Role Models: Role-based Debloating for Web Applications

Babak Amin Azad
baminazad@cs.stonybrook.edu
Stony Brook University

Nick Nikiforakis
nick@cs.stonybrook.edu
Stony Brook University

ABSTRACT

The process of debloating, i.e., removing unnecessary code and features in software, has become an attractive proposition to managing the ever-expanding attack surface of ever-growing modern applications. Researchers have shown that debloating produces significant security improvements in a variety of application domains including operating systems, libraries, compiled software, and, more recently, web applications. Even though the client/server nature of web applications allows the same backend to serve thousands of users with diverse needs, web applications have been approached monolithically by existing debloating approaches. That is, a feature can be debloated only if none of the users of a web application requires it. Similarly, everyone gets access to the same "global" features, whether they need them or not.

Recognizing that different users need access to different features, in this paper we propose role-based debloating for web applications. In this approach, we focus on clustering users with similar usage behavior together and providing them with a custom debloated application that is tailored to their needs. Through a user study with 60 experienced web developers and administrators, we first establish that different users indeed use web applications differently. This data is then used by *DBLTR*, an automated pipeline for providing tailored debloating based on a user's true requirements. Next to debloating web applications, *DBLTR* includes a transparent content-delivery mechanism that routes authenticated users to their debloated copies. We demonstrate that for different web applications, *DBLTR* can be 30-80% more effective than the state-of-the-art in debloating in removing critical vulnerabilities.

KEYWORDS

Software Debloating, Web Applications, Attack Surface Reduction

ACM Reference Format:
Babak Amin Azad and Nick Nikiforakis. 2023. Role Models: Role-based Debloating for Web Applications. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy (CODASPY '23), April 24–26, 2023, Charlotte, NC, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3577923.3583647

1 INTRODUCTION

Since the introduction of the World Wide Web, there has been an exponential growth in the number of online websites and web services [38]. Our lives are entangled with online services, and companies and governments are hosting their vital infrastructure online. Both the quality and the complexity of these services have increased drastically over the past decade. As the need for more complex online services rose, the development community matured and started building more and more reusable pieces of code.

The industry standard in web-development practices switched from writing in-house code from "scratch" to the use of professionally developed and maintained third-party modules [25, 27, 30]. Modern web applications commonly incorporate frameworks and packages from public sources to provide routine features such as page management, user authentication, error handling, testing, and logging [28]. Inherent to the use of off-the-shelf software, is the resulting amalgam of useful and non-useful features. Packages provide a variety of features (e.g., support for multiple database backends) to be useful to as many projects as possible. At the same time, this added flexibility comes at the price of code bloat. Code bloat refers to parts of the application source code that serve no purpose to its users. In the example of database APIs, if the website only interacts with a MySQL database, the source code for other database APIs still remains in the application. While this may seem benign, flaws in the unused parts of applications can lead to security vulnerabilities [2, 5].

One line of research called *software debloating* focuses on identifying and neutralizing unused parts of applications. Existing systems perform debloating either directly at the source-code level by rewriting the code to remove/block unused code paths [2, 18], or alternatively limit the underlying APIs available to each page to reduce the impact of exploits [5]. Unlike binaries, when dealing with web-application vulnerabilities, attackers can inject data and execute code, but *cannot* jump to arbitrary instructions within the code. Therefore, removing dead code is only of limited benefit for web applications. As a result, web-application debloating mechanisms commonly remove live code that is deemed as unnecessary based on dynamic code-coverage traces and static analysis.

One of the main limitations of prior work is the focus on one-size-fits-all debloating. In such schemes, all users of the web application regardless of their role and access type, receive the same treatment. In other words, existing debloating systems produce a *single* copy of a debloated application to serve all public and authenticated users. One may be hopeful that the authentication and access control modules within the web applications would only provide access to critical features for those users who need them, yet this assumption is critically flawed. First, not all popular web applications provide fine-grained access control (e.g., phpMyAdmin), and for those that do provide it (e.g., WordPress), the predefined list of available roles may not match the behavioral patterns of users, leading to over-authorization.

Furthermore, access-control flaws where users have access to features that they should not have access to, are commonly found

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CODASPY'23, April 24-26, 2023, Charlotte, NC, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0067-5/23/04…$15.00
https://doi.org/10.1145/3577923.3583647

even in popular platforms [7, 8]. Among other examples, attackers have been able to directly invoke privileged vulnerable modules in WordPress, which allowed them to fully bypass the authentication and authorization of the main application [16].

To address the limitations of prior web application debloating systems we propose *DBLTR*, an automatic role-based debloating pipeline that identifies clusters of similar usage patterns among users which can be considered equivalent to a dynamically generated access-control role. After identifying the optimal number of roles (*N*), *DBLTR* creates *N* debloated copies of the web application tailored to the true needs of each subset of users. *DBLTR* orchestrates the access to these applications through the use of a transparent reverse proxy that captures the successful authentication requests and subsequent authentication cookies to route known users to their custom debloated web applications. This process is done without the need to modify target web applications beyond the debloating process, and requests are routed transparently from the perspective of users.

DBLTR yields multiple concrete advantages compared to prior schemes of debloating web applications. First, it creates a separation between public and authenticated users which protects web applications even in the face of access-control errors. Second, it limits the damage that is possible by a given authenticated user (e.g., due to compromised credentials or client-side attacks) by limiting the attack to the parts of the web application that the user requires for their tasks. Third, the clustering of users into sets that access differently-debloated web applications, provides a fine-grained access control mechanism, which operates on top of any existing access-control mechanism and can capture the real needs of users on a feature-by-feature basis. For instance, a WordPress administrator that only publishes blog posts and replies to comments will receive access to a tailored WordPress application where critical features (e.g., theme modification and plugin installation) are neutralized for that user, yet remain available to other privileged users in the same deployment who rely on them. Lastly, due to its modular nature, *DBLTR* can integrate with any future static/dynamic end-to-end web application debloating scheme to improve their debloating results.

To better understand how the various components of *DBLTR* work in unison to differentially-debloat and secure web applications, we have prepared the following demonstration. We show a scenario involving a CSRF exploit on phpMyAdmin (CVE-2019-12616) and two users who both visit the same exploit-launching page prepared by an attacker. The video of our demonstration is available online at: https://dbltr.debloating.com/.
Overall, we make the following original contributions:

- To back our intuition that different privileged users utilize web applications in different ways, we perform a user study including 60 participants to understand how experienced developers and administrators interact with popular web applications.
- We propose *DBLTR*, an automated web-application debloating and content-delivery system which is capable of reducing the size of applications by more than 70% and removing as much as 80% of severe security vulnerabilities beyond the state-of-the-art in web application debloating.
- We analyze the security gains and quantify the attack-surface reduction of our debloating scheme based on various source code (e.g., line reduction) and security metrics (e.g., CVE reduction, Critical API removal, etc.)

To motivate additional research in the area of software debloating, and to ensure reproducibility of our findings, we will be releasing *all* of our software artifacts upon publication of this paper.

2 BACKGROUND

In this section, we review the topics that will be used as building blocks for the remainder of the paper. First, we discuss the details of debloating web applications based on dynamic code-coverage traces. Then, we review several source-code metrics that can be used to measure the effectiveness of debloating from the perspective of attack-surface reduction.

2.1 Debloating based on dynamic usage traces

Debloating based on dynamic traces for web applications was first introduced by the "Less is More" work of Amin Azad et al. [2]. The authors incorporated a set of automation tools and scripts to *simulate* user behavior while recording the executed server-side files as well as their respective lines.

The authors discussed two debloating strategies, file-level and function-level debloating. File debloating only removes the whole AST of a file, if none if its underlying statements are ever exercised based on the dynamic usage traces. Conversely, function debloating analysis is more fine grained and can remove sub-graphs of the AST if it identifies a function with no code-coverage, and therefore, can produce smaller applications with fewer vulnerabilities.

The "Less is More" system incorporates the XDebug PHP module that allows it to record the list of executed PHP files and lines. In this work, we follow a similar method to record code-coverage, and extend the debloating ideas of Amin Azad et al. by analyzing *real* user data, and identifying clusters of users that perform similar actions. We use these clusters to demonstrate the debloating improvements made possible by role-based debloating.

2.2 Debloating metrics

Debloating by definition removes or neutralizes parts of the application that users do not require. In order to demonstrate the security gains by removing a piece of code, previous work has used several source-code metrics.

Size reduction: measures how much code was removed through the debloating process. McConnell discussed in his work that the size of the code positively correlates with the number of software bugs it contains [20]. The reduction in Logical Lines of Code (LLOC) measures the size reduction by counting the number of statements in an application pre- and post-debloating and is resilient to changes in the syntax and coding style.

Reduction of security vulnerabilities: is another metric that focuses on historic CVEs. By mapping public CVEs to the source code of web applications, we can identify whether the vulnerable piece of code is removed by debloating. This is a powerful metric as it focuses on real and mostly exploitable vulnerabilities as opposed to proxy variables (such as LLOC) that may or may not result in real vulnerabilities. In terms of downsides, next to the effort required to map vulnerabilities to source code, CVE-reduction is only meaningful in hindsight since it can be used to understand how a debloating system would have performed if an application was debloated *before* the now-known vulnerabilities were discovered.

Critical API Calls (CAC) reduction: PHP applications interact with their environment through the APIs provided by the PHP engine, which are also known as built-in functions. These functions expose low-level C API implementations and provide a variety of functionality to perform network, database, and file-system operations. Similarly, PHP extensions which are also written in C, expose their functionality through defining new APIs.

Protecting CACs has received the attention of binary debloating and exploit prevention research in the past. Namely, Shredder [21], ROPGuard [11], and kBouncer [29] have emphasized the importance of protecting Critical APIs. In the realm of web applications, the literature on taint analysis for vulnerability detection commonly incorporates the list of such critical APIs that attackers can use to execute various types of attacks. For instance, attackers can abuse APIs exposed through the MySQLi PHP extension to mount SQL injection attacks. Similarly, the exploitation of file-system APIs can lead to arbitrary file-write attacks. Therefore, reducing the access of attackers to such functions through debloating provides tangible security benefits. The RIPS tool by Dahse et al. provided a comprehensive list of Critical APIs, which were treated as sinks for PHP taint analysis [6]. We incorporate the 205 sinks from RIPS and measure the removal of such critical API calls (CACs) from the debