. This category is marked as ‘category 0’ and is discarded.

```python
@staticmethod
def _categorize(
    target_request: Request, flow: list[Exchange], volatile_params: set[str]
) -> Category:
    """
    Category 0: Exactly the same.
    Category 1: A request that has a unique host, method, and path combination.
    Category 2: A request that has a non-unique host, method, and path combination,
        but unique keys
    Category 3: A request that has a non-unique host, method, and path combination,
    ↪ with
        non-unique keys, but with unique values.

    """

    for exchange in flow:
        if SelectionHandler._id_equal(target_request, exchange.request):
            if SelectionHandler._params_equal(
                target_request, exchange.request, volatile_params
            ):
                return Category.EQUAL  # Category 0, Seen before
            elif SelectionHandler._keys_equal(
                target_request, exchange.request, volatile_params
            ):
                return Category.DIFF_PARAMS  # Category 3, different parameter values
            else:
                return Category.DIFF_KEYS  # Category 2, Same endpoint, but different
    ↪ keys
    return Category.DIFFERENT  # Category 1, Completely different
```

Listing 3.5: Method for determining the uniqueness category of a request within a flow.

Of course, this method is not without issues. Some requests are mistakenly placed in this category. Requesting the page `GET https://ac-publishing/book/1/read?action=nextPage` may lead to a new action every time the request is sent. This edge case is currently not dealt with.

Lastly, the selected exchanges of the three interesting categories (`DIFFERENT`, `DIFF_PARAMS` ↪ , and `DIFF_KEYS` are added to the Selection object. Listing 3.6 is a Selection object is pickled and cached for future use. A number of details about the Selection object can be printed in CSV format. This is useful for verifying the results of the selection process. For example, see Listing 3.7.

```python
class Selection:
    """The selection of requests: A relative complement of right_crawl in left_crawl.
    The categories are the requests that exist in the left crawl, but not
    in the right crawl.
    An access control check will reperform the requests from categories using
    authentication from the right crawl.
    """

    def __init__(
        self, left_crawl: Crawl, right_crawl: Crawl, exchanges: list[Exchange]
```

```
    ):

        self.left_crawl = left_crawl
        self.right_crawl = right_crawl
        self.exchanges = exchanges
        self.id = self._generate_id(self.left_crawl.login_page)
...
```

Listing 3.6: Selection class initialization.

```
category,method,host,path,status_code,reason,query,urlencoded_form,multipart_form
1,GET,example1.nl,/dashboard.html,302,Moved Permanently,[],[],[]
1,GET,example1.nl,/profile.html,200,OK,"[('id','123')]",[],[]
3,GET,example1.nl,/profile.html,200,OK,"[('id','124')]",[],[]
1,POST,example1.nl,/changeProfile.html,302,Moved Permanently,[],[],"[('username', '
    ↪ admin'), ('profile_pic','<base64>') ]"
1,GET,example1.nl,/changeProfile.html,200,OK,[],[],[]
1,POST,example1.nl,/newLogin.html,302,Moved Permanently,[],"[('username', 'admin'),('
    ↪ password','hunter2')] ",[]
```

Listing 3.7: Selection CSV output.

3.3 Request replay

In order to test for access control issues, the request needs to be replayed in another user context. To this end, the `ReplayHandler` class can be used (Listing 3.8). Several parameters can be passed. Most importantly, the `Selection` object created by `SelectionHandler` needs to be passed. In addition, several authentication modes are provided (Listing 3.9). Depending on the application, authentication can either be performed once (`XDRIVER_REAUTH`), before every request (`XDRIVER_REAUTH`), or not at all (`NONE`). The XDriver option uses the Selenium-based XDriver module created for the research 'The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws' [19].

The XDriver module is used for authentication because it has proven successful in identifying arbitrary login forms and submitting credentials correctly. This allows the program to simply use the credentials specified in the Crawl (Listing 3.1) object without requiring complex authentication schemes.

Some situations require the authentication process to be performed each time. For example, some endpoints may deauthenticate users when unexpected input is received. One downside to this, however, is that it is far slower due to the startup time of the XDriver instances and the extra network requests made as part of the authentication process. If the XDriver module is not able to authenticate correctly, the security tester can configure alternative methods. They can set hardcoded cookies and headers via the `add_header` and `add_cookie` methods. Alternatively, they can set the `set_proxy` header to proxy requests through another program, such as Burp or Mitmproxy, where more advanced session handling rules can be configured. [4]. In any case, setting the authentication mode to `None` allows for other authentication methods to take over.

[4]For example, see the Burp documentation on its session handler [49]

Other parameters that can be supplied are the number of threads to run the replayer with. For cases with a considerable number of requests, multiple threads can be run concurrently to divide the workload. However, as with the crawl, this reduces the determinism of a test. Timing differences between threads can lead to different results. For debugging options, it is possible to set the `headless` option to false. This runs the XDriver authenticator in headed mode, allowing the security tester to inspect the authentication process visually.

```
class ReplayHandler:
    """ Handles the replaying of requests."""

    def __init__(
        self,
        selection: Selection,
        auth_mode: AuthMode = AuthMode.XDRIVER,
        threads: int = 1,
        headless: bool = True,
    ):
        if auth_mode not in AuthMode:
            raise ValueError("Invalid authentication mode.")

        self.auth_mode = auth_mode
        self.proxy = None
        self.n_threads: int = threads
        self.selection: Selection = selection
        self.preflight: Optional[Preflight] = None
        self.start_regex = None
        self.end_regex = None
        self.headers = dict()
        self.cookies = dict()
        self.params = list()
        self.authenticator = None
        if headless:
            XDriver.enable_headless()
```

Listing 3.8: ReplayHandler class initialization.

```
class AuthMode(Enum):
    """Authentication modes"""
    NONE = auto()
    XDRIVER = auto()
    XDRIVER_REAUTH = auto()
```

Listing 3.9: AuthMode options.

As relayed in chapter 2, and also in section section 3.1, one of the biggest challenges in automated interaction with web applications is session-relevant data, such as CSRF tokens and ASP.NET ViewStates. Especially troublesome are ViewStates, which contain information about previously made requests, among other things. Although these *can* be forged, with effort, secure implementations encrypt the ViewStates. If an invalid ViewState is sent, such as when performing a replay request, the server may decide to not further process the request.

In order to accommodate the various methods of session tracking, several authentication methods are provided as options. As stated before, automated authentication is provided via the XDriver, and hardcoded cookie and header values can be set. For handling CSRF tokens, a preflight request can be sent prior to every replayed request. The preflight request can be enabled using `set_preflight` (Listing 3.10), creating a `Preflight` object (Listing 3.11). The URL, method, query parameters, and body parameters can be supplied. If no URL is supplied, the preflight request is performed to the same endpoint as the replay request.

```
def set_preflight(
    self,
    url: Optional[str] = None,
    method: str = "GET",
    params: dict = {},
    data: dict = {},
):
    """
    If `url` is set to `None`, preflight goes to the same endpoint.
    Example: replayer.set_preflight('http://localhost/endpoint')
    """
    self.preflight = Preflight(method, params, data, url)
```

Listing 3.10: Enable preflight requests.

```
class Preflight:
    def __init__(
        self, method: str,
        params: dict,
        data: dict, url:
        Optional[str] = None
    ):
        self.method = method
        self.url = url
        self.params = params
        self.data = data
        self.regexp = None
        self.last_response = None
        self.last_selection = None
        self.mode = None
        self.param_name: Optional[str] = None

    def set_regex(self, regex: str):
        regexp = re.compile(regex)
        if regexp.groups != 1:
            raise ValueError("Exactly 1 regex capture group expected.")
        self.regexp = regexp

    def set_mode(self, mode: PreflightParamTypes):
        if PreflightParamTypes.has_member_key(mode):
            raise ValueError("Unknown mode option.")
```

Listing 3.11: Preflight class.

The preflight request can be configured to extract a value from the response using regular expressions (Listing 3.12). CSRF tokens are typically sent in a meta tag in the HTML response. (Listing 3.13). Using a regular expression with a capture group, such as `<meta name="csrf-token" content="(.*)">`, the CSRF value can be extracted. The value can then be added as either a cookie, header, or parameter value by setting the appropriate mode and providing a parameter name.

```
    def configure_preflight_value(
        self,
        regex: str,
        mode: PreflightParamTypes,
        param_name: str
    ):
        """Use Regex to select the value returned by the preflight response.
        Example: replayer.configure_preflight_value(
        '<meta name="csrf-token" content="(.*)">',
        "header",
        "X-Csrf-Token",
        )
        Limitation: Only selects first value.
        """
        if not self.preflight:
            raise TypeError("Preflight not configured.")
        self.preflight.set_regex(regex)
        self.preflight.set_mode(mode)
        self.preflight.param_name = param_name
```

Listing 3.12: Configure the preflight capture group.

```
<meta name="csrf-token" content="CSOib4uk-D7bANLQEygByTrYTnPa2gP2-R3U"/>
```

Listing 3.13: Example CSRF token.

Once the authentication, preflight, cookie, header, and parameter configurations are in place, the request handler can be run. After the necessary authentication steps are taken, each request in each exchange in the passed Selection object is replayed using `request_replay`. (Listing 3.14) The implementation at the time of writing handles URL-encoded and multipart content types. Other content types can be added, such as JSON, XML, and streams. The request is configured to use headers, cookies, query parameters, multipart parameters, URL-encoded parameters where appropriate. The new request-response pair is added to the Exchange object (Listing 3.3) as the `ac_request` and `ac_response`.

```
def request_replay(self, request: Request):
    """Replays a request."""
    query = deepcopy(request.query)
    urlencoded_form = deepcopy(request.urlencoded_form)
    headers = deepcopy(request.headers)
    cookies = deepcopy(request.cookies)
    multipart_form = deepcopy(request.multipart_form)

    self.process_params(query, multipart_form, urlencoded_form)
    self.process_headers(headers)
```

```
    self.process_cookies(cookies)
    self.remove_cookies_from_headers(headers)

    # Handle content type.
    if request.content_type and request.content_type.startswith(
        "multipart/form-data"
    ):
        body = None
        self.remove_multipart_boundary(request, headers)
    elif request.content_type and request.content_type.startswith(
        "application/x-www-form-urlencoded"
    ):
        body = urlencoded_form
        multipart_form = None
    else:
        body = request.content
        multipart_form = None

    if self.preflight:
        self.do_preflight(request.url)

    return requests.request(
        method=request.method,
        url=request.url,
        params=query,
        data=body,
        files=multipart_form,
        cookies=cookies,
        headers=headers,
        proxies=self.proxy,
        verify=False,
        allow_redirects=False,
    )
```

Listing 3.14: Request replay method.

After all replayed requests and responses are collected, the responses can be compared in order to detect access control vulnerabilities.

3.4 Response comparison

Each Exchange object in the selected requests now contains two request-response pairs, one for the original request and one for the AC-tested request. Using the `ComparisonHandler` class, the responses of each Exchange object can be compared (Listing 3.15). The class only takes the populated Selection object as input. Using the `Run` method, response comparison is started.

Three features are compared: the status codes, the content lengths, and the response body difference ratio. Different status codes are the most reliable way of differentiating different application behavior. In the case of functioning access controls, the original request often returns a `200 OK` status code, whereas the AC-tested returns a `401 Unauthorized` or `403 Forbidden` status code. Alternatively, servers respond with a `301 Found` redirecting to the

login page, or even return a `500 Internal Server Error` code. However, many applications return a `200 OK` when an unauthorized request is made, instead informing the user in the body of the response.

```
class ComparisonHandler:
    def __init__(self, selection: Selection):
        """Compares the responses of the original and AC-tested requests."""
        self.selection = selection
        self.directory = self._generate_directory(selection)
        self.findings = []
        self.column_width = 80

        self.results = DataFrame(
            columns=[
                "uuid",
                "method",
                "request",
                "diff_ratio",
                "status_code_same",
                "status_code_original",
                "status_code_ac",
            ],
            index=range(len(self.selection.exchanges)),
        )
        self.results["uuid"] = [str(uuid4()) for _ in range(len(self.results.index))]
...
```

Listing 3.15: ComparisonHandler class initialization.

