EAR Scanner: Suite of Execution after Redirection Detection Tools

Aya Habbas ahabbas1@jh.edu

Emily Berger eberge11@jh.edu

Yujian He yhe99@jh.edu

Abstract

The EAR Scanner Tools project presents a comprehensive suite of security utilities designed for the detection and analysis of Execution after Redirection (EAR) vulnerabilities in web applications. The project consists of two primary components: a Chrome extension for immediate vulnerability scanning of active web pages, and a robust fuzzing scanner tailored for subdomain discovery and scanning. To enhance efficiency and reduce repetitive scanning, the project incorporates a MongoDB [9] database to store scan results, utilising cached data for repeated scans within a seven-day timeframe. This innovative approach not only simplifies the process of identifying potential EAR vulnerabilities but also contributes significantly to proactive web security management.

This project leverages HTML and CSS for frontend development, and Python for backend development, incorporating the *gobuster* [8] tool for effective subdomain scanning. The EAR Scanner Tools stand out as an essential asset in web security, offering an accessible, user-friendly interface while ensuring thorough and accurate vulnerability assessment. Through its dual-component design, the project adeptly addresses the critical need for comprehensive EAR vulnerability detection in today's rapidly evolving digital landscape.

Introduction

In the realm of web application security, identifying and mitigating vulnerabilities is important to protecting sensitive data and maintaining the integrity of online services. In an increasingly online world, where hyperlinks are commonplace, developers must make sure that they provide as safe a browsing experience for their users as possible. In the realm of logic errors, one such vulnerability, known as Execution after Redirection (EAR), poses a significant threat to web applications by granting unauthorised access to protected resources.

The EAR issue can arise in scenarios where execution continues after a redirection command. As mentioned, the issue represents a form of logic error, wherein developers are not aware of the features of the chosen programming language or framework. Some languages, such as Javascript, immediately halt the execution of code after a redirection, whereas vulnerable languages, such as PHP scripts, do not. For instance, consider the following snippet of code:

```
1  <?php
2  if (!$user->is_premium_member())
3  (
4       header("Location: /signup.php");
5  )
6  echo "Premium content that requires a subscription.";
7  ?>
8                    <!-- premium content -->
```

Fig 1. Example of EAR vulnerability in PHP scripts.

The code attempts to redirect non-premium users to the signup page using the **header("Location: /signup.php")** function. However, the script continues to execute after the **header** call, outputting the message "Premium content that requires a subscription.". The echo statement after the **header** call sends content to the browser. According to HTTP standards, headers should be sent before any output from the script [1]. In some server configurations or PHP versions, output before the header call can cause the redirection to fail, and the premium content message is sent to the client, which could be a security issue. In this case, we need to control access and add the Exit/Die statement to follow a **header** redirection, which prevents the server from sending any further output and mitigates the EAR vulnerability.

```
1  <?php
2  if (!$user->is_premium_member())
3  (
4       header("Location: /signup.php");
5       end();
6  )
7  echo "Premium content that requires a subscription.";
8  ?>
9
                    <!-- premium content -->
```

Fig 2. Example of how PHP scripts can be fixed.

The insidious nature of logic errors means that although this is a particularly straightforward solution within PHP scripts, it can be very difficult to detect. Developers may lack detailed knowledge of language details, and since the code in Fig 1. doesn't produce erroneous behaviour, it can easily slip through the cracks.

The project introduces a novel approach to EAR vulnerability detection by combining two powerful tools: a Chrome extension for real-time web page scanning, and a fuzz scanner for

comprehensive subdomain analysis. This combination offers a versatile and effective solution for both quick assessments and in-depth security audits. The Chrome extension allows users to seamlessly scan the web page they are currently viewing. By integrating directly into the browser, the extension provides an immediate assessment of the active page's vulnerability to EAR, highlighting potential security threats as users browse the internet. Complementing the Chrome extension, the Fuzz Scanner tool delves deeper into a domain's infrastructure. It systematically discovers and scans subdomains, uncovering vulnerabilities that might otherwise go unnoticed. This thorough examination is crucial for a complete security assessment, as subdomains often contain overlooked security gaps.

The Chrome extension tool feeds results into a MongoDB [9] database, optimising the scanning process by reusing previous results within seven days. This intelligent caching mechanism significantly reduces redundant scans and speeds up the process, making the EAR Scanner Tools not only thorough in its security analysis but also efficient in its operation. In the following sections, we will explore the context and background of EAR vulnerabilities, the methodology behind the EAR Scanner Tools, their implementation, and an evaluation of their effectiveness in detecting EAR vulnerabilities.

Content and Background

The area of EAR research is very small. For one thing, it is framework-specific, meaning that research into EARs cannot be widely applicable to a wide range of languages. Additionally, as Doupe et al. [2] recognise, EARs are not categorised as a unique vulnerability type. Instead, EARs are spread over a vast variety of categories that share little commonalities, highlighting the misunderstood nature of EARs.

The current landscape of EAR research is primarily focused on detection methods. Doupe et al. [2] are a seminal paper on EARs and proposed a white-box detection method for Ruby on Rails application. They aimed to find a path through a Control Flow Graph (CFG) that contains a redirect and then reachable code afterwards. After pruning infeasible routes, authors categorised EAR vulnerabilities based on whether there is a path from subsequent code to modify the database. In particular, their heuristic broadly detected a list of 16 modifying functions. Using this method, they were able to find that of the then 20,000 analysed open-source projects on Github, about 6% contained any EAR at all. Of that 6%, 21% contained vulnerable EARs, meaning that they led to database modifications that could be exploited.

Although this paper is influential, the decision of heuristic led to a large number of false vulnerable EARs, due to an incorrect examination of the CFG. Additionally, the heuristic did not consider the base type of the object the functions were called on, meaning that **delete** functions for maps would be falsely flagged as vulnerable EARS, although it would not modify any database. Additionally, this paper is incredibly old and many of the techniques and frameworks utilised are not viable today.

A follow-up paper by Chai et al. [3] sought to improve the heuristic methodologies of Doupe et al. and extend the algorithm to web applications written in PHP. The authors propose a path-sensitive interprocedural analysis to enhance the previously proposed detection model method. Their approach involves constructing a control-flow graph, conducting static analysis based on the graph, and detecting EAR vulnerabilities by considering path conditions. To mitigate false positives, they introduce a set of critical operations and employ an SMT solver to

validate path conditions. The paper evaluates the approach using snippets from the CVE database, demonstrating improvements in detecting certain patterns that the original method missed. However, the authors acknowledge limitations in handling dynamic features and suggest avenues for future work, emphasising the need for more automated means and addressing non-string operations in web applications.

Another follow-up paper to A Doupe et al. takes a drastically different approach. Payet et al. [4] use a black box method of detecting EAR vulnerabilities. The paper uses the HTTP redirect response as an indication of whether or not an EAR is present. There is a preliminary classification of the redirect types (whether there is an information leakage, garbage content, or generic redirect). In addition, the researchers provide a robust classification system of EARs based on the HTTP responses. This system identifies four categories of EAR responses, each with its characteristics. Some example information leakages are pre-login access (accessing restricted content) and error messages which reveal the particular framework of that application, which can be a security risk. Black box detection methods in general can only find explicit EARs rather than potential EARs. In addition, their categorisation system is prone to false positives and negatives, particularly for sites not in English.

Methodology

The EAR Scanner Tools project leverages a multi-faceted methodology to provide a robust and user-friendly system for detecting Execution After Redirection (EAR) vulnerabilities. The approach encompasses the adaptation of an EAR scanning algorithm into a web application, the use of a MongoDB database for storing scan results, the creation of a Chrome browser extension for user-initiated scans, and the deployment of a fuzz scanning tool for automated subdomain analysis.

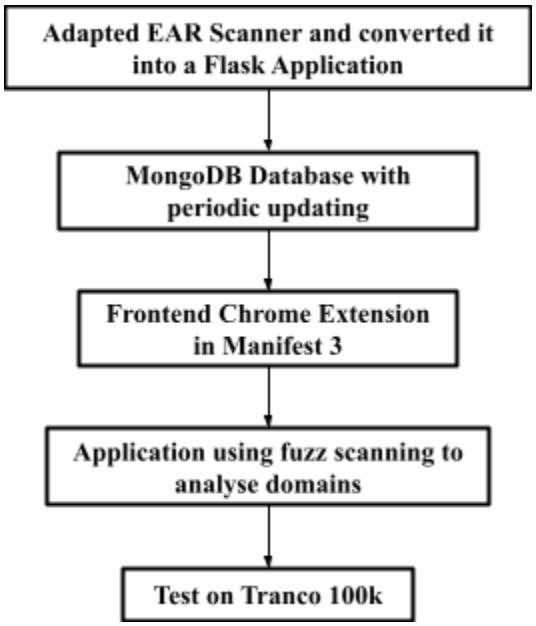


Fig 3. The overall methodology of this project.

The Core EAR Scanning Algorithm

At the heart of the EAR Scanner Tools project is the core EAR scanning algorithm, encapsulated within a Python-based EAR_Scanner class. For this project, we used the EARScanner Repository on GitHub, adapting it as needed to suit our needs [5]. It is designed to be flexible and robust, capable of being run in both a manual, single-URL mode and an automated, multi-URL fuzzing mode. The scanner can accept command-line arguments for various configurations such as target URL (-u), wordlist for fuzzing (-w), timeout settings (-t), number of threads (-th), and expected content length (-c).

