International Journal of Information Security manuscript No.
(will be inserted by the editor)

Cookie Baker: Gray-box Login Automation for Web Application Security Testing

Simone Bozzolan · Stefano Calzavara · Davide Porcu

Received: date / Accepted: date

Abstract We present Cookie Baker, the first gray-box login automation framework designed to enhance web application security testing. Cookie Baker is designed as a conservative extension of Cookie Hunter, a state-of-the-art black-box login automation tool. By combining static analysis and automated credential harvesting, Cookie Baker significantly increases the success rate of Cookie Hunter and improves the diversity of the available account types, thus making security testing more effective and realistic. Our experimental evaluation on public web applications shows that the additional capabilities of Cookie Baker make it able to automatically login on 4x more web applications than Cookie Hunter. This substantial improvement in login automation translates into greater code coverage during web crawling, ultimately leading to a higher rate of vulnerability detection. The integration of Cookie Baker with the Wapiti security scanner enables us to identify several new potential vulnerabilities in existing software, including two confirmed stored XSS. These findings highlight Cookie Baker's potential to enhance web application security testing, equipping security researchers and penetration testers with a powerful tool to uncover security flaws that would otherwise remain undetected.

S. Bozzolan
Università Ca' Foscari Venezia, Venice, Italy
E-mail: simone.bozzolan@unive.it

S. Calzavara
Università Ca' Foscari Venezia, Venice, Italy
E-mail: stefano.calzavara@unive.it

D. Porcu
Università Ca' Foscari Venezia, Venice, Italy
E-mail: davide.porcu@alumni.unive.it

Keywords Gray-box testing · Web authentication · Web security · Web automation

1 Introduction

Authentication is a necessary precondition to holistic security testing, because many web application functionalities are only accessible after login. For example, a recent study by Rautenstrauch et al. identified that the security landscape of web applications in the wild significantly changes before and after login [36]. This means that for a web application security analysis tool to be effective, it must be able to log into the application. While this can be done manually when testing just a few applications, as the developer or user can manually log in and provide the session cookies to the tool, the process becomes progressively more tedious and time-consuming as the number of applications to be tested grows. Therefore, automating the login process is essential to researchers interested in web application security, as manually logging into web applications to analyze is impractical at scale. Unfortunately, it is well known that automating the account creation and authentication process is a challenging task [12,16].

Prior work proposed black-box approaches to login automation based on carefully crafted heuristics to detect and interact with registration and login forms, whose effectiveness is limited. For example, Drakonakis et al. reported that their state-of-the-art tool Cookie Hunter managed to successfully register an account and perform login just on 13.7% of the websites where it identified a registration form [16]. However, there are many cases where a black-box approach is a sub-optimal choice in practice, because the application's source code and documentation are available for analysis. For in-

stance, security researchers often conduct their analysis over datasets of open-source and well-documented web applications [24, 33, 43]. Moreover, there is a subtle and relevant aspect to consider: security is not just about authentication, but also about authorization. Many web applications allow users to register with a generic "customer" role, but do not expose any public interface to create more privileged accounts, e.g., new administrators. Having access to different types of accounts is important to analyze the security posture of different components of the web application, e.g., to determine whether the admin panel is protected against Cross-Site Request Forgery (CSRF). Still, black-box techniques have no visibility of any account type that cannot be created through the web application front-end, such as highly privileged admin accounts that are normally created upon installation or using external scripts.

Contributions. We make the following contributions:

1. We perform a critical analysis of Cookie Hunter, a state-of-the-art black-box login automation framework. Our analysis shows that the black-box nature of Cookie Hunter unnecessarily limits its effectiveness and generality on web applications whose source code and documentation are available for analysis. We substantiate our discussion with concrete examples from real-world applications (Section 2).
2. We present a *gray-box* extension of Cookie Hunter, called Cookie Baker, which leverages access to the web application code and its companion documentation to extract complementary information, which can boost the effectiveness of login automation. By combining static analysis and automated credential harvesting, Cookie Baker allows Cookie Hunter to identify additional registration and login forms, as well as to gain access to web application functionality that was inaccessible before (Section 3).
3. We experimentally show on open-source web applications that the new analysis layer implemented in Cookie Baker is practically useful to boost the effectiveness of Cookie Hunter. Indeed, Cookie Baker can login on 4x more web applications than Cookie Hunter when tested on a dataset of 250 Python repositories from GitHub. Moreover, the integration of Cookie Baker with the Wapiti [42] security scanner enables us to identify several new potential vulnerabilities in our dataset, including two confirmed stored XSS. Additional experiments on a catalog of self-hosted web applications developed using different programming languages further confirm the generality of our findings (Section 4).

The main take-away messages of our study are summarized in Section 5.

2 Motivation

Cookie Hunter is a state-of-the-art tool for the automation of the account creation and authentication process on web applications [16]. A key feature of Cookie Hunter is its black-box nature, i.e., its ability to operate without any access to the server-side code of the web application. This feature is useful, because it makes Cookie Hunter applicable in a wide range of settings, e.g., for web measurements over popular websites [36]. Unfortunately, the limited visibility into web application internals makes Cookie Hunter also limited in its effectiveness: according to its authors, Cookie Hunter managed to successfully register an account and perform login just on 13.7% of the websites where it identified a registration form. We here critically evaluate different aspects of Cookie Hunter, using real-world GitHub applications from our experimental evaluation (Section 4) to substantiate our claims.

2.1 Limitations of Web Crawling

Cookie Hunter tries to discover registration and login forms by crawling the web application. Unfortunately, web crawling is a hard task and existing crawlers suffer from many limitations that undermine their coverage: a recent survey showed that there exists no single best-performing crawling configuration generalizing to different web applications [39]. This means that Cookie Hunter may fail to discover either the registration form or the login form of a web application due to the inherent limitations of web crawling.

For example, BDIC3023J-Software-Methodology-Q-A-Platform [41] is a forum designed to facilitate connections and knowledge sharing among university students. It has a clearly visible sign-up button; however, the crawler fails in the detection of a registration form, because the sign-up button dynamically transforms the login form into a sign-up form without reloading the page (see Figure 1). Since there is no page reload or URL change, the crawler does not detect any update when interacting with the sign-up button and, therefore, misses the appearance of a registration form.

Another example is sikteeri [26], a membership management system developed by Kapsi, a Finnish nonprofit organization that offers various web services, including shell accounts, email, web hosting, and DNS. In this case, Cookie Hunter fails to locate the login form because the login page is inaccessible from any of the web application's pages, meaning that even a perfect web crawler would be unable to detect the login form. This happens because the login functionality is only meant for administrators and not for reg-

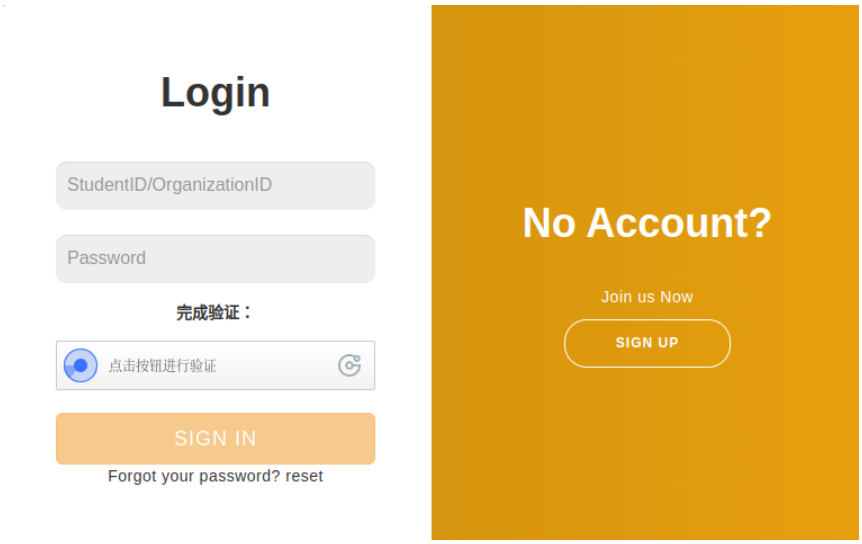

Fig. 1 In this example, when clicking on the SIGN UP button, the login form dynamically changes to a sign-up form without reloading the page. The sign-up form is then missed by Cookie Hunter.

ular users of the application. However, authenticating with an administrator account is essential to comprehensively scan an application for vulnerabilities. In fact, it can be argued that scanning for vulnerabilities using cookies from administrator accounts is even more critical than using those from regular user accounts, because vulnerabilities affecting administrators may lead to vertical privilege escalation.

2.2 Challenges of Automated Registration

Automating the registration process poses significant challenges, as real-world web applications often implement countermeasures to prevent automated account creation and enforce strict formatting requirements for specific form fields. For instance, CAPTCHAs are a common countermeasure against automated account creation, while fields such as passwords, home addresses, and Tax ID codes frequently have specific formatting requirements that are difficult to handle automatically without knowing them in advance. Cookie Hunter features a basic CAPTCHA solver capable of handling only the simplest types of CAPTCHAs. It also attempts to infer input field requirements from the HTML code. However, web applications are not forced to define these requirements in the HTML, as such validations are often implemented at the server side. We identified several examples where Cookie Hunter correctly identified the registration form, but was unable to complete the registration process.

For example, collectives [6] is a web application used to plan events within mountain sports clubs. Cookie Hunter was unable to complete the registration process because the application requires users to fill in a custom field called “license number” and validates the input upon form submission. The HTML of the registration page does not include any hint about formatting requirements for valid license numbers. Since Cookie Hunter attempts to populate all unrecognized custom fields with random data, it fails to meet the validation criteria and cannot complete the sign-up process.

Another example is racetime.gg [35], a web application for organizing races online, particularly focused on video games and speedruns (races conducted within video games). In this application, Cookie Hunter fails to complete the registration process due to the presence of a CAPTCHA (as shown in Figure 2), even though Cookie Hunter does include a basic CAPTCHA solver.

2.3 Limitations of Automated Registration

Many web applications are based on different types of accounts, but implement a registration form just for a subset of them. For example, a web application may have a single administrator and multiple regular users, hence no functionality in the web application front-end enables the creation of new administrators. This means that even a perfect automation of the registration process may provide insufficient visibility of the web application logic.

For example, Carbon 0 [14] is a web application designed to help individuals create carbon-negative lifestyles for themselves and their communities. Cookie Hunter successfully creates and logs into regular accounts; however, the web application also features a single administrator account. Since there is no front-end

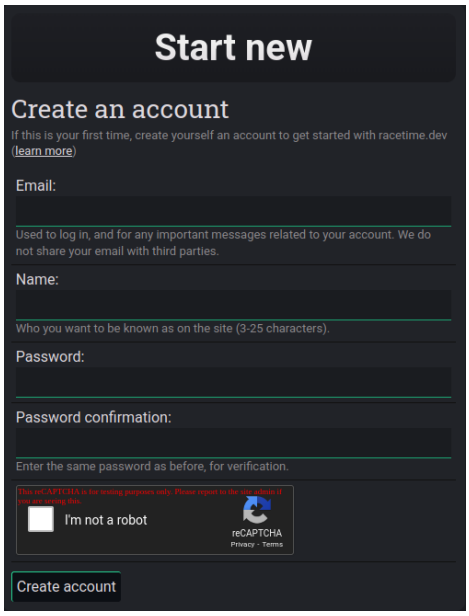

Fig. 2 Example of a failed registration due to the presence of a CAPTCHA.

functionality for creating new administrator accounts, Cookie Hunter is unable to collect cookies for this type of account. This limitation should be addressed, as it is essential to cover all account types supported by a web application when scanning for vulnerabilities. Certain parts of the application may be restricted to specific account types, and overlooking these areas could result in an incomplete security assessment of the web application.

3 Cookie Baker: Design and Implementation

We here explain the key ideas underlying the design of Cookie Baker and we provide full details about its implementation.

3.1 Architecture

Cookie Baker is designed as a gray-box extension of Cookie Hunter, which mitigates its main limitations by exploiting an improved visibility of the web application code and its companion documentation. The architecture of Cookie Baker is based on two modules, shown in Figure 3:

1. The *analysis* module uses static analysis of the web application code to improve the detection of both registration and login forms. Moreover, it analyzes both the source code and the available documentation to identify any hard-coded credentials that may be used to automate login, while bypassing the delicate challenges of the account creation process.
2. The *runtime* module executes the web applications and relies on Cookie Hunter to automate the login process. The results of the analysis module are fed into Cookie Hunter to improve its automation capabilities with the additional extracted information.

In the rest of the section, we discuss details about the analysis and runtime modules, along with their main components.

3.2 Analysis Module

The analysis module includes two main components: (*i*) a static analysis component to improve the detection of both registration and login forms, and (*ii*) a credential harvesting component, which is in charge of detecting any hard-coded credentials that may be used to automate login without performing account creation. The combination of the two components mitigates the main limitations affecting the effectiveness of Cookie Hunter.

3.2.1 Static Analysis

Cookie Baker uses static analysis to identify registration and login endpoints based on the invocation of account creation and authentication functions available in web development frameworks. The current implementation of Cookie Baker supports Python web applications developed using the popular Django and Flask frameworks. Our implementation supports a single programming language because static analysis is inherently language-based, hence working with multiple languages would require additional engineering effort to detect the relevant coding patterns. Indeed, the restriction to a single programming language is shared with other hybrid analysis tools like Chainsaw [3] and NAVEX [4]. We chose Python because it is extremely popular among web developers according to recent surveys [21,38] and it is natively supported by a popular static analyzer like CodeQL [20]. As for frameworks, we focus on Django and Flask, because they are the most popular web development frameworks for Python according to recent research [37]. Nevertheless, the ideas put forward in the design of Cookie Baker are general and can be ported to other settings with reasonable engineering efforts, in particular because CodeQL natively supports multiple programming languages, including other popular web development languages like Java and JavaScript.

In the following, we outline the primary patterns supported by our analysis. When considering the login

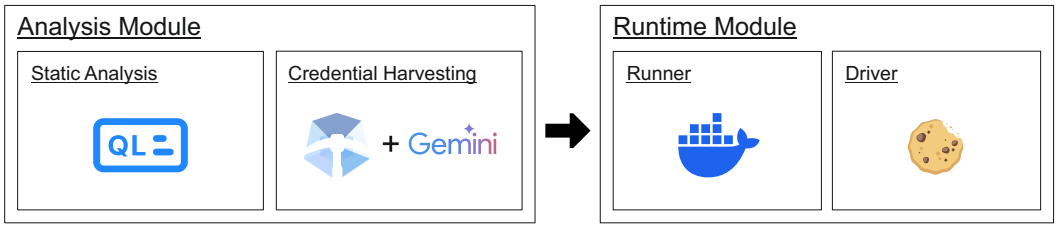

Fig. 3 Cookie Baker architecture.

functionality in Flask applications, we look for functions that invoke the `login_user` method of Flask-Login [28], which is the standard authentication library of Flask applications. For Django applications, we look for functions that either call the low-level `authenticate` method, invoke the `login` method, use Django's built-in `LoginView` class, or incorporate Django's built-in authentication URLs and functions by including them in the `urlpatterns` variable. These are all the standard authentication methods for Django applications that we are aware of.

Similarly, when looking for the sign-up functionality, we follow a comparable approach by identifying the most commonly used methods for handling user registration. For Flask applications, we look for functions that utilize Flask-WTF [19] forms containing at least one `PasswordField`. In Django applications, we search for functions using forms extending the `UserCreationForm` class. In addition, our queries also look for functions that create new users in the database. For Flask applications, this includes functions that utilize Flask-SQLAlchemy [18], Flask-Mongoengine [17], or Flask-Login's model class to save a new user object. The user object is typically defined as a new instance of a class named "user" (or similar) that extends one of the model classes. For Django applications, we look for functions that use the built-in `User` model class to create and save new user objects into the database.

After collecting all the relevant functions, i.e., those that perform user login, signup, or creation, we proceed to extract their associated URLs, i.e., the HTTP endpoints triggering such function invocations. For Flask, this involves analyzing the decorators of each function to identify those that declare the function URL. For Django, we examine the `urlpatterns` variable to locate the URLs corresponding to these functions. This process results in a list of functions handling login or sign-up, along with their respective URLs.

3.2.2 Credential Harvesting

Cookie Baker leverages two different approaches to automatically detect candidate authentication credentials. The reason behind this is that credentials may be located in either the source code or the documentation, each requiring a different type of analysis. Documentation is typically shorter than source code and consists of human-readable text, whereas source code is larger and requires the use of program analysis techniques, e.g., parsing, to identify embedded credentials. To conduct an extensive analysis of the source code, we require an efficient method. At the same time, the patterns to search for in the source code are generally more limited. In contrast, analyzing documentation requires a more advanced tool; one that, while not needing to be as efficient due to the smaller data size, is capable of processing natural language and identifying potential locations for credentials. By combining these two approaches, we leverage the strengths of both, enabling us to construct a comprehensive list of candidate credentials for each application. This feature is useful because it allows one to bypass the challenges of automated registration and enables the collection of credentials for different types of accounts, e.g., admin accounts often have default credentials advertised in the documentation.

Credential harvesting from the source code is based on regular expressions, which are generally well-suited for this task, as source code often follows predictable syntactic structures and patterns. For this purpose, we leverage Credential Digger [29], a static analysis tool designed specifically to detect hard-coded secrets and credentials in the source code. While Credential Digger also incorporates a machine learning model to reduce false positives identified through regular expressions, we chose to disable it due to its high computational cost in our experimental evaluation. As a result, we accept a higher number of false positives, but this is not an issue because credentials are eventually checked for validity in our pipeline. A significant challenge in processing Credential Digger's output is that usernames and passwords may not always appear within the same group extracted by the regular expression. This necessitates a more complex matching strategy that is implemented by our post-processing custom script. The idea behind the script is to pair each extracted username with all subsequent extracted passwords until the next username is encountered in a single file.

Credential harvesting from the documentation, instead, uses a Large Language Model (LLM) to extract credentials from human-written text such as README files and documentation, since this is a task where traditional regular expression-based approaches struggle, due to the complex and nuanced nature of human language. We decided to use Google Gemini through its API [22] for this task. We selected Gemini as our state-of-the-art LLM because it is freely accessible and offers generous token limits even for non-paid accounts, making it suitable for large-scale or complex evaluations without subscription constraints. Specifically, Google Gemini is used to analyze README files, `.rst` documentation files, and `.env` configuration files. These file types contain descriptive text, comments, and configuration details written in natural language, making them suitable targets for Gemini's natural language processing capabilities. By applying Gemini to these files, we complement the regex-based analysis performed by Credential Digger on the source code. In the following, we report the prompt used to query Gemini:

> You are an expert web application developer helping us locate credentials in documentation and configuration files that can be used to authenticate into the web application at hand. Be helpful and answer in detail while preferring to use information from reputable sources.
> I will provide a text file that may contain one or more usernames and passwords, or emails and passwords. Credentials can be hidden anywhere in the text: be smart! In particular, pay attention to code snippets that are delimited by the “‘ symbol. Please provide a list of all the credentials you find in there.
> Output JSON with this format "credentials": ["username":"...", "password":"..."].
> If there are no credentials, just return "credentials": [].
> Here's the content of the file:

3.3 Runtime Module

The runtime module includes two main components: (*i*) a runner component in charge of automatically executing the analyzed web applications, and (*ii*) a driver component, which executes Cookie Hunter after feeding it with the additional information extracted from the analysis module.

3.3.1 Runner Component

Automatically running web applications at scale is challenging due to the widespread adoption of custom installation procedures, yet Cookie Baker is gray-box in nature and it must successfully start the web applications on its local host to interact with them. We created a script to automatically detect and launch packaged copies of the target web applications, which were built using Docker, a widely used solution for application packaging. The simplest and most effective approach is to execute all `docker-compose.yml` and `Dockerfile` files until one that works is found. However, in some cases, this may not be sufficient, as the application might require some configuration to be done beforehand, or the `docker-compose.yml` and `Dockerfile` files might be misconfigured. To address common Docker deployment issues, our script incorporates several automated solutions. Specifically, it addresses scenarios where required Docker networks, which are virtual networks created within Docker that allow containers to communicate with each other, are missing. This is particularly important for applications that require multiple containers to function (e.g., separate containers for the database, backend, and frontend). The script automatically creates these networks when they are not present. Moreover, since developers often include multiple sample environment files within a project, and only a subset of these might work, the script iteratively attempts to launch the application using each available environment file. Finally, if no exposed ports are detected on container startup, indicating a potential misconfiguration by the developer, the script forcefully exposes all container ports to ensure the application is reachable.

3.3.2 Driver Component

The driver component is a patched version of Cookie Hunter designed to leverage the information extracted by the analysis module and improve the effectiveness of login automation. In particular, Cookie Hunter receives additional URLs and credentials, which are then fed into its existing crawling and login logic. First, we supplement the list of URLs found by Cookie Hunter's crawler with those discovered through static analysis. Second, after Cookie Hunter completes the sign-up attempt, we generate a list of candidate credentials by combining those extracted during the credentials harvesting phase with the ones created during the sign-up attempt and then make Cookie Hunter attempt the login using this list. Finally, after each successful login, we save the collected cookies, reset Cookie Hunter, and relaunch it with the remaining credentials. In this way, we can test the validity of multiple credentials and expose the existence of different account types.

4 Experimental Evaluation

We here quantify the improvements enabled by Cookie Baker over Cookie Hunter along multiple dimensions. Our experiments show that, when the source code and the companion documentation are available, the effectiveness of Cookie Hunter can be significantly improved through static analysis and credential harvesting. Moreover, the improvement in the login automation process leads to a significant increase in terms of vulnerability detection capabilities.

4.1 Experimental Setup

We performed our experimental evaluation on a dataset of Python web applications downloaded from GitHub. We focus on GitHub because it offers immediate access to the source code of the analyzed web applications. Using the GitHub REST API, we identified Python repositories that import the Django or Flask modules, which are the most popular web development frameworks for Python [37].

Following the approach in [9], we started with 1,999 repositories that use either the Flask or Django frameworks and satisfy the following criteria:

1. The most recent commit was made in 2020 or later, ensuring the project is not outdated.
2. They fall in the fourth quartile (top 25%) of the distribution of all repositories using either the Flask or Django frameworks for each of the following metrics:
   - *Number of stars*: a standard indicator of a repository's popularity;
   - *Number of contributors*: an estimate of both the popularity and complexity of the application;
   - *Number of commits*: demonstrates that the project has been actively developed, at least for a period of time;
   - *Source code*: we use the size of the Python code as a proxy for application complexity. We retain only projects that also include HTML and CSS, since our focus is exclusively on web applications.
3. They do not contain keywords (such as test, demo, ctf, exercise...) which strongly indicate that a repository may not host a real web application (e.g., it could be a library, tutorial, or capture-the-flag exercise). This maximizes the likelihood that we retain only repositories containing actual web applications.

This filtering guarantees the representativeness of our dataset, meaning it contains actual web applications intended to be deployed and used. To automate the execution of the downloaded web applications, we then restricted our focus to the 1,223 repositories including packaged copies of the target web applications based on `docker-compose.yml` or `Dockerfiles` files. Although the presence of these files is a promising indicator of the possibility of systematically launching such applications at scale, it is not a guarantee of success and potential reasons for failure abound. For example, Docker files might be outdated and unmaintained, Docker might have to be run after custom configuration steps, Docker images might be missing, etc.

Starting from the 1,223 repositories including Docker Compose files or Docker files, we successfully managed to launch 655 applications. We then filtered these applications, keeping only those that responded to at least one HTTP request with a status code below 400, providing preliminary evidence that they are functioning correctly. Following this filtering step, we obtained 250 applications, which we used for our experimental evaluation. Note that we do not know in advance how many of these applications implement a login functionality, hence the applications amenable for login automation may be less than 250.[1]

4.2 Account Creation and Login

We compare the effectiveness of Cookie Baker and Cookie Hunter in terms of successful logins. Observe that Cookie Baker is based on the same crawler as Cookie Hunter, hence differences in the results can only be attributed to the new components introduced in Cookie Baker rather than to randomness in crawling. We report results in Table 1, where we also show statistics for two intermediate configurations, i.e., the extension of Cookie Hunter with either the static analysis alone or the credential harvesting alone.

4.2.1 Cookie Hunter

We start by discussing the results for Cookie Hunter. The first observation we make is that Cookie Hunter is brittle, because a successful login requires multiple preconditions: a registration form must be found, a login form must be found, and the registration process must succeed. In our experimental evaluation, Cookie Hunter could find a login form on 106 applications, while it could find a registration form just on 38 applications. Interestingly, these are not a subset of the applications where a login form was detected, because there are just 25 applications where Cookie Hunter found both a registration form and a login form, hence registration was

[1] To determine this, we would need to manually verify each application for the presence of a login feature, which would entail substantial manual effort.

Table 1 Comparison between Cookie Hunter (CH) and Cookie Baker (CB). The intermediate columns CH+Static and CH+Creds report the performance of CH when including either the static analysis alone or the credential harvesting alone.

	CH	**CH+Static**	**CH+Creds**	**CB**
Registration form found	38	41	38	41
Login form found	106	112	106	112
Registration was attempted	25	30	25	30
Login was attempted	12	15	102	110
Login was successful	6	8	21	23

attempted. In the end, the actual login attempts were just 12, due to a number of failures in the registration process, and the entire workflow succeeded just on 6 applications.

4.2.2 Static Analysis

The inclusion of static analysis improves the effectiveness of Cookie Hunter by guiding a more effective detection of registration and login forms. The number of detected login forms increased from 106 to 112 (+6), while the number of detected registration forms increased from 38 to 41 (+3). Correspondingly, the number of registration attempts increased from 25 to 30 (+5) and the number of login attempts increased from 12 to 15 (+3). Most importantly, we saw a rise in the number of successful logins from 6 to 8 (+2). This limited increase can be explained by the empirical observation that most registration and login forms are easy to detect by crawling, likely because most web applications are designed to encourage users to create new accounts.

Still, we identified some applications where form detection failed. An example where static analysis aided Cookie Hunter in successfully performing a login is Pervane [40]. Pervane is a note taking and knowledge base building web application. Cookie Hunter was not able to find the sign-up page, as it is inaccessible from any of the web application's pages. However, with the help of static analysis, we were able to find the sign-up page URL and feed it to Cookie Hunter, who is then able to create an account and login into the web application, thus getting access to additional functionality for security testing.

4.2.3 Credential Harvesting

Although the inclusion of static analysis is useful for recovering some applications, the experimental results show that Cookie Hunter suffers from inherent and significant limitations in the automation of the registration process. The original version of Cookie Hunter attempted registration on 25 applications, but attempted login just on 12 applications, meaning that the success rate of the registration process is 48%. After introducing the static analysis component, we observe a similar success rate (50%).

We now shift our focus to the inclusion of credential harvesting, which allows Cookie Hunter to bypass the challenges of automated account creation. Compared to the original version of Cookie Hunter, the number of login attempts increased from 12 to 102 (+90), while the number of successful logins increased from 6 to 21 (+15). The success rate of the login process thus decreased from 50% to 21%, due to the presence of a number of invalid credentials. This is because our credential harvesting component was designed to collect as many potentially valid credentials as possible, accepting a number of false positives to minimize false negatives. The rationale of this approach is that by gathering a superset of potentially valid credentials, we avoid missing valid candidates, while the invalid ones are subsequently filtered out during the login attempts. This way, we are able to maximize the total number of successful logins. Indeed, when considering the absolute number of successful logins, the inclusion of credential harvesting has a significant impact.

An example application where credential harvesting was useful to perform a successful login is Gapps [30], a security compliance platform that makes it easy to track your progress against various security frameworks. In this case, Cookie Hunter is not able to find a registration form. However, when provided with the credentials extracted during the credential harvesting phase, it successfully logs in to the web application and gets access to its most interesting parts.

4.2.4 Cookie Baker

Finally, we observe that Cookie Baker, our extension of Cookie Hunter with both static analysis and credential harvesting, offers the best effectiveness in practice. Compared to the inclusion of credential harvesting alone, the number of login attempts increased from 102 to 110 (+8), while the number of successful logins increased from 21 to 23 (+2). The success rate of the login process thus remained the same (21%), but the combination of static analysis and credential harvesting

allowed Cookie Baker to attempt and automate login on the highest number of applications.

For example, Conreq [8] is a content requesting platform that allows users to request content from media organizers (Sonarr or Radarr). In this web application, Cookie Hunter, with the aid of static analysis, successfully identified a login form, but no login attempt was made. Since the original version of Cookie Hunter could not detect the login form in this application, relying solely on credential harvesting also resulted in no login attempts. It is only through the combination of static analysis and credential harvesting that we were able to perform a login attempt in this application.

4.3 Critical Analysis of the Results

At the end of our experiments, we can estimate the actual success rate of Cookie Hunter and Cookie Baker. In the original Cookie Hunter paper [16], the authors report that Cookie Hunter managed to successfully register an account and perform login on 13.7% of the websites where it identified a registration form. On our dataset, Cookie Hunter attempted registration on 25 applications and eventually logged in on 6 applications, leading to a higher success rate (24%). We can only make educated guesses about the increased effectiveness of Cookie Hunter on our dataset. While the difference in dataset size may account for the variation in results, another plausible explanation is that Cookie Hunter's original evaluation was conducted on popular live websites, whereas our evaluation focuses on a dataset of Python applications built with Django and Flask. This results in smaller diversity and improved compliance with disciplined web programming practices supported by web development frameworks.

Still, our run with Cookie Baker identified login forms on 112 applications, meaning that the 6 successful logins by Cookie Hunter are a very low number, leading to an estimated success rate of 5%. In turn, Cookie Baker performed 23 successful logins, corresponding to an estimated success rate of 21%. Room for further improvement is limited, because registration forms have been detected in combination with login forms just on 30 applications, leading to a successful registration in 15 cases. If the automation of the registration process was perfect, Cookie Baker would recover at most 15 applications, leading to a success rate of 34% (+13%). This tells us that *the registration process is a fundamental bottleneck of login automation tools like Cookie Hunter. The success rate of automated registration is limited (around 50% at best) and many web applications do not even offer a registration form.* The latter observation can be motivated by our focus on open-source applications designed to be self-hosted, which provides a useful vantage point for multiple reasons. First, security researchers may need to carry out security evaluations of local installations of popular web applications at scale [24,33,43]. For this line of work, automation through Cookie Hunter is likely bound to fail. Moreover, self-hosted web applications are different from top sites, yet they are routinely exposed on the Web. For example, the popular WordPress application does not have a registration form, but relies on a special admin account to allow the creation of new users with different roles. Without an existing database of known credentials like BugMeNot [11], analyzing such applications at scale beyond the authentication barrier would be impossible.

It is instructive to further look into our results to better understand the key limitations of Cookie Hunter and identify room for improvement. We already mentioned that the success rate of the registration process is relatively low, because Cookie Hunter performed 25 registration attempts, but attempted login just 12 times. We manually investigated the 13 cases of registration failures and observed the following: in 8 cases Cookie Hunter was unable to correctly fill the registration form, in 3 cases Cookie Hunter was unable to submit the registration form, and in the last 2 cases registration was successfully performed, but Cookie Hunter was unable to confirm it. This shows that (presumed) *registration failures are variegated in nature and there is no simple fix to the issue, in particular because registration forms can be complex and difficult to fill automatically.*

Finally, we observe that also the success rate of login attempts after a successful registration is rather low, because Cookie Hunter logged in successfully just in 6 out of 12 attempts. We then manually looked into the 6 failed attempts, observing that one login failed because of ineffective account activation over email, one login failed because it was attempted over the wrong login form, and one login succeeded, but Cookie Hunter failed to acknowledge that. The other 3 cases are false positives of our experimental evaluation due to generic reasons. If we filter out false positives, we conclude that 6 out of 9 legitimate login attempts were successfully executed and detected as such.

4.4 Effectiveness of Credential Harvesting

Our experimental evaluation showed that credential harvesting is particularly important to improve the effectiveness of login automation. In total, we identified 16 applications with valid credentials across all applications. This number would be hard to improve with the

existing pipeline, as our methodology prioritizes generating new candidate credentials, even at the cost of introducing some false positives that are later removed by testing their validity. To increase the number of working credentials, improvements to the form-filling capabilities and login state detection of Cookie Hunter would likely be necessary, because the login process of Cookie Hunter does not even start on some applications. Additionally, web applications may lack pre-existing accounts and require users to define their own usernames and passwords in custom configuration files.

Out of the two approaches used to extract credentials, harvesting credentials from source code with a simple regex-based method outperforms credential harvesting from documentation using Google Gemini's LLM. Of the 16 working credentials, only 2 were identified using Gemini, while 7 were found using Credential Digger, and the other 7 were discovered by both. These results highlight the necessity of combining both approaches to achieve optimal results. Moreover, while it might seem intuitive to assume that most credentials for pre-existing accounts would be found in documentation, our findings suggest this is not the case. Indeed, credentials extracted from the source code are likely associated with test accounts that remain active. While the existence of these accounts is beneficial for our purposes, they should be disabled before the web application is released. Such accounts may possess special privileges required for testing, which could be exploited by attackers, as identifying these credentials from the source code is straightforward. This further underlines the importance of our analysis in discovering these types of accounts.

An interesting benefit of credential harvesting is that it is possible to identify multiple credentials for the same application, which might expose the existence of different types of accounts. Out of the 23 applications where Cookie Baker was able to perform a successful login, we identified 22 applications with candidate credentials, including 16 with valid credentials. In 4 cases we were able to obtain valid credentials for different types of accounts. For example, A+ [1] is an interoperable and extendable Learning Management System that uses a combination of services to enable teachers to fully customize their e-learning materials and to flexibly combine different kinds of materials. In this web application we were able to login into four different types of accounts, all with different privileges: a student account, an assistant account, a teacher account, and the root account. Another example is Carbon 0 [14], which was one of our motivating cases (see Section 2.2). As previously reported, Cookie Hunter can successfully create and log into a regular account. However, this application also includes a single administrator account without the option to create additional ones. Thanks to the credentials collected during the credential harvesting phase, Cookie Hunter is now able to log into the administrator account as well.

4.5 Code Coverage and Vulnerability Detection

Finally, we show that Cookie Baker offers improved code coverage and vulnerability detection capabilities with respect to Cookie Hunter. We first integrate Cookie Baker with Wapiti [42], an open-source security scanner combining a web crawler with a vulnerability detection module. A similar integration could have been performed for other popular scanners like Burp Suite [34] (by implementing an extension) or OWASP ZAP [32] (by modifying its open-source code).

We start by measuring client-side code coverage, estimated in terms of distinct lines of JavaScript code reached through the web crawler available in Wapiti. Table 2 reports for the different applications where Cookie Baker was able to successfully login the code coverage observed with plain Wapiti (unauthenticated state), the code coverage enabled by the inclusion of Cookie Hunter, and the code coverage enabled by the adoption of Cookie Baker. With the original version of Cookie Hunter, we achieve a 37% improvement in code coverage, while Cookie Baker delivers a 111% improvement. This highlights the critical importance of authenticating into web applications during testing or vulnerability scanning, as authentication allows scanners to cover twice as many lines of client-side code. An additional observation is that, in three cases, plain Wapiti outperforms Cookie Baker in terms of code coverage. This occurs because, after authentication, some pages, such as login, sign-up, and reset password pages, become inaccessible. These pages can sometimes be larger and more complex than the pages accessible post-login. This underscores the necessity of scanning applications in both authenticated and unauthenticated states to achieve comprehensive results. Note that these uncommon scenarios are easily dealt with by running scans using plain Wapiti along with Cookie Baker.

To better appreciate the improvement in code coverage enabled by Cookie Baker, we focus on the 4 applications where we were able to obtain valid credentials for different types of accounts. Table 3 shows the overlap in code coverage for the different account types in the analyzed web applications. The table covers just three applications, because we were unable to login to the fourth one with Wapiti due to the implementation of a defense mechanism that invalidates cookies when the user agent changes, a situation that occurs in our

Table 2 Comparison of Cookie Hunter and Cookie Baker in terms of client-side code coverage (distinct lines of JavaScript code reached through web crawling).

Application	Wapiti	Cookie Hunter	Cookie Baker
App 1	68,614	**115,242**	**115,242**
App 2	**1,514,989**	1,211,592	1,211,592
App 3	0	0	**990,188**
App 4	3,007,657	3,007,657	**4,199,783**
App 5	**344,355**	**344,355**	4,066
App 6	1,160,717	1,160,717	1,160,717
App 7	4,665,974	**17,276,290**	**17,276,290**
App 8	326,671	326,671	326,671
App 9	1,906,870	1,906,870	**3,388,994**
App 10	561,320	561,320	561,320
App 11	37,622	37,622	**1,564,113**
App 12	**228,973**	**228,973**	183,418
App 13	1,160,717	1,160,717	1,160,717
App 14	50,747,109	**67,968,349**	**67,968,349**
App 15	1,403,335	1,403,335	**1,403,768**
App 16	10,751,095	10,751,095	**37,943,771**
App 17	730,210	**10,385,911**	**10,385,911**
App 18	23,080,451	23,080,451	**23,082,796**
App 19	259,994	**1,069,533**	**1,069,533**
App 20	362,427	362,427	**29,914,767**
App 21	4,085,790	4,085,790	4,085,790
App 22	787,370	787,370	**8,642,231**
App 23	27,104	27,104	**9,803,089**
Total	107,219,364	147,259,391	**226,443,116**

measurement methodology. In all reported cases, we observe that different credentials provide access to distinct parts of the web application that are inaccessible with other credentials, because the amount of unique lines of code for each account type is significant. This demonstrates that not only is it essential to scan web applications both in authenticated and unauthenticated states, but also to perform scans using different account types.

The increase in code coverage enabled by Cookie Baker pays dividends in terms of vulnerability detection. We scanned the applications using Wapiti in its default configuration, modified as follows: (*i*) we disabled all modules related to HTTPS vulnerabilities, as our web applications run on the local host, and (*ii*) we enabled modules for detecting CSRF and XML external entities (XXE) vulnerabilities, which were not enabled by default. Table 4 presents the vulnerabilities detected during scans performed in two configurations: with plain Wapiti (unauthenticated state), and using cookies collected by Cookie Baker. We omit the results of Wapiti with cookies collected by Cookie Hunter, as these would be a subset of the cookies collected by Cookie Baker. In the table, we report only the vulnerabilities that Wapiti was able to detect in either configuration across all applications. As expected, logging into each application yields the most comprehensive results. Comparing Cookie Baker to plain Wapiti, we observe a 37% increase in vulnerability detection. To validate our findings, we manually analyzed the additional vulnerabilities detected during scans with Cookie Baker, focusing on the most critical types: stored XSS and command injection. These two types of vulnerabilities are dangerous as they allow attackers to execute arbitrary code on the client side and server side, respectively. Both stored XSS vulnerabilities were confirmed, while the command injection vulnerability turned out to be a false positive. The false positive was due to Wapiti incorrectly reporting a command injection in the request for the jQuery library. These results underscore the importance of logging in during vulnerability scanning.

The two stored XSS vulnerabilities were discovered in the same web application.[2] The application is designed to facilitate teaching and learning by supporting asynchronous student interactions. Unlike traditional teacher-centered methods, it promotes peer-to-peer engagement, enabling students to both teach and learn from one another. The project is actively maintained by multiple contributors and developed within a prominent academic research group. Specifically, the application includes a forum-like feature where users can post questions and others can reply. However, the form used to submit questions does not implement any protection against XSS attacks, allowing users to input JavaScript

[2] The actual name of the application has been omitted for ethical and legal reasons, because the vulnerabilities have not been patched yet.

Table 3 Comparison between different credentials in terms of client-side code coverage (distinct lines of JavaScript code reached through web crawling).

App	Credentials	Overlap	Unique to Type 1	Unique to Type 2
App 1	Cred 1 & No Auth	2,237,428	1,936,722	762,867
	Cred 2 & No Auth	2,233,396	1,362,716	766,788
	Cred 3 & No Auth	2,237,428	207,122	762,534
	Cred 4 & No Auth	2,241,113	218,447	759,421
	Cred 1 & Cred 4	2,444,852	1,729,362	15,119
	Cred 1 & Cred 2	3,588,532	585,147	8,443
	Cred 3 & Cred 1	2,440,718	4,651	1,733,318
	Cred 3 & Cred 2	2,440,397	4,676	1,155,616
	Cred 3 & Cred 4	2,437,033	8,336	22,760
	Cred 4 & Cred 2	2,435,757	23,679	1,161,218
App 2	Cred 1 & No Auth	1,758,682	15,295,284	2,666,079
	Cred 2 & No Auth	1,758,622	15,294,143	2,664,763
	Cred 3 & No Auth	1,553,275	15,876,148	3,126,873
	Cred 1 & Cred 2	12,372,330	6,826,513	6,973,075
	Cred 1 & Cred 3	13,910,231	3,124,352	3,119,452
	Cred 2 & Cred 3	15,120,459	4,609,932	4,763,782
App 3	Cred 1 & No Auth	1,356,194	2,029,829	550,676
	Cred 2 & No Auth	1,394,517	2,634,532	520,158
	Cred 3 & No Auth	1,376,086	2,228,220	536,968
	Cred 4 & No Auth	1,394,517	2,212,900	524,860
	Cred 1 & Cred 2	3,148,503	221,149	901,916
	Cred 1 & Cred 4	3,169,102	210,018	416,622
	Cred 3 & Cred 1	3,151,078	438,091	185,419
	Cred 3 & Cred 2	3,150,371	475,131	903,226
	Cred 3 & Cred 4	3,557,786	73,647	27,579
	Cred 4 & Cred 2	3,189,797	401,484	876,070

Table 4 Comparison of Wapiti and Cookie Baker in terms of found vulnerabilities.

Vulnerability	Wapiti	Cookie Baker
MIME type confusion	11	11
Stored XSS	0	**2**
CSRF	13	**33**
Lack of clickjacking protection	12	12
Command injection	0	**1**
CSP misconfiguration	27	27
Total	63	**86**

code that is subsequently executed in the browsers of anyone viewing the question. The form for submitting answers suffers from the same issue, as it does not validate or sanitize user input. These vulnerabilities allow attackers to steal user data and execute arbitrary code in victims' browsers. We responsibly disclosed these issues to the developers on January 14, 2025. However, as of October 28, 2025, we have not received any response.

4.6 Beyond Python

Our analysis showed that Cookie Baker is a useful extension of Cookie Hunter, because it increases the number of successful logins, code coverage, and vulnerability detection in our dataset of Python applications. We focused on Python alone because static analysis is inherently language-based; indeed, other gray-box analysis tools support a single programming language like Cookie Baker [3,4]. Nevertheless, the key ideas underlying Cookie Baker can be generalized to different programming languages and frameworks with appropriate engineering efforts. The static analysis component of Cookie Baker uses CodeQL, which supports multiple programming languages, including other popular web development languages like Java and JavaScript. The credential harvesting component, instead, is language-independent and can be reused as is. The regex-based approach applies general expressions to the source code, treating it as plain text files without relying on language-specific patterns, as outlined in the original paper [29]. Similarly, the LLM-based approach processes configuration files and companion documentation written in natural language, and is inherently language-independent.

To show the benefits of Cookie Baker beyond the Python ecosystem, we leverage two observations supported by our experimental data. First, credential harvesting is more important than static analysis for successful login automation. Since credential harvesting is language-independent, we can reap its benefits also for web applications developed without using Python.

Moreover, Cookie Baker is particularly useful for self-hosted web applications, which represent the ideal application target of our tool. We then carry out a different experiment where we use Cookie Baker to perform login automation over the web applications available in the Awesome-Selfhosted catalog [7]. The catalog includes 1,197 web applications intended to be self-hosted and developed using different programming languages, such as Java, JavaScript, and PHP. In total, we found 528 applications with associated Docker Compose files or Docker files, including 209 applications that we managed to run correctly. The distribution of the programming languages used to develop the 209 applications that we managed to run correctly is shown in Figure 4.

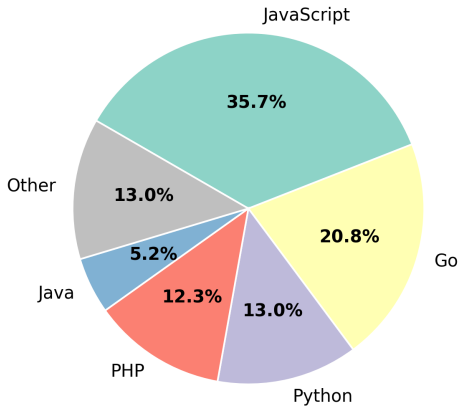

Fig. 4 Distribution of programming languages used to develop the applications that we were able to run successfully from the Awesome-Selfhosted dataset.

Table 5 reports the results of our comparison between Cookie Hunter and Cookie Baker on the Awesome-Selfhosted catalog. Cookie Hunter attempted registration on 17 applications and successfully logged in on 6 of them, resulting in a login success rate of 35%, which is better than the 24% observed on the Python dataset (25 attempted registrations, 6 successful logins). Cookie Baker, instead, identified login forms on 70 applications and achieved 11 successful logins, corresponding to a success rate of 16%, which closely matches the previously observed 19% on the Python dataset (112 applications with identified login forms, 21 successful logins when just credential harvesting is activated for Cookie Baker). The overall consistency of the numbers suggests that the heuristics and dataset construction methodology adopted for Python repositories were effective in producing a representative collection of web applications, comparable to a manually curated dataset of diverse web applications.

Table 5 Comparison between Cookie Hunter (CH) and Cookie Baker (CB) on the Awesome-Selfhosted catalog.

	CH	CB
Registration form found	43	43
Login form found	70	70
Registration was attempted	17	17
Login was attempted	9	70
Login was successful	6	11

When examining the impact of the harvested credentials on the number of successful logins, we observe that the benefits are less pronounced on the Awesome-Selfhosted dataset than on the Python dataset. Specifically, in the Python dataset, the use of harvested credentials enabled successful authentication on 15 additional web applications (a 3.5x increase, from 6 to 21), whereas in the Awesome-Selfhosted dataset the number of successful logins increased by only 5 (a 1.8x increase, from 6 to 11). Although its benefits are less pronounced, Cookie Baker still improves the effectiveness of Cookie Hunter by a factor of 1.8x, which is far from marginal, especially for large-scale experiments. This further highlights the significance of Cookie Baker's additional modules in enhancing Cookie Hunter's login capabilities, enabling it to overcome its limitations and authenticate across a larger and more diverse set of web applications (when gray-box analysis is feasible).

4.7 Ablation Study

We designed Cookie Baker after a number of preliminary evaluations of its effectiveness and efficiency. To better motivate the presented solution, we perform an ablation study of how alternative designs of Cookie Baker would impact the performance of the tool. In particular, we consider the following alternatives:

1. *Credential Digger configuration*: Credential Digger comes with an associated machine learning model to reduce the number of invalid credentials extracted by regular expressions [29]. In our implementation of Cookie Baker, we disabled this model due to its computational cost based on preliminary experiments. We now assess how the activation of the machine learning model would impact on the overall execution time of the tool.
2. *Supported file types*: Cookie Baker uses Gemini to extract credentials from different types of files, including README files, `.rst` documentation files, and `.env` configuration files. Since credentials may be extracted from other file types as well, we now extend the set of files to include `.md` files, `.ini` files, `.yaml` files, and `.json` files. The more files we in-

Table 6 The table shows how different changes to the original Cookie Baker design (see Section 4.7) impact on the effectiveness and the efficiency of the tool.

	Original	**Credential Digger with ML**	**Extended Gemini**	**Default Credentials**	**All Credentials**
Total candidate credentials	32,024	1,175	34,073	47,864	49,913
Candidate credentials on apps where login was attempted	5,143	215	5,459	6,853	7,169
Candidate credentials on apps where login was successful	527	41	579	917	942
Valid credentials	11	11	12	12	13
Applications with valid credentials	11	11	12	12	13
Credential harvesting time (in hours)	2.8 h	65.7 h	4.2 h	2.8 h	4.2 h
Credential testing time (in hours)	38.5 h	1.5 h	40.9 h	51.3 h	53.7 h
Performance overhead	-	+62.7 %	+9.2 %	+31.0 %	+40.2 %

clude, the more credentials we can find, but with a higher computational cost.

3. *Credential extraction*: Cookie Baker may be extended to use known popular credentials in addition to those obtained through credential extraction. We here try to automate login attempts by using the top 5 passwords and usernames available in the OWASP SecLists database [31]. The set of default credentials built comprises all 25 combinations formed by pairing the top-5 usernames with the top-5 passwords, plus 5 additional credentials in which each top username is reused as the password, for a total of 30 credentials.

Our analysis is conducted on the Awesome-Selfhosted dataset for its language-agnostic nature, as we focus on the credential harvesting component, which is also language-agnostic. The ablation study sheds light on an inherent trade-off: every additional candidate credential increases the chance of a successful login, but has a measurable impact on the execution time of Cookie Baker, in terms of both credential harvesting and testing.

The results of our ablation study are presented in Table 6. The table presents five key metrics: the total number of candidate credentials, the number of credentials used by Cookie Baker to attempt logins, the number of credentials associated with applications where Cookie Baker successfully logged in, the total number of valid (working) credentials, and the total number of applications with valid credentials. We then report the total time required to harvest credentials under different configurations, the total time spent by Cookie Baker testing credentials during login attempts, and the performance overhead introduced by each configuration relative to the original version of Cookie Baker.

As the table shows, reducing the time wasted testing invalid credentials requires minimizing false positives during the credential harvesting phase, but this comes at a performance cost. When enabling Credential Digger's machine learning model, a transformer based on RoBERTa [27], the total number of valid credentials remains unchanged, but the total number of credentials used on applications where login attempts were made drops from 5,143 to 215 (4.2%). This demonstrates that the model is very effective at reducing false positives. However, while the time spent testing invalid credentials decreases from 38.5 hours to 1.5 hours, the credential harvesting time increases from 2.8 hours to 65.7 hours. In the end, the total running time increases by 62.7%. This highlights that minimizing false positives as much as possible is not always beneficial, since removing them is more expensive than simply testing the credentials, leading to reduced efficiency. As our objective is to maximize successful logins while minimizing execution time, the optimal strategy is to balance the time spent harvesting credentials and the time wasted testing invalid ones, prioritizing overall efficiency rather than raw precision. Of course, with more powerful hardware, the transformer model would run faster (in this case, it was executed on a machine with 32 GB of RAM, a 12-core AMD Ryzen 5 CPU, and no GPU), but such resources are not always available and come at an additional cost.

Additionally, the results show that extending Gemini's analysis to support additional file types and adding a default list of popular credentials both increase the total number of successful logins by one. However, extending Gemini's analysis incurs significantly less performance overhead (9.2%) compared to adding default credentials (31.0%). When combining both approaches the total number of successful logins increases from 11 to 13 (+2), at the cost of a 40.2% performance overhead. This shows that while it is possible to further

boost successful logins by extracting and testing more candidate credentials, this comes at a cost in terms of time and efficiency, which must be considered when designing the tool. Consequently, since our goal was to run a large-scale analysis, we chose the most efficient components to maximize successful logins while minimizing execution time. Therefore, we opted not to use default credentials and implemented the tool with the non-extended version of Gemini, along with Credential Digger with its machine learning model disabled. Although this resulted in losing two additional successful logins, it reduced execution time by 40.2%, making the tool better suited for large-scale analysis. Nevertheless, thanks to Cookie Baker's modular design, users can enable all modules if time and efficiency are not a concern, thereby maximizing the number of successful logins.

5 Discussion

We here summarize the main take-away messages of our study and conclude by discussing compliance with research best practices.

5.1 Take-Away Messages

Our research confirms and further highlights the complexity of automating login at scale [12,16,23]. Prior work tested login automation frameworks on live websites and justified the complexity of login automation as the result of different, variegate programming practices observed in the wild. Our analysis, instead, largely focused on Python applications developed with Django and Flask, which offer much smaller diversity and rely on disciplined authentication practices, yet this does not make the problem of login automation trivial to solve. Out of 112 applications providing a login form, a state-of-the-art solution like Cookie Hunter could automate registration and login just on 6 applications. The registration process is a fundamental bottleneck of login automation tools like Cookie Hunter, because the success rate of automated registration is limited (around 50%). Registration failures are diverse in nature, making them challenging to address with a single solution. Even when the registration process does not involve complex interactions like email-based activation, registration forms are hard to fill and submit automatically.

An important new insight of our research is that black-box login automation frameworks like Cookie Hunter are inappropriate for assessing the security of self-hosted web applications. Such applications often do not even offer a registration form, because accounts are created upon installation or using external tools, meaning that even perfect automation of the registration process would not help a comprehensive security analysis. Self-hosted web applications are interesting targets for web security research, because their code is publicly available and they are routinely deployed on the Internet, e.g., as in the case of content management systems. Prior work on web application security indeed tested the effectiveness of vulnerability detection techniques using self-hosted web applications [24, 33, 43]. Luckily, automated credential harvesting can effectively recover valid credentials for self-hosted web applications: in our experimental evaluation on Python applications, we identified valid credentials for 14% of the analyzed repositories. These credentials can also be associated with different types of accounts, thus increasing code coverage and making security evaluation more comprehensive. For example, in A+ [1] we successfully logged into four distinct account types, each granting access to different sections of the web application, resulting in significant differences in the code covered using the different account types. Our findings largely generalize beyond the Python ecosystem, as discussed in Section 4.6.

In the end, we conclude that Cookie Hunter is appropriate for large-scale web security measurements [15], but sub-optimal on smaller datasets of web applications such as those used in the literature on web vulnerability detection [24, 33, 43]. The low success rate of Cookie Hunter can be accepted when working with millions of websites whose source code is not publicly available, because Cookie Hunter still allows researchers to measure security trends on the Web. Nevertheless, Cookie Hunter does not leverage useful information such as the source code and the companion documentation, which is normally available when doing research on vulnerability detection, e.g., several existing detection approaches are static or hybrid, thus presupposing the availability of the source code. Since passing the login barrier is a prerequisite of meaningful security assessments, as shown by prior work [36] and confirmed by our study, a solution like Cookie Baker is more useful than Cookie Hunter when working with smaller datasets of open-source web applications.

5.2 Open Science and Research Ethics

We make our code and data publicly available to improve the transparency and reproducibility of our research [10]. Please note that our repository does not include the updated Cookie Hunter code, because the original version of Cookie Hunter is only available upon request to its authors. Note that some fixes had to be implemented to run Cookie Hunter on locally hosted

web applications, such as how the different logs and results were saved. In addition, a number of changes were implemented in order to be able to use the results of the analysis module, albeit in a limited capacity: in particular, only the driver component is new. These modifications are available upon request, provided the requester has access to the original Cookie Hunter code. After getting access to our modified version of Cookie Hunter, our results can be reproduced by downloading the reported applications from GitHub and using our scripts as documented in the online repository.

Our research follows ethical best practices. All the experimental evaluation was performed on local installations of publicly available web applications, so we never interacted with live websites. We responsibly disclosed the confirmed security vulnerabilities to the interested parties.

6 Related Work

Automating the registration and login process is a hot topic nowadays, because the security posture of web applications significantly changes before and after authentication [36]. Early work on the security of web authentication manually performed the registration process and used simple techniques to automate the login process [13]. Jonker et al. proposed Shepherd, the first login automation framework designed to enable large-scale security scans of websites requiring login [23]. Shepherd was used to perform a large-scale security measurement of web sessions in the wild, providing a comprehensive analysis motivating the utility of automated login approaches [12]. Since Shepherd does not automate the account creation process, later work in the field of web authentication investigated techniques to further expand the capabilities of security scanners and perform registration automatically. Drakonakis et al. developed Cookie Hunter, the first fully automated login automation framework supporting the challenges of new account creation [16]. Unfortunately, the success rate of Cookie Hunter is rather low (the original paper estimates 13.7%), meaning that the tool is only amenable to large-scale security measurements - indeed, the original study started from a set of 1.5M domains. Raising the success rate of Cookie Hunter is important to enable targeted studies on smaller datasets of open-source web applications, like those typically considered in the literature on vulnerability detection [24,33,43]. For such studies, a gray-box solution taking advantage of the source code and its companion documentation, like Cookie Baker, is more effective in practice.

Besides Cookie Hunter, other solutions dealing with automated account creation and login exist. Alroomi and Li used a similar system in their evaluation of login policies and password creation policies at scale [2, 5], but their tools are not publicly available for evaluation. Kubicek et al. presented an automated framework for website registration to study GDPR compliance [25]. Their tool does not automate login, just the registration to websites and newsletters, given its different focus on privacy concerns. Rautenstrauch et al. implemented a semi-automated, open-source login automation framework similar to Cookie Hunter [36]. This was motivated by the limited availability of competitor solutions, which cannot be readily used by the research community. Still, they acknowledge that the heuristics used in their tool are similar to those implemented in Cookie Hunter [36], hence we do not expect fundamentally different pictures to be drawn when considering their framework in place of Cookie Hunter.

7 Conclusion

Automating the login process at scale is important to enable meaningful security assessments of web applications implementing password-based authentication. Unfortunately, prior work established that automating account creation and login is a challenging problem, leading to sub-optimal performance on real-world web applications [16,23]. The limited effectiveness of existing login automation frameworks can be accepted for large-scale web measurements on popular websites, but the current state of the art falls short when dealing with smaller datasets of applications requiring a targeted analysis, as is common, e.g., in vulnerability detection research. Our study shows that the challenges of login automation are not inherent to the variegated and complex nature of live websites, but also affect self-hosted web applications developed using well-known web development frameworks, which are expected to enforce disciplined programming practices amenable to automation. Luckily, when the source code and the companion documentation of a web application are available, traditional black-box approaches to login automation can be extended into gray-box solutions leading to higher effectiveness in practice. Our gray-box extension of Cookie Hunter, called Cookie Baker, is able to increase the success rate of Cookie Hunter up to a factor of 4x, extending its code coverage and enabling a more effective detection of web vulnerabilities.

We foresee our research as a fundamental building block of future web application scanners based on a combination of static and dynamic analysis. Cookie Baker is already useful to increase the effectiveness of login automation and enable more credible post-login security assessments, however our work also sheds light

on the complexity of running web applications automatically at scale. In this work, we explored the use of Docker as an effective enabler of self-hosting automation, yet we also identified major challenges in automating the execution of Docker images. Improving the effectiveness of self-hosting automation would thus be an interesting avenue for future work. Moreover, we plan to explore the introduction of code injection capabilities within Cookie Baker, which may be used to automate the process of account creation through the selective invocation of sign-up functions detected through static analysis.

Data Availability

All data and code supporting the findings of this study are available at the following URL: `https://github.com/Asterius27/CookieBaker`.

Declarations

Funding. This research was supported by Agenzia per la cybersicurezza nazionale under the 2024-2025 funding programme for promotion of XL cycle PhD research in cybersecurity – CUP N.H71J24001710005. The research also acknowledges support from project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. The views expressed are those of the authors and do not necessarily represent the funding institutions.

Competing interests. The authors have no relevant financial or non-financial interests to disclose.

Ethical approval. The authors declare full compliance with ethical standards. This article does not contain any studies involving humans or animals performed by any of the authors.

Acknowledgement. This version of the article has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: `https://dx.doi.org/10.1007/s10207-025-01168-z`. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use `https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms`

References

1. A+ LMS. A+. `https://github.com/apluslms/a-plus`.
2. Suood Abdulaziz Al-Roomi and Frank Li. A large-scale measurement of website login policies. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 2061–2078. USENIX Association, 2023.
3. Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and V. N. Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 641–652. ACM, 2016.
4. Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V. N. Venkatakrishnan. NAVEX: precise and scalable exploit generation for dynamic web applications. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 377–392. USENIX Association, 2018.
5. Suood Alroomi and Frank Li. Measuring website password creation policies at scale. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 3108–3122. ACM, 2023.
6. Club Alpin Annecy. collectives. `https://github.com/Club-Alpin-Annecy/collectives`.
7. Awesome-Selfhosted. Awesome-Selfhosted. `https://github.com/awesome-selfhosted/awesome-selfhosted`.
8. Mark Bakhit. Conreq. `https://github.com/Archmonger/Conreq`.
9. Simone Bozzolan, Stefano Calzavara, Florian Hantke, and Ben Stock. Behind the curtain: A server-side view of web session security. In *IEEE Secure Development Conference, SecDev 2025, Indianapolis, IN, USA, October 14-16, 2025*. IEEE, 2025.
10. Simone Bozzolan, Stefano Calzavara, and Davide Porcu. Artifacts. `https://github.com/Asterius27/CookieBaker`.
11. BugMeNot. BugMeNot: find and share logins. `https://bugmenot.com/`.
12. Stefano Calzavara, Hugo Jonker, Benjamin Krumnow, and Alvise Rabitti. Measuring web session security at scale. *Comput. Secur.*, 111:102472, 2021.
13. Stefano Calzavara, Gabriele Tolomei, Andrea Casini, Michele Bugliesi, and Salvatore Orlando. A supervised learning approach to protect client authentication on the web. *ACM Trans. Web*, 9(3):15:1–15:30, 2015.
14. Carbon0-Games. Carbon0. `https://github.com/Carbon0-Games/carbon0-web-app`.
15. Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. Reproducibility and replicability of web measurement studies. In Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini, editors, *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, pages 533–544. ACM, 2022.
16. Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for

web authentication and authorization flaws. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1953–1970. ACM, 2020.
17. Flask-MongoEngine. Flask-MongoEngine documentation. `https://docs.mongoengine.org/projects/flask-mongoengine/en/latest/index.html`.
18. Flask-SQLAlchemy. Flask-SQLAlchemy Documentation (3.1.x). `https://flask-sqlalchemy.readthedocs.io/en/stable/`.
19. Flask-WTF. Flask-WTF Documentation (1.2.x). `https://flask-wtf.readthedocs.io/en/1.2.x/`.
20. GitHub. CodeQL. `https://codeql.github.com/`.
21. GitHub. The top programming languages. `https://octoverse.github.com/2022/top-programming-languages`, 2022.
22. Google. Google AI for Developers. `https://ai.google.dev/`.
23. Hugo Jonker, Jelmer Kalkman, Benjamin Krumnow, Marc Sleegers, and Alan Verresen. Shepherd: Enabling automatic and large-scale login security studies. *CoRR*, abs/1808.00840, 2018.
24. Soheil Khodayari and Giancarlo Pellegrino. JAW: studying client-side CSRF with hybrid property graphs and declarative traversals. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2525–2542. USENIX Association, 2021.
25. Karel Kubicek, Jakob Merane, Ahmed Bouhoula, and David A. Basin. Automating website registration for studying GDPR compliance. In Tat-Seng Chua, Chong-Wah Ngo, Ravi Kumar, Hady W. Lauw, and Roy Ka-Wei Lee, editors, *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*, pages 1295–1306. ACM, 2024.
26. Kapsi Internet käyttäjät ry. sikteeri. `https://github.com/kapsiry/sikteeri`.
27. Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.
28. Flask Login. Flask-Login 0.7.0 documentation. `https://flask-login.readthedocs.io/en/latest/`.
29. Sofiane Lounici, Marco Rosa, Carlo Maria Negri, Slim Trabelsi, and Melek Önen. Optimizing leak detection in open-source platforms with machine learning techniques. In Paolo Mori, Gabriele Lenzini, and Steven Furnell, editors, *Proceedings of the 7th International Conference on Information Systems Security and Privacy, ICISSP 2021, Online Streaming, February 11-13, 2021*, pages 145–159. SCITEPRESS, 2021.
30. Brendan Marshall. Gapps. `https://github.com/bmarsh9/gapps`.
31. Daniel Miessler, Jason Haddix, Ignacio Portal, and g0tmi1k. SecLists. `https://github.com/danielmiessler/SecLists`.
32. OWASP. OWASP ZAP (Zed Attack Proxy): a dynamic web application security testing tool. `https://www.zaproxy.org/`.
33. Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. Deemon: Detecting CSRF with dynamic analysis and property graphs. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1757–1771. ACM, 2017.
34. PortSwigger. Burp Suite: a software tool for security assessment and penetration testing of web applications. `https://portswigger.net/burp`.
35. racetime.gg. racetime.gg. `https://github.com/racetimeGG/racetime-app`.
36. Jannis Rautenstrauch, Metodi Mitkov, Thomas Helbrecht, Lorenz Hetterich, and Ben Stock. To auth or not to auth? A comparative analysis of the pre- and post-login security landscape. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 1500–1516. IEEE, 2024.
37. Marco Squarcina, Pedro Adão, Lorenzo Veronese, and Matteo Maffei. Cookie crumbles: Breaking and fixing web session integrity. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 5539–5556. USENIX Association, 2023.
38. Stack Overflow. Stack Overflow Developer Survey 2023. `https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023`, 2023.
39. Aleksei Stafeev and Giancarlo Pellegrino. Sok: State of the krawlers - evaluating the effectiveness of crawling algorithms for web security measurements. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.
40. Hakan Uysal. pervane. `https://github.com/hakanu/pervane`.
41. Yuyang Wang. Student Exchange Forum of BJUT. `https://github.com/echo-cool/BDIC3023J-Software-Methodology-Q-A-Platform`.
42. Wapiti Web Application Scanner. Wapiti: a Free and Open-Source web-application vulnerability scanner in Python. `https://wapiti-scanner.github.io/`.
43. Malte Wessels, Simon Koch, Giancarlo Pellegrino, and Martin Johns. SSRF vs. developers: A study of ssrf-defenses in PHP applications. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.