Especially the inconsistent HTTP status code behavior makes detecting access control vulnerabilities more difficult. Response bodies must, therefore, also be compared. Response bodies are seldom identical due to volatile data, such as timestamps, CSRF tokens, and state and user-specific data. Nevertheless, responses with high similarity are helpful indicators of broken access controls. This difference ratio is tied to the difference in content length. Regardless, the difference in content length can sometimes provide some extra context. For example, if a request to */nonexistent* returns 200 for both responses, with identical response bodies, but with a very short content length, then this observation indicates may indicate missing or broken functionality instead of an access control issue. But again, this depends on context. Many asynchronous data-changing requests, such as a `POST` request for updating profile information, can return a response with an empty body.

An HTML table is created, with each row containing information about an exchange (Figure 3.1). The columns contain links to the response difference HTML pages (Figure 3.2), links to the request difference HTML pages, a summary of the request, such as the method and the endpoint, the original selection category, the difference ratio, whether the status codes are equal, and the status codes. The table is sorted firstly by whether the status codes are then same, and secondly descending by the difference ratio.

For each response pair, an HTML page is generated with a visual indication of differences in responses (Figure 3.2). This allows for easy and immediate inspection of response differences. For good measure, the same is done for the request pairs. This allows verification of the original requests, debugging, and manual request replay.

Figure 3.1: Comparison HTML table.

Response diff	Request diff	Method	Request ID	Category	Diff Ratio	Status Code Equal	Status Code Original	Status Code AC
Responses	Requests	POST		2.0	0.9967637540453075	True	200	200
Responses	Requests	GET		1.0	0.9696969696969697	True	200	200
Responses	Requests	POST		2.0	0.9375	True	302	302
Responses	Requests	POST		3.0	0.9375	True	302	302
Responses	Requests	GET		1.0	0.8571428571428571	True	302	302
Responses	Requests	POST		2.0	0.5	False	302	401
Responses	Requests	POST		3.0	0.5	False	302	401
Responses	Requests	POST		2.0	0.1319796954314721	False	302	200
Responses	Requests	GET		3.0	0.0748663101604278	False	200	401
Responses	Requests	GET		3.0	0.0748663101604278	False	200	401
Responses	Requests	GET		3.0	0.0748663101604278	False	200	401
Responses	Requests	GET		3.0	0.0748663101604278	False	200	401
Responses	Requests	GET		3.0	0.05223880597014925	False	200	401

Figure 3.2: Highlighted comparison between responses. Cut between line 22 and 1499.

Request: POST

Response Comparison

Colors	Links
Added	(f)irst change
Changed	(n)ext change
Deleted	(t)op

Original.

```
1  X-Frame-Options: sameorigin
2  Strict-Transport-Security: max-age=31536000; includeSubDomains
3  X-XSS-Protection: 1
4  X-Content-Type-Options: nosniff
5  Referrer-Policy: same-origin
6  Pragma: No-cache
7  Expires: Thu, 01 Jan 1970 00:00:00 GMT
8  Cache-Control: no-cache, no-store
9  Content-Type: text/html;charset=ISO-8859-1
10 Content-Language: en-US
11 Transfer-Encoding: chunked
12 Date: Sat, 11 Jun 2022 09:46:45 GMT
13 Connection: close
14 Server: unknown
15
16
17
18
19
20
21 <!DOCTYPE html>
22 <html xmlns="http://www.w3.org/1999/xhtml">
1499
   >
   >
1500
   >
1501
   ><input type="radio" class=  id=  name=  value=  >
   >  <BR/>
1502
   >
```

AC-tested.

```
1  X-Frame-Options: sameorigin
2  Strict-Transport-Security: max-age=31536000; includeSubDomains
3  X-XSS-Protection: 1
4  X-Content-Type-Options: nosniff
5  Referrer-Policy: same-origin
6  Pragma: No-cache
7  Expires: Thu, 01 Jan 1970 00:00:00 GMT
8  Cache-Control: no-cache, no-store
9  Content-Type: text/html;charset=ISO-8859-1
10 Content-Language: en-US
11 Date: Sun, 12 Jun 2022 18:07:08 GMT
12 Connection: close
13 Server: unknown
14 Content-Length: 52811
15
16
17
18
19
20
21 <!DOCTYPE html>
22 <html xmlns="http://www.w3.org/1999/xhtml">
1499
   >
   >
1500
   >
1501
   ><input type="radio" class=  id=  name=  value=  >
   >  <BR/>
1502
   >
1503
   ><input type="radio" class=  id=  name=  value=  >
   >  <BR/>
1504
   >
1505
   ><input type="radio" class=  id=  name=  value=  >
   >  <BR/>
1506
   >
```

Chapter 4

Evaluation

4.1 Setup

Over the course of several months, the BACS implementation was evaluated on testing environments of web applications during web application security assessments by DongIT. The testing environments are near-identical to production environments, but contain testing data instead of production data. For each assessment, the vulnerability scan was performed after the original security tester(s) performed the security assessment.

Due to the sensitive nature of client application information and especially vulnerabilities, the actual data cannot be publicized. However, anonymized examples are provided. The described vulnerabilities are real, but the paths and parameters have been changed. Clients or client applications are not referred to by name.

Due to the custom configurations presently required for each of the four BACS phases, it was not feasible to run the application on a large number of web applications. Instead, the web application vulnerability scanner was run on ten different web applications. As a result, the analysis of this evaluation setup is qualitative instead of quantitative.

Out of the ten applications examined, it was only possible to fully complete the entire BACS pipeline for four applications. Access control vulnerabilities of varying severity were found for each of those four applications. In some cases, this uncovered vulnerabilities that were not found during testing. In other cases, the BACS implementation missed vulnerabilities that the security tester found, although always due to the limitations of crawling and correct request replaying.

4.2 Failures in crawling and request replay

Completing the BACS pipeline failed for six out of ten of the tested web applications. In each case, this was related to either the ability to crawl the application, or to successfully replay requests. In one case, the crawl only returned a minimal amount of endpoints, missing most functionality due to the substantial use of dynamic JavaScript. In another case, web sockets were extensively used, which could not be correctly proxied through Mitmproxy. In two cases, a large number of destructive functionalities were present. Any authenticated

user was able to delete objects from the web application. Running the web crawler on these applications could have led to an unusable web application. For the last two failed web applications, the crawl did succeed, but correctly replaying requests was not feasible. These two examples are further expanded upon below.

In one case, the application added a Hash-based Message Authentication Code (HMAC) to a custom header in each request. The key of the HMAC was provided after authentication and stored in the local browser storage. The HMAC was performed over a timestamp and a nonce returned by the server. Since all these values are user-controlled, this request signature can be recreated by the client. However, the broken access control scanner was not equipped to deal with this level of granularity and custom session-relevant data.

Another application made use of Cross-Site Request Forgery tokens with dynamic parameter names. If a POST request was sent to an endpoint, the CSRF token parameter name was derived from the name of that endpoint. For example, in Listing 4.1, the CSRF token parameter name `csrf_edit_user` matches the path `edit_user`.

These challenges could not be overcome within the remaining research time set for the penetration test.

```
1 POST /edit_user HTTP/2
2 Host: website.nl
3 Content-Type: application/x-www-form-urlencoded
4
5 id=1&name=Alice&csrf_edit_user=WfF1szMUHhiokx9AHFply5L2xAOfjRkE
```

Listing 4.1: POST request with dynamic CSRF token parameter name.

All in all, the replay failure cases can typically be overcome by implementing extra features. However, this quickly increases the complexity of the codebase for what is probably rare behavior. Tackling these issues is possible, but this is rather part of longterm goals instead of an initial research. Regarding the failures due to crawling, these are simply the limitations of web crawlers. Better coverage can typically by achieved, but requires increasingly more application-understanding and custom configuration.

4.3 Successfully discovered broken access controls

The access control vulnerability scanner found access control issues in four web applications. In two cases, this only concerned low-impact endpoints. These were generic endpoints that did not reveal any user-specific data. For example, in one case, an unauthenticated request to the endpoint `/user/profile/2` returned the template of the user profile page, but without any actual user data. Nevertheless, this provides unauthenticated attackers insight into the inner workings of the web application.

More importantly, what these results did reveal was that the applications did not systemically handle unauthorized access cases. Typically web applications ought to restrict access by default, only granting access after privileges are checked explicitly. Observing multiple low-impact endpoints without access control can indicate ad-hoc access control rules. Even if more severe access control vulnerabilities are not present at the moment, they are likely

to occur throughout development. In any case, this application behavior warrants additional manual investigation by either the security tester or the developer.

Two applications were found that had high-impact access control vulnerabilities. These applications, referred to as Alpha and Beta, are investigated more in-depth below.

4.3.1 Vulnerable application Alpha

Several access control vulnerabilities were found in web application 'Alpha'. Three kinds of access control vulnerabilities were found. Firstly, low-privileged authenticated users could edit the permissions of other users. This could be achieved with a `POST` request to `/modules/configuration/users/permissions_addedit.php?user_id=<ID>`, where a user role could be supplied. The vulnerability allows authenticated users to escalate their privileges to administrative users.

Secondly, uploaded files to `/documents/` were not access-controlled. A low-privileged authenticated user had no access to the `/documents/` page. However, she was able to download files using `/documents/download.php?file_id=<ID>`. This includes documents uploaded by administrative users and documents containing personally identifiable information and business secrets. Similarly, unauthorized users were able to upload files to the documents page. Due to the missing antivirus, it was possible to upload malware to the server, which can be downloaded and executed by other users.

Thirdly, several less-sensitive endpoints were not appropriately access-controlled. For example, a low-privileged user was able to create helpdesk tickets in the web application's ticketing system. This functionality should have been restricted to support users. Although this by itself cannot be used directly for data extraction or privilege escalation, data-changing functionality can often be abused with sufficient ingenuity. In this case, an attacker could create a support ticket for an administrative user, requesting an email change or password reset. A request could also be issued for access to a sensitive service, such as FTP, or the company's intranet.

Multiple other instances of broken access control were discovered. These vulnerabilities mostly concerned functionalities of lower impact, such as the retrieval of data graphs, and created checklist entries in the checklist functionality.

4.3.2 Vulnerable application Beta

For web application Beta, access control was structurally missing on the `/ajax/` endpoint. As a low-privileged authenticated user, it was possible to execute various AJAX (*Asynchronous JavaScript And XML*) requests. For example, it was possible to retrieve user data via a GET request to `/ajax/getUserInfo/<ID>`. This endpoint was intended as an administrative functionality. By itself, being able to enumerate user information such as usernames, email addresses, and phone numbers is already immensely useful for an attacker. However, in this case, the AJAX endpoint also returned password hashes, session identifiers, and the multi-factor authentication TOTP seed (Time-based One-Time Password). In other words, all authenticated users could take over the sessions of all other users. In addition, attackers

could gather and crack the password hashes of user, which can then be used for credentials stuffing attacks.

Many administrative functionalities were accessible via the AJAX endpoints, such as extracting invoices containing personally identifiable data, business details, and financial details from other users. This one instance of broken access control had dire consequences.

Chapter 5

Discussion

In this chapter, the results of the evaluation are discussed. Firstly, the results are interpreted. It is argued that the Broken Access Control Scanning methodology is a viable method for detecting broken access control, but only when the entire pipeline can be completed for a particular web application. Secondly, the implications of these results are stated. The BACS methodology provides several benefits over manual testing: 1) Automated scanning can cover more functionality, 2) Automated scanning can better deal with the complexity of multiple users, and 3) The exhaustiveness of scanning enables security testers to identify structural issues better. At the same time, manual testing remains necessary for both verification and divergent application behavior. Thirdly, the limitations of this research are discussed. The limitations are divided into feature, gradual, and inherent limitations. Fourthly and lastly, three avenues for future research are suggested: 1) Research into additional ways of establishing context awareness, 2) Broad internet-wide research, and 3) Deep qualitative research into the usage of BACS by security testers.

5.1 Interpretation

The hypothesis posited in this research is that deriving interpretable context is required to differentiate authorized from unauthorized functionality when testing for access control vulnerabilities in a gray-box setting. The suggested method for deriving contextual awareness is to assume that all functionality a user has access to via a web application's user interface (UI) is all functionality that a user should be authorized for. From the results, it appears to be possible to use the contextual awareness derived programmatically from the functionality accessible via the UI.

There are still challenges to be overcome, but these results provide ample proof that the BACS methodology is a viable method for at least certain applications. The independence of a specific crawling technology, and the modularity of the three other components (request selection, request replay, and response comparison), allow for this methodology to be easily adoptable, extensible and improvable.

5.2 Implications

Three primary benefits of the access control vulnerability scanner were identified. Firstly, although each of the vulnerabilities above can be found through manual inspection, using automated crawls for different users allows security tester to cover far more ground than they can test manually.

Secondly, comparing all permutations becomes increasingly difficult when an application has many roles. For example, one of the tested applications allows multiple companies to authenticate to the same application. Company 1 would authenticate to `https://www.website.com/company1/login`, and company 2 to `https://www.website.com/company2/login`. Each company has one regular user, and one administrative user role. We would want to test at least the following access control boundaries:

- Unauthenticated access control, within one company
  - `company1_regular_user1` vs. `unauthenticated_user`
  - `company1_admin_user1` vs. `unauthenticated_user`
- Horizontal access control, within one company
  - `company1_regular_user1`, `company1_regular_user2`
  - `company1_admin_user1`, `company1_admin_user2`
- Vertical access controls, within one company
  - `company1_regular_user1`, `company1_admin_user1`
- Access controls, between companies
  - `company1_regular_user1`, `company2_regular_user1`
  - `company1_admin_user1`, `company2_admin_user1`

These combinations are the bare minimum to check. An exhaustive assessment would require all combinations to be tested. For example, this provides no guarantee that the access control boundary between `company1_regular_user1` and `company2_admin_user1` is secure. It is unlikely that if an administrative user of company 1 cannot access functionalities from company 2, a regular user from company 1 *can* access administrative functionalities from company 2. However, this is not an impossibility and can easily be an oversight during testing. In addition, this only assumes a low number of roles. Being able to test many combinations easily can, therefore, significantly reduce the research time.

Thirdly, the broadness of the vulnerability scan allows security testers to identify structural issues. This provides justification for security testers to spend time investigating the access control configurations more thoroughly. If a number of low-impact access control inconsistencies are found, chances are that high-impact issues can also be found. In addition, it provides client developers with an overview of vulnerable endpoints, which they

themselves can use as a basis to identify inconsistent access control implementations. Being able to argue that a structural problem exists helps with rooting out the underlying issue instead of applying ad-hoc patches.

5.3 Limitations

There are still many limitations that prevent the BACS implementation from being a 'click-and-point' broken access control solution. These limitations can be separated into roughly three categories: 1. Feature limitations, 2. Gradual limitations, and 3. Inherent limitations.

5.3.1 Feature limitations

Web applications can use many different technologies and protocols, but not all technologies or protocols are supported by the BACS implementation. For example, many web applications use websockets for full-duplex communication between the client and server. However, neither the Burp crawler nor any other parts of the BACS pipeline is equipped to handle websocket requests. Re-establishing a websocket connection and replaying a websocket request likely would require an entire new pipeline. Nevertheless, access control vulnerabilities can occur just as well in websocket connections as in HTTP connections.

5.3.2 Gradual limitations

Much improvement can still be made by improving the functionality in any of the four pipeline phases. However, based on the evaluation results, most improvements can be found in the crawling phase and the replay phase. In either of the stages, a primary challenge is creating a program that can interface with any web application with minimal prior knowledge. As with other research on black-box vulnerability discovery [17], improving the crawler is directly tied to the total application surface that can be covered. This was also observed during the evaluation phase, where known endpoints were not reached via the crawler.

Improving the request replayer mainly entails the better handling of non-standard or less-common authentication and session mechanisms. This could be seen in the custom HMAC function and the dynamic CSRF parameter name. Instead of developing ad-hoc features for each edge-case as its encountered, it may be best to allow security testers to configure a generic callback request per request.

5.3.3 Inherent limitations

Lastly, some issues appear to be inherent to the automation web application interfacing. Actions that change the application state can be complicated to test. Especially with regard to destructive functionality, we see from the evaluation results that this can obstruct automated crawling. Although it is possible to configure denylists and allowlists for crawlers, configuration becomes increasingly meticulous if an application is sufficiently complex.

Web applications are too various to create a fully autonomous catch-all system. Prior knowledge of the authentication and session flows, potentially dangerous features, and ap-

plication quirks are required to appropriately configure the crawler and the request replayer to full effectiveness. For now, human oversight is imperative for ensuring good coverage and accurate results.

5.4 Future research

For future research, at least three promising topics can be investigated. Firstly, further research can explore different ways in which contextual awareness can be determined. For this research, the choice was made to use the UI to infer authorized behavior. However, this may not be the only method of establishing context. The methodology could be further augmented for cases where white-box access is provided, such as through application logs, source code analysis, or debug code injection.

Secondly, this research setup could be expanded to internet-wide research. The current implementation is designed to allow security testers to configure the scanner to achieve maximal code coverage on a particular application. However, the strategy could be adapted to allow access control scanning on a large number of applications with minimal required configurations, akin to the research setup of Drakonis et al. [19]. Drakonis et al. created a crawler that can automatically register and authenticate to applications. Combining these researches can lead to large-scale access control vulnerability research.

Thirdly, instead of expanding this thesis setup for broad research, future researchers can opt for deeper research into how testers work with automated access control tooling. In this research, the scanner was only used by the author. Further research into how security testers prefer to use such tools and what blindspots they may have in practice could drastically increase the efficacy of the broken access control scanner.

Chapter 6

Related work

6.1 White-box access control vulnerability detection

Little has been written on automated discovery access control vulnerabilities in web applications in a gray box setting. On the contrary, much has been written on various white-box access control vulnerability prevention and testing techniques. Monshizadeh et al. have used automated static code analysis to establish a baseline of "`authorization context consistency`". Deviations from this baseline could be used to detect access control vulnerabilities [37]. The *Nemesis* methodology was introduced in a research by Dalton and Zeldovich, which combines a predefined access control policy with dynamic traffic inspection to detect access control anomalies [13]. Similarly, Muthukumaran et al. present *FlowWatcher*, which also uses a predefined access control ('user-data-access (UDA)') policy with dynamic traffic and application state analysis [39]. A research of Wang et al. focused on authentication and access control vulnerabilities in Software Development Kits (SDKs), discovering authentication and authorization issues by comparing semantic models with dynamic state assertions and HTTP traffic observations [61]. Felmetsger et al. published a related research on logic flaws, again combining dynamic analysis of application traffic with code verification-based methods [22]. One popular and recent development is research into mining various data sources, such as access logs, to create access control rules. The research of Cotrini et al. is mostly focused on non-web application access control policies [9, 8]. However, [52] shows how access control policy mining can be used for establishing attribute-based access control (ABAC) schemes for web applications, in this case Amazon Web Services.

6.2 Black and gray-box vulnerability scanning

Several studies on automated black and gray-box vulnerability discovery have been published throughout the years. Some of this research focuses on vulnerability scanning (and its issues) as a whole. Specifically, about a decade ago, Doupé et al. researched black box vulnerability scanners, comparing several black-box vulnerability scanners and black-box vulnerability scanning methodologies [18, 17]. This research was especially insightful with

regard to the limitations of black-box scanning, such as the reliance on code coverage. Bau et al. published a similar research, where several web application vulnerability scanners were compared [5]. Other research concerned the development of general web application vulnerability scanners, such as the Web Application Vulnerability and Error Scanner (WAVES) framework developed by Huang et al. [26].

Other research focuses on detecting specific vulnerability types. Many vulnerability types have dedicated research published, including cross-site scripting [20, 58], SQL injection [25], cross-site request forgery (CSRF) [4], parameter tampering [6], and prototype pollution [3]. One especially relevant vulnerability type is logic flaws. Logic flaws are vulnerabilities where the intended flow of an application can be circumvented. Pellegrino et al. create parameter mutations, similar to parameter tampering, to test the program flows of E-commerce web applications [42]. A research by Li and Xue proposes the black-box BLOCK methodology for detecting state violations [30]. Wang et al. performed manual research into Cashier as a service (CaaS) applications, showing the difficulties of formal verification for complex logic models [60].

Logic vulnerabilities share similarities with state-dependent access control issues. State-dependent access control issues are currently not considered in this research, but it is a natural extension for future research. Logic flaws are, therefore, an interesting area of research to explore further.

Lastly, the research of Zuo et al. introduced AuthScope, a black-box tool that uses differential traffic analysis of mobile application API requests to find authentication and authorization issues. However, as opposed to this thesis, the AuthScope research focuses primarily on the authentication flow, and not on an exhaustive coverage of the target application functionality [63].

6.3 Crawling

For crawling-based vulnerability discoverers, better code coverage via web application crawling is directly linked to a better testing coverage of the application code [18, 17, 5]. Research on web application crawling is therefore of great importance for vulnerability discovery. One primary challenge is the treatment of dynamic functionality via JavaScript. The research of Mesbah et al. finds and triggers all elements in the DOM tree that are capable of changing state, thereby creating new application paths [34]. The research of Bos et al. hooks into the JavaScript APIs to detect various kinds of dynamic JavaScript behavior, using this information for establishing new paths [43]. Furthermore, combining automated user registration and authentication with automated crawling is of vital importance for scaling access control vulnerability research. Automated user registration and logins via Single-Sign On providers is used in AuthScope for testing mobile applications [63]. Drakonis et al. apply a similar methodology for web application research, but also introduce the automated detection and processing of registration and login forms for web applications [19]. In addition, Drakonis et al. implement an application-agnostic web crawler. However, code review of [19] revealed that only links are followed, not application forms.

Web application crawling appears to be a topic for which research has been performed

for specific challenges, such as [43], but recent research into integrating these specific challenges into one web application-agnostic crawler is lacking. Additionally, software developed during research is often, understandably, not maintained for years after publication, such as the research of Mesbah et al. [34]. This is an issue for the rapid development of web application technologies. An opportunity exists here for scientific research on open-source full-feature crawling methodologies and implementations.

Chapter 7

Conclusion

This thesis proposed a methodology and implementation for Broken Access Control Scanning, or BACS. A gap in the academic literature on gray and black-box automated access control testing was identified. The recent denomination of access control vulnerabilities as the first among the OWAPS 2021 Top 10 drives the urgency for research on this topic. The combinatorial explosion that comes with access control testing makes the predominant method of manual testing impractical and error-prone, leading to critical vulnerabilities being overlooked.

Establishing contextual awareness was identified as the primary challenge in automated access control vulnerability scanning. The proposed hypothesis is that all functionality a user can reach via the user interface is the only functionality a user is authorized for. Based on this hypothesis, the BACS methodology was proposed. The BACS methodology consists of four phases: Crawling, request selection, request replay, and response comparison. An accompanying Python implementation was developed and evaluated on ten web applications during web application penetration tests for DongIT. Only four out of ten web applications could be fully processed by all four phases of the BACS pipeline. However, access control vulnerabilities were found in each of those four cases. In two cases, high-impact vulnerabilities were discovered.

Overall, these results lead to the conclusion that the BACS methodology is a viable strategy for automating broken access control scanning. By extension, this lends support to the hypothesis that contextual awareness of the authorization scheme can be inferred via the user interface of a web application.

Several limitations and challenges remain. The diversity in web applications is especially an obstacle for crawling and request replaying. Some of these limitations are inherent to the challenge of automated web application interaction. Nevertheless, some of the observed limitations can be resolved by further improving upon the core functionality of the broken access control scanner.

All in all, the methodology and implementation presented in this thesis were found to be directly applicable to the security testing of access control. These results may form the basis for novel research into gray and black-box access control vulnerability discovery.

Bibliography

[1] Michael P. Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E. Culler, and Raluca Ada Popa. WAVE: A Decentralized Authorization Framework with Transitive Delegation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1375–1392, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/andersen.

[2] Apache. Apache HTTP Server Tutorial: .htaccess files - Apache HTTP Server Version 2.4, no date. URL https://httpd.apache.org/docs/2.4/howto/htaccess.html. Last accessed: June 24, 2022.

[3] Marco Balduzzi, Carmen Torrano-Gimenez, Davide Balzarotti, and Engin Kirda. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *NDSS*, May 2011.

[4] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*, page 75, Alexandria, Virginia, USA, 2008. ACM Press. ISBN 978-1-59593-810-7. doi: 10.1145/1455770.1455782. URL http://portal.acm.org/citation.cfm?doid=1455770.1455782.

[5] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345, Oakland, CA, USA, 2010. IEEE. ISBN 978-1-4244-6894-2. doi: 10.1109/SP.2010.27. URL http://ieeexplore.ieee.org/document/5504795/.

[6] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V. N. Venkatakrishnan. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. CCS '10, pages 607–618, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866375. URL https://doi-org.tudelft.idm.oclc.org/10.1145/1866307.1866375.

[7] Composer. The composer.json schema - Composer, no date. URL https://getcomposer.org/doc/04-schema.md. Last accessed: June 24, 2022.

[8] Carlos Cotrini, Thilo Weghorn, and David Basin. Mining ABAC Rules from Sparse Logs. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 31–46, April 2018. doi: 10.1109/EuroSP.2018.00011.

[9] Carlos Cotrini, Luca Corinzia, Thilo Weghorn, and David Basin. The Next 700 Policy Miners: A Universal Method for Building Policy Miners. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 95–112, New York, NY, USA, November 2019. Association for Computing Machinery. ISBN 978-1-4503-6747-9. doi: 10.1145/3319535.3354196. URL http://doi.org/10.1145/3319535.3354196.

[10] Sam Curry. Hacking chess.com and accessing 50 million customer records: Sam Curry, December 2020. URL https://samcurry.net/hacking-chesscom/.

[11] CVE. CVE-2021-24405, no date. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-24405. Last accessed: June 24, 2022.

[12] CVE. CVE-2022-0824, no date. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0824. Last accessed: June 24, 2022.

[13] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. pages 267–282, January 2009.

[14] Joel Dawson and J. Mcdonald. Improving Penetration Testing Methodologies for Security-Based Risk Assessment. April 2016. doi: 10.1109/CYBERSEC.2016.016.

[15] G. Deepa and P. Santhi Thilagam. Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Information and Software Technology*, 74:160–180, June 2016. ISSN 09505849. doi: 10.1016/j.infsof.2016.02.005. URL https://linkinghub.elsevier.com/retrieve/pii/S0950584916300234.

[16] Django. Using the Django authentication system | Django documentation | Django, no date. URL https://docs.djangoproject.com/en/3.2/topics/auth/default/. Last accessed: June 24, 2022.

[17] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In Christian Kreibich and Marko Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Lecture Notes in Computer Science, pages 111–131, Berlin, Heidelberg, 2010. Springer. ISBN 978-3-642-14215-4. doi: 10.1007/978-3-642-14215-4_7.

[18] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. *USENIX Security 12*, August 2012.

[19] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1953–1970, New York, NY, USA, October 2020. Association for Computing Machinery. ISBN 978-1-4503-7089-9. doi: 10.1145/3372297.3417869. URL http://doi.org/10.1145/3372297.3417869.

[20] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 37–48, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 978-1-4503-2278-2. doi: 10.1145/2557547.2557550. URL https://doi-org.tudelft.idm.oclc.org/10.1145/2557547.2557550.

[21] Shamal Faily, John Lyle, Ivan Flechais, and Andrew Simpson. Usability and Security by Design: A Case Study in Research and Development. In *Proceedings 2015 Workshop on Usable Security*, San Diego, CA, 2015. Internet Society. ISBN 978-1-891562-40-2. doi: 10.14722/usec.2015.23012. URL https://www.ndss-symposium.org/ndss2015/ndss-2015-usec-programme/usability-and-security-design-case-study-research-and-development.

[22] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. pages 143–160, September 2010.

[23] HackerOne. #446585 Exfiltrate and mutate repository and project data through injected templated service, November 2018. URL https://hackerone.com/reports/446585. Last accessed: June 24, 2022.

[24] HackerOne. #796379 [Critical] Insufficient Access Control On Registration Page of Webapps Website Allows Privilege Escalation to Administrator, February 2020. URL https://hackerone.com/reports/796379. Last accessed: June 24, 2022.

[25] William Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL Injection Attacks and Countermeasures. January 2006.

[26] Yao-Wen Huang, Chung-Hung Tsai, Tsung-Po Lin, Shih-Kun Huang, D.T. Lee, and Sy-Yen Kuo. A testing framework for Web application security assessment. *Computer Networks*, 48(5):739–761, August 2005. ISSN 13891286. doi: 10.1016/j.comnet.2005.01.003. URL https://linkinghub.elsevier.com/retrieve/pii/S1389128605000101.

[27] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, August 2019. ISBN 1-59327-828-4. Last accessed: June 24, 2022.

[28] Laravel. Authorization - Laravel - The PHP Framework For Web Artisans, no date. URL https://laravel.com/docs/9.x/authorization. Last accessed: June 24, 2022.

[29] Laravel. HTTP Session - Laravel - The PHP Framework For Web Artisans, no date. URL https://laravel.com/docs/9.x/session. Last accessed: June 24, 2022.

[30] Xiaowei Li and Yuan Xue. BLOCK: A Black-Box Approach for Detection of State Violation Attacks towards Web Applications. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 247–256, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076767. URL https://doi-org.tudelft.idm.oclc.org/10.1145/2076732.2076767.

[31] Steve Lipner. Security development lifecycle: Security considerations for client and cloud Applications. *Datenschutz und Datensicherheit - DuD*, 34(3):135–137, March 2010. ISSN 1614-0702, 1862-2607. doi: 10.1007/s11623-010-0021-7. URL http://link.springer.com/10.1007/s11623-010-0021-7.

[32] Thomas Maillart, Mingyi Zhao, Jens Grossklags, and John Chuang. Given enough eyeballs, all bugs are shallow? Revisiting Eric Raymond with bug bounty programs. *Journal of Cybersecurity*, 3(2):81–90, October 2017. ISSN 2057-2085. doi: 10.1093/cybsec/tyx008. URL https://doi.org/10.1093/cybsec/tyx008.

[33] Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo Arango Argoty. Secure coding practices in Java: challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 372–383, New York, NY, USA, May 2018. Association for Computing Machinery. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180201. URL http://doi.org/10.1145/3180155.3180201.

[34] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web*, 6(1):1–30, March 2012. ISSN 1559-1131, 1559-114X. doi: 10.1145/2109205.2109208. URL https://dl.acm.org/doi/10.1145/2109205.2109208.

[35] Microsoft. ASP.NET View State Overview — Microsoft Docs, October 2014. URL https://docs.microsoft.com/en-us/previous-versions/aspnet/bb386448(v=vs.100). Last accessed: June 24, 2022.

[36] Mitmproxy. Mitmproxy documentation, no date. URL https://docs.mitmproxy.org/. Last accessed: June 24, 2022.

[37] Maliheh Monshizadeh, Prasad Naldurg, and V.N. Venkatakrishnan. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications. pages 690–701, November 2014. doi: 10.1145/2660267.2660337.

[38] Mozilla. Authorization - HTTP — MDN, no date. URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization. Last accessed: June 24, 2022.

[39] Divya Muthukumaran, Dan O'Keeffe, Christian Priebe, David Eyers, Brian Shand, and Peter Pietzuch. FlowWatcher: Defending against Data Disclosure Vulnerabilities in Web Applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 603–615, New York, NY, USA, October 2015. Association for Computing Machinery. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813639. URL http://doi.org/10.1145/2810103.2813639.

[40] OWASP. Authorization Cheat Sheet - OWASP Cheat Sheet Series, no date. URL https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html. Last accessed: June 24, 2022.

[41] OWASP. OWASP Top 10:2021, no date. URL https://owasp.org/Top10/. Last accessed: June 24, 2022.

[42] Giancarlo Pellegrino and Davide Balzarotti. Toward Black-Box Detection of Logic Flaws in Web Applications. In *Proceedings 2014 Network and Distributed System Security Symposium*, San Diego, CA, 2014. Internet Society. ISBN 978-1-891562-35-8. doi: 10.14722/ndss.2014.23021. URL https://www.ndss-symposium.org/ndss2014/programme/toward-black-box-detection-logic-flaws-web-applications/.

[43] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In Herbert Bos, Fabian Monrose, and Gregory Blanc, editors, *Research in Attacks, Intrusions, and Defenses*, volume 9404, pages 295–316. Springer International Publishing, Cham, 2015. ISBN 978-3-319-26361-8 978-3-319-26362-5. doi: 10.1007/978-3-319-26362-5_14. URL http://link.springer.com/10.1007/978-3-319-26362-5_14.

[44] PHP docs. PHP: phpinfo - Manual, no date. URL https://www.php.net/manual/en/function.phpinfo.php. Last accessed: June 24, 2022.

[45] PortSwigger. It's now easier than ever to scan at scale with Burp Suite Enterprise Edition — Blog - PortSwigger, August 2021. URL https://portswigger.net/blog/its-now-easier-than-ever-to-scan-at-scale-with-burp-suite-enterprise-edition. Last accessed: June 24, 2022.

[46] PortSwigger. Crawling - PortSwigger, May 2022. URL https://portswigger.net/burp/documentation/scanner/crawling. Last accessed: June 24, 2022.

[47] PortSwigger. Crawl options - PortSwigger, May 2022. URL https://portswigger.net/burp/documentation/desktop/scanning/crawl-options. Last accessed: June 24, 2022.

[48] PortSwigger. Recorded login sequences - PortSwigger, July 2022. URL https://portswigger.net/burp/documentation/desktop/scanning/recorded-logins#limitations-for-recorded-login-sequences. Last accessed: June 24, 2022.

[49] PortSwigger. Configuring Burp's Session Handling rules - PortSwigger, no date. URL https://portswigger.net/support/configuring-burp-suites-session-handling-rules. Last accessed: June 24, 2022.

[50] PortSwigger. Burp Suite Options: Upstream Proxy Servers - PortSwigger, no date. URL https://portswigger.net/support/burp-suite-upstream-proxy-servers. Last accessed: June 24, 2022.

[51] RFC. RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace, July 2005. URL https://www.rfc-editor.org/rfc/rfc4122. Last accessed: June 24, 2022.

[52] Matthew W Sanders and Chuan Yue. Mining least privilege attribute based access control policies. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, pages 404–416, New York, NY, USA, December 2019. Association for Computing Machinery. ISBN 978-1-4503-7628-0. doi: 10.1145/3359789.3359805. URL http://doi.org/10.1145/3359789.3359805.

[53] Scrapy. Downloader Middleware — Scrapy 2.6.1 documentation, no date. URL https://docs.scrapy.org/en/latest/topics/downloader-middleware.html. Last accessed: June 24, 2022.

[54] Selenium. Selenium, no date. URL https://www.selenium.dev/. Last accessed: June 24, 2022.

[55] Hossein Shafagh, Lukas Burkhalter, Sylvia Ratnasamy, and Anwar Hithnawi. Droplet: Decentralized authorization and access control for encrypted data streams. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2469–2486. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/shafagh.

[56] N. Singh, V. Meherhomji, and B. R. Chandavarkar. Automated versus Manual Approach of Web Application Penetration Testing. In *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–6, July 2020. doi: 10.1109/ICCCNT49239.2020.9225385.

[57] Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. John Wiley & Sons, Incorporated, Hoboken, UNITED STATES, 2011. ISBN 978-1-118-17522-4.

[58] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. cross site scripting prevention with dynamic data tainting and static analysis. January 2007.

[59] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391, May 2018. doi: 10.1109/SP.2018.00003.

[60] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In *2011 IEEE Symposium on Security and Privacy*, pages 465–480, Oakland, CA, USA, May 2011. IEEE. ISBN 978-1-4577-0147-4. doi: 10.1109/SP.2011.26. URL http://ieeexplore.ieee.org/document/5958046/.

[61] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. Explicating sdks: uncovering assumptions underlying secure authentication and authorization. pages 399–414, August 2013.

[62] M. Weulen Kranenbarg, T.J. Holt, and J. Van der Ham. Don't shoot the messenger! a criminological and computer science perspective on coordinated vulnerability disclosure. *Crime Science*, 7, 2018. doi: https://doi.org/10.1186/s40163-018-0090-8.

[63] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. AUTHSCOPE: Towards Automatic Discovery of Vulnerable Authorizations in Online Services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 799–813. Association for Computing Machinery, New York, NY, USA, October 2017. ISBN 978-1-4503-4946-8. URL http://doi.org/10.1145/3133956.3134089.