Option	Usage	Option	Usage
-u,--url **(Target URL)**	• Used to specify a single URL that the EAR Scanner tool will scan for EAR vulnerabilities. • Example usage: www.google.com	**-t,** **--timeout** **(Timeout Settings)** **Default: 60s**	• Sets the timeout for HTTP requests in seconds. This helps to avoid hanging the scanning process on unresponsive URLs. • Example usage: 30/60/120 seconds
-w, --wordlist **(Wordlist for Fuzzing)**	• Accompanies the --fuzz-scan parameter to provide a wordlist file that *gobuster* or similar tools will use for fuzzing. • If not specified, it defaults to "content_discovery_all.txt". The "content_discovery_all.txt" file contains a comprehensive list of commonly used subdomain patterns, which can be utilized with tools like *gobuster* to brute-force DNS subdomains and virtual hostnames on targeted web servers.	**-th,** **--thread** **(Number of Threads)** **Default: 100**	• Determines the number of parallel HTTP requests that can be made by the scanner. Increasing the number of threads can speed up the scanning process but might also increase the load on the server and network. • Example usage: 50/100/150
		-c, **--content-length** **(Expected Content Length):** **Default: 200 bytes**	• Used to specify a threshold content length for EAR vulnerability confirmation. If the content length of the response from a redirected URL is greater than or equal to this value, the URL is considered likely vulnerable. • Example usage: 50/100/150/200/250/300

Fig 3. Description of the different options employed by the EAR Scanner tool.

The algorithm employs Python's concurrent.futures module to execute multiple scan tasks concurrently. This approach significantly speeds up the scanning process, particularly during

fuzz scanning when a large number of subdomains may need to be checked for vulnerabilities. Upon receiving the HTTP response, the algorithm checks for a 302 Found status code, which indicates that the requested resource has been temporarily moved to a different URI provided by the Location header. If the response's content length is greater than the specified content_length, it is flagged as potentially vulnerable. This is based on the idea that a vulnerable page may provide a substantial response body even when redirecting. Results are categorised based on their vulnerability status and can be written into an output file. Vulnerable URLs are prefixed with [VULNERABLE], while non-vulnerable URLs are prefixed with [OK]. Any errors encountered during the scan are recorded with an [ERROR] prefix.

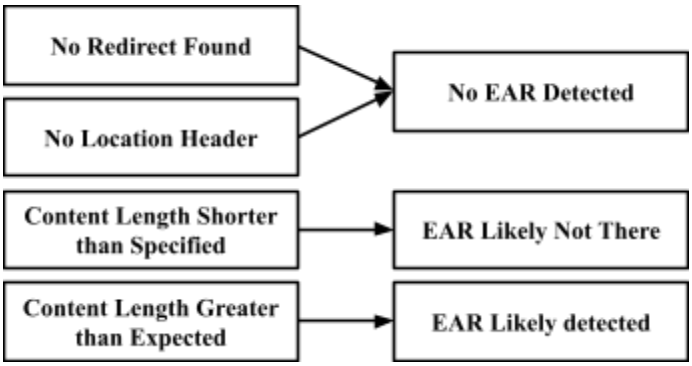


Fig 4. Graph explaining logic steps for EAR detection

Adapted EAR Scanner into Flask Application

The core EAR scanning repository was adapted into a Flask application to provide a backend service. This service is responsible for handling scan requests, executing the EAR scanning logic, and returning results. The Flask framework was chosen for its simplicity and flexibility, allowing for quick development cycles and easy integration with other components of the project. We deploy the Flask backend for the single URL scan on Render.com, as it requires low overhead and users can directly use the Chrome extension without the need for any setup.

MongoDB Database with Periodic Updating

A MongoDB database was integrated into the system to store scan results. This ensures that repeated scans within a seven-day window can be served from cached data, significantly reducing the scanning time and server load. The database schema is designed to store URLs, scan results, timestamps, and vulnerability status. Periodic updates ensure that the database reflects the most current state of scanned URLs.

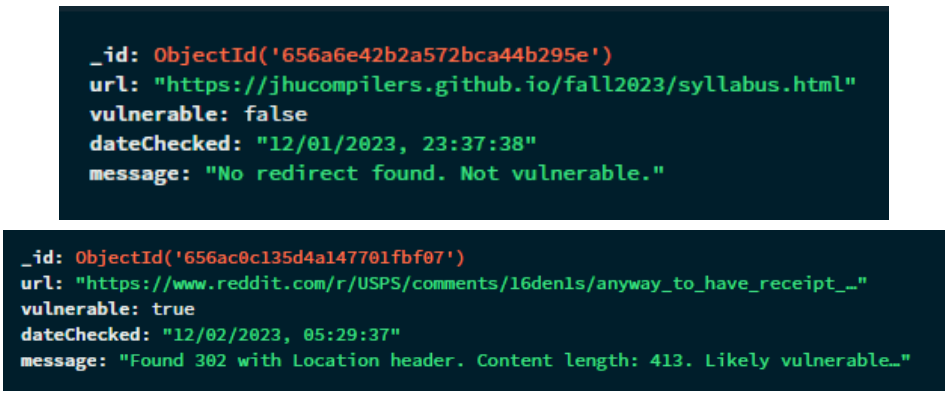


Fig 5 & 6: MongoDB Atlas collection dashboard. Top: Examples of non-vulnerable URLS. Bottom: Examples of vulnerable URLS.

Frontend Chrome Extension in Manifest 3

A Chrome extension serves as the frontend interface for the EAR Scanner. It utilises Chrome's Manifest V3 to ensure up-to-date compliance with browser extension security and performance standards. The extension scans the user's currently active web page, streamlining the process of checking for EAR vulnerabilities. The specifics of its usage will be explained later.

Application Using FuzzScan to Analyze Domains

The FuzzScan tool is an application that automates the process of subdomain discovery using tools such as *gobuster*. *Gobuster* can perform brute-force attacks on URIs, attempting to discover hidden directories and files within websites. Also, it supports DNS subdomain enumeration, including handling wildcard subdomains, identifying virtual hostnames on web servers and searching TFTP (Trivial File Transfer Protocol) servers, which greatly expands the surface area and uncovering potentially unlisted or forgotten pages for EAR vulnerability scanning. Our tool scans each discovered subdomain for EAR vulnerabilities, compiling results that are then processed by the backend Flask application. This automated process allows for extensive coverage of a domain's subdomain structure, providing a comprehensive security assessment.

Tested on Tranco 100k

To evaluate the efficacy and performance of the EAR Scanner Tools, the system was tested against the Tranco list of the top 100k websites. This benchmarking ensures that the tools are capable of operating at a large scale and can handle the diverse range of web application architectures and configurations found on popular sites.

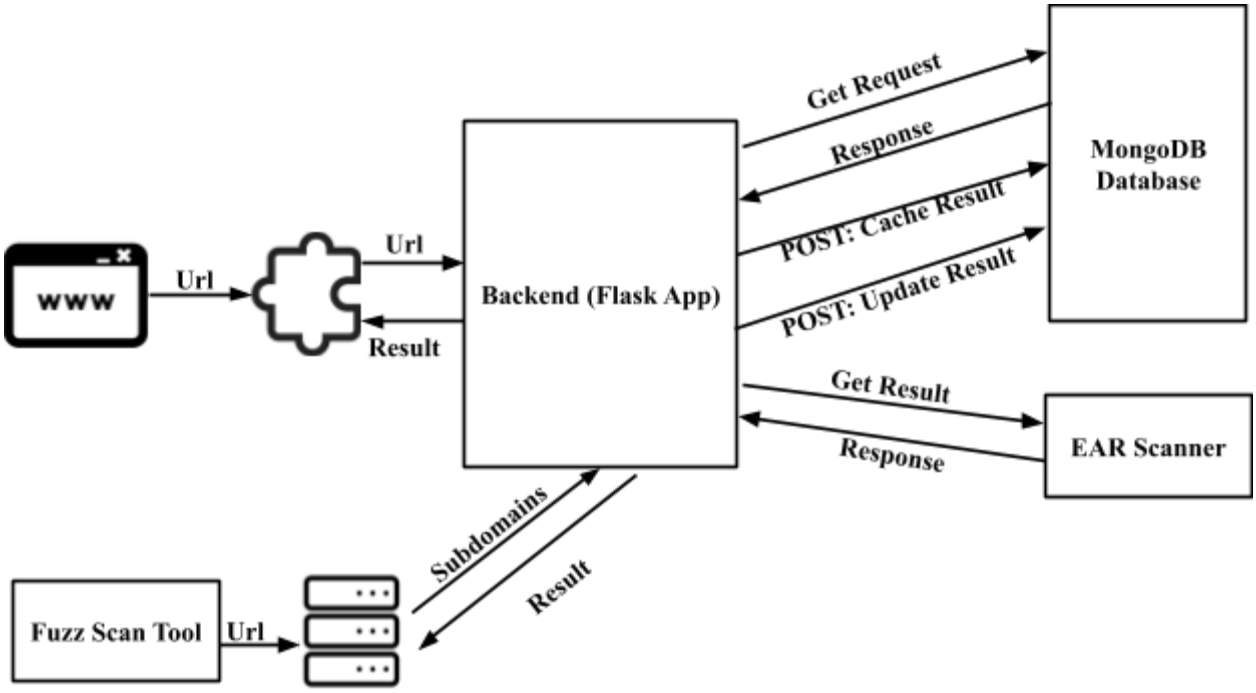


Fig. 7: Diagram describing the overall structure of the project.

Implementation

As mentioned previously, a previously existing EAR scanner tool was utilised as a basis for the development of the underlying scanning tool [5]. However, it was built upon to increase its usefulness and applicability to this research project.

Flask Backend Application

By adding the backend application, the EAR scanner was able to be integrated into the Chrome extension effectively and without issue. The initial EAR scanner was written in Python so for ease of integration, using a Python-based framework for the backend was ideal. Popular backend frameworks based on Python include Flask and Django, both of which were considered for the project. Due to time considerations and previous familiarity with Flask, it was chosen to be the framework for the project. Flask is a framework that doesn't require tools or libraries and is relatively basic when considering things like database abstraction layers or form validation [6]. Though it was known that database functionality would be added, familiarity with Flask outweighed the potential ease of database integration that could be found with Django [7].

Database Storage

By adding database storage, fewer scans were necessary as a check in the database for an identical URL would be sufficient if a previous lookup was done in the previous 7 days. A NoSQL database fit the needs of the project, though options such as Postgres (relational) databases were considered as well. MongoDB Atlas was used for hosting the database, due to its reputation and popularity. Database queries were done using the database's associated Data API and the Python request library.

Fig 8: Top - Chrome extension after EAR detected; Middle - Chrome extension EAr is potentially found; Bottom - Chrome extension on safe site.

Chrome Extension

Additionally, adding the Chrome extension as a frontend allows for more effective and efficient usage by end users, enabling them to have more awareness of potential security risks as they are browsing the internet. Manifest V3 was used for this project, in conjunction with JavaScript, HTML, and CSS, to create the Chrome extension. After users load the extension onto their web browser, they can navigate to any site. The extension will communicate with the backend, and display a small badge on the extension icon. There are two potential outputs: " 👂 ", indicating an EAR and "✅" indicating no definitive EAR was found. If the user clicks on the extension, they will be greeted with a funny image and some information about the scanner result. The extension will also display the most recent time the website was scanned. The overall response time is less than 1 second, which though fast, on occasion can feel clunky.

As we can see in Fig 8, the pop-up displays more information about the result. For instance, the middle image and the right-hand image both are marked with ✅, but the middle image specifies that there is a potential EAR on the page, it just is not likely. We could have offered a third badge option, but there is a good chance of a false positive with the second case, and so it isn't necessary.

Fuzz Scanner

The Fuzz Scanner tool is a crucial component of the EAR Scanner Tools suite, designed to conduct exhaustive subdomain discovery for a given domain. This tool incorporates *gobuster*, a popular utility famed for its efficiency in brute-forcing URIs and DNS subdomains [8]. The Fuzz Scanner initiates by enumerating all possible subdomains of the target domain using *gobuster*. This process systematically generates and tests a list of potential subdomains against the domain's DNS server, retrieving any that are valid. For each discovered subdomain, the tool subsequently performs an EAR detection check. As a practical example, we applied the Fuzz Scanner to the academic domain “jhu.edu”. The tool methodically scanned a myriad of subdomains, as illustrated in the provided screenshot. Despite encountering numerous HTTP 301

status codes—indicating redirection or that the subdomains may no longer exist—it thoroughly checked each one. This comprehensive approach ensures no potential EAR issue is overlooked. The Fuzz Scanner provides a thorough assessment of a domain's subdomains, a task that would be prohibitively time-consuming and complex if performed manually. Therefore, we advise against using this tool on very large domains, such as google.com. It is particularly effective for small to medium-sized websites, like personal sites. Also, given its extensive scanning process, the Fuzz Scanner is deliberately separated from the EAR Scanner Chrome extension, which ensures that the Chrome extension remains lightweight and user-friendly.

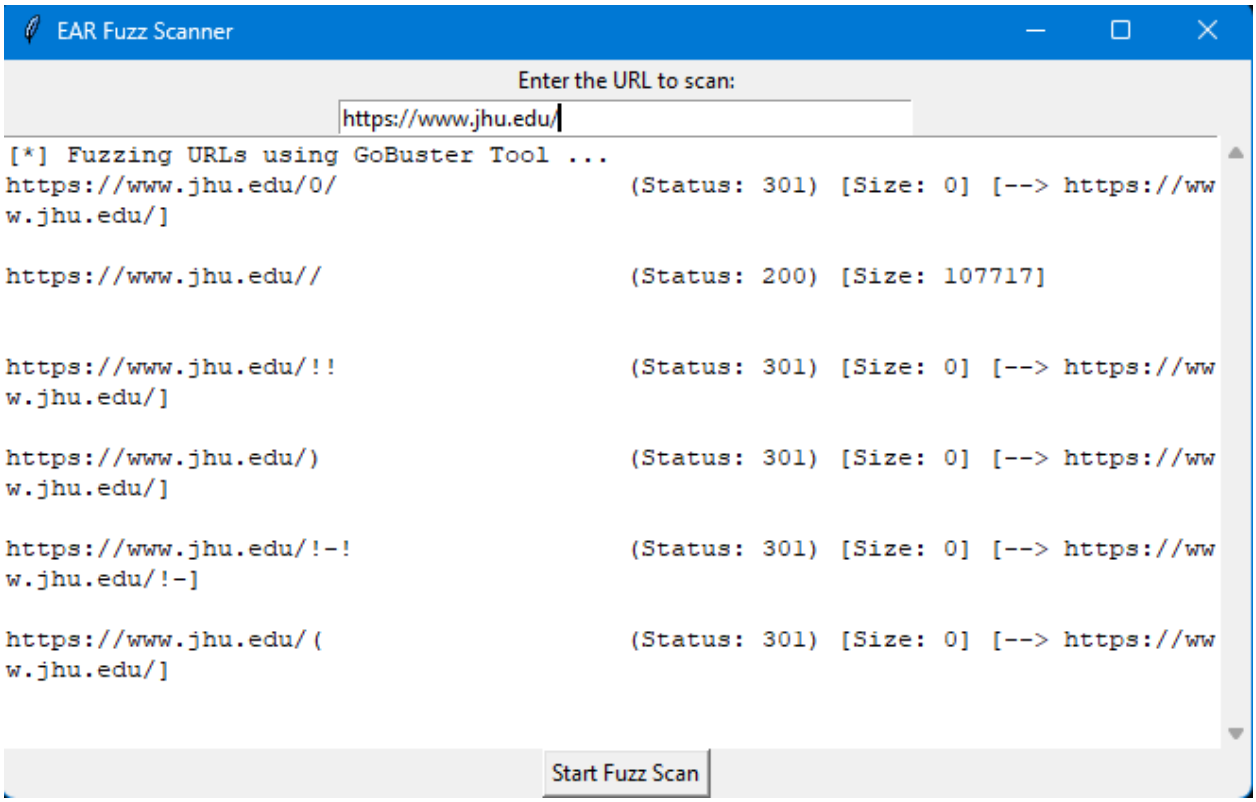


Fig 9: Fuzz Scan Tool

Results

The single-url scanner tool (not Fuzz Scanner) was used to test a multitude of popular websites, to better understand the existence and severity of EARs across the internet. The Tranco list, a list of the top websites, was used as input. Tests were run on a variety of dataset sizes to best understand the limits of the program and to accommodate time constraints. Starting from running the scanner across 100 sites, tests were run on every factor of 10 between 100 and 100K sites (i.e., tests were run at 100, 1000, 10K, and 100K sites). At 100K, the tests took approximately 2 hours to run, so we opted out of analysing further. Within each category, approximately 1% were vulnerable, though at the lowest sample size of 100, a 2% rate of vulnerability was found.

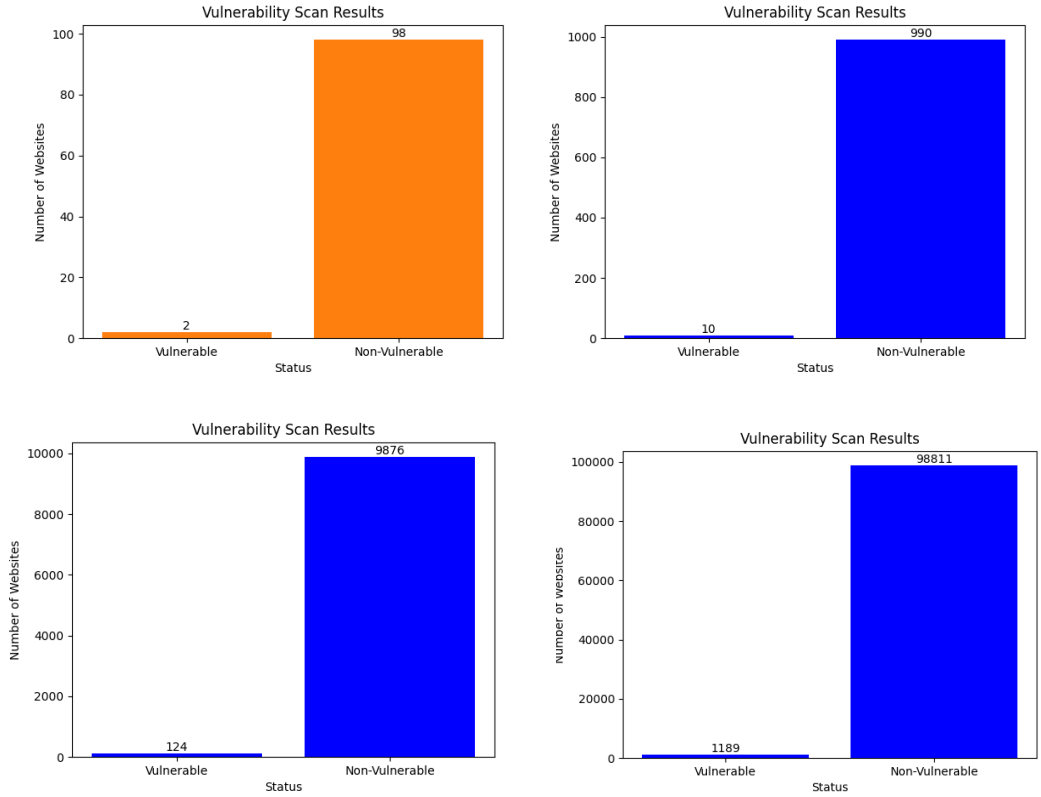


Fig 10. Final results against 100 sites (top left), 1,000 sites (top right), 10,000 sites (bottom left), and 100,000 sites (bottom right).

Additionally, the impact of changing content length within the EAR scanner was tested against the percentage of EARs found by the application. This was done to consider potential changes in the content length threshold throughout the process and in future improvements. It was found that there was a fairly steep drop-off between a content length of 100 and 150 (-0.91% EAR's found), and again between 200 and 250 (-0.75% EAR's found). This makes a lot of sense. In general, with 50 and 100 content lengths, we've lowered the threshold of detectable EARs, allowing some content that otherwise would not have been considered, potentially increasing the rate of false positives. On the other hand, the two longest content lengths may be *overly* restrictive, as there are potential EARs from error messages exposing framework information that may be too short in these scenarios. Interestingly, it seems that 50-100, 150-200, and 250-300 share similar EARs found, but adjacent groups have a larger difference. This may be due to different message types generally having similar lengths, but this was not investigated further.

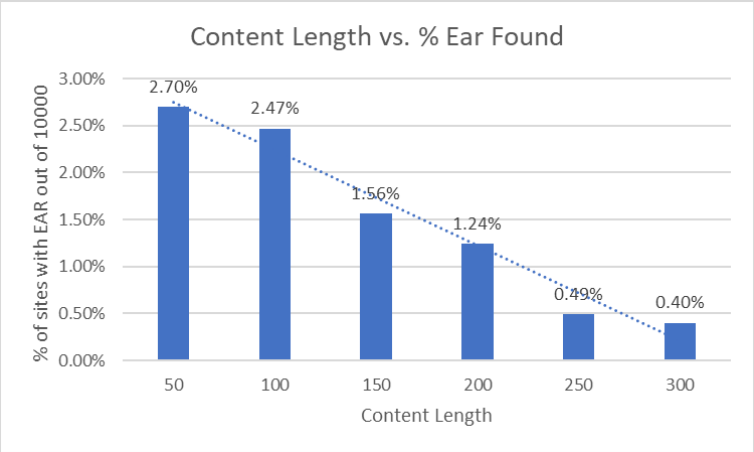
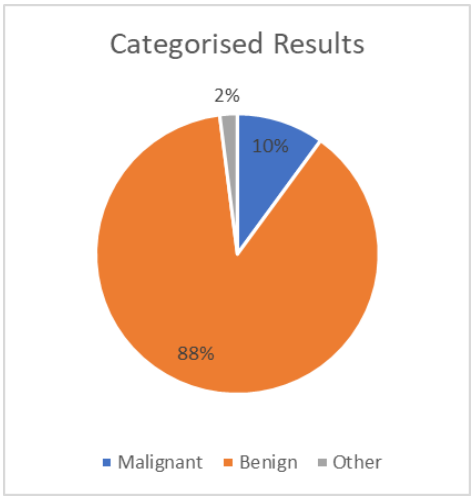


Fig 11. Comparison of content length and the effectiveness of finding EARs.

To better understand the impact of the EAR vulnerabilities found, a further investigation of content responses was necessary. As such, 50 of the top vulnerable websites were further investigated by inspecting the content in the response. Three categories were decided on: benign, malignant, and other. Benign responses included responses that were determined to pose no threat or to be a benefit to the user, such as a page informing the user that a website page was relocated. Malignant responses included responses that were determined to pose some threat either to the user or to the website provider. This could include pages that exposed underlying frameworks or vulnerabilities of the website, which are what were found in malignant websites that were inspected. Additionally, other responses included responses that returned content that wasn't meaningful, such as a page with a header and valid tags, but were otherwise empty.

Approximately 10% of responses were found to be malignant. Notably, 88% of flagged sites were determined to be benign, and further, 90% were determined to be benign or other which would encapsulate a category of 'non-threat', or falsely positive responses.



Category	Number of Results
Malignant	5
Benign	44
Other	1
Total	50

Figure 12 & 13. Categorisation of 50 vulnerable sites.

Limitations

While the EAR scanner tool is a more integrated way of allowing users to test whether websites they are visiting are vulnerable, there are a variety of limitations to the approach taken both in the extension and in the fuzz scanner.

HTTP Response Codes

In general, several response codes are used for redirects. Our project only utilised status code 302 which although ubiquitous with redirect requests, by constraining ourselves to just this code, we leave out many other potential instances of EARs. Further research can consider the impact of different status codes, and whether that would increase the rate of EAR discovery.

Fuzz Scanner

The Fuzz Scanner consumes a lot of resources and time when running over medium or large domains. It is implemented with *gobuster*, a tool to aid in the discovery of domains and subdomains, but which uses brute force to find each of the potential pages. As such, using it on medium or large domains necessitates more time than is efficient. For this reason, it was not integrated into the Chrome extension (to do so would make the Chrome extension lag and storing a large amount of data in the database would make future queries more inefficient as well). Currently, the use case for this remains a small website that is being investigated by its developer to see whether any vulnerabilities exist in its program.

False Positives in Results

As was indicated earlier, there was a high rate of false positives within our data. More precisely, many of the websites that were flagged as potentially vulnerable had benign responses (as

manually classified). This wasn't something that was controlled for in the current application but would be a necessary future expansion to increase accuracy. Implementation of this could involve NLP analysis of responses — as one of the tests, some results were manually investigated to determine the level of severity or vulnerability, but having something like that integrated into the application would perhaps help with controlling the rate of false positives. As mentioned earlier, Payet et al. [4], utilised a robust filtering method that they automated using different methods such as regular expressions and Normalised Compression Distance (NCD). These methods still require some upfront manual analysis first, but their categorisation, in general, is more vigorous. Integrating this may make response time slower for new sites, but in general, will produce more accurate results.

Database Access/Storage Improvements

With an increased volume of URLs stored in the database, retrieval may slow down. As such, future improvements may include potential storage efficiency improvements, such as potentially storing items with a high hit rate in a separate table with a relatively small limit, to decrease the response rate for popular items.

Future Work

Future contributions or improvements to this project could include any number of additional methodologies. Implementing a more efficient domain and subdomain discovery tool to use in the Fuzz Scanner could enable more use cases for that tool. Implementing NLP analysis of response text, to determine the degree of vulnerability, could lead to reduced levels of false positives when using the scanner and thus the extension. Additionally, integrating some level of static analysis could be an effective tool for targeting popular frameworks more efficiently. Currently, the tool uses dynamic analysis to be language agnostic and thus more applicable over a broad variety of websites. However, static analysis could lead to more accurate results as they would directly be checking the code for problematic patterns, but at the same time is language-specific. Although there is a trade-off, integrating the two approaches could lead to a tool with improved accuracy.

Conclusion

We have shown that despite EAR's obscurity, they are incredibly widespread. Our suite of tools is robust enough to find and detect EARs, and simple and pleasant to use for the layman. We've shown that the solutions are not difficult, but they require a knowledge of frameworks developers can sometimes lack, owing to assumptions and inexperience. As more of our content experiences shift to be online and hyperlinks become more ubiquitous, all developers need to improve their understanding of this exploitable flaw.

References

[1]"PHP: header - Manual," *Php.net*, 2023. https://www.php.net/manual/en/function.header.php#:~:text=Remember%20that%20header()%20must,before%20header()%20is%20called (accessed Dec. 12, 2023).

[2] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, “Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities,” 2011. Accessed: Dec. 12, 2023. [Online]. Available: https://sites.cs.ucsb.edu/~chris/research/doc/ccs11_ear.pdf

[3] C. Chai, X.-B. Yan, Q. Wang, and S. Liu, “Static detection of execution after redirect vulnerabilities in PHP applications,” *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pp. 489–492, Aug. 2016, doi: https://doi.org/10.1109/icsess.2016.7883115.

[4] P. Payet, A. Doupé, C. Kruegel, and G. Vigna, “EARs in the wild,” *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pp. 1792–1799, Mar. 2013, doi: https://doi.org/10.1145/2480362.2480699.

[5] “PushpenderIndia/EARScanner: Execution After Redirect (EAR) / Long Response Redirection Vulnerability Scanner written in python3, It Fuzzes All URLs of target website & then scan them for EAR,” *GitHub*, 2023. https://github.com/PushpenderIndia/EARScanner/tree/main (accessed Dec. 12, 2023).

[6] “Design Decisions in Flask — Flask Documentation (3.0.x),” *Palletsprojects.com*, 2023. https://flask.palletsprojects.com/en/3.0.x/design/#what-does-micro-mean (accessed Dec. 12, 2023).

[7] “Django,” *Django Project*, 2023. https://www.djangoproject.com/ (accessed Dec. 12, 2023).

[8] “OJ/gobuster: Directory/File, DNS and VHost busting tool written in Go,” *GitHub*, Aug. 14, 2023. https://github.com/OJ/gobuster (accessed Dec. 12, 2023).

[9]“MongoDB Atlas Database | Multi-Cloud Database Service,” *MongoDB*, 2023. https://www.mongodb.com/atlas/database (accessed Dec. 12, 2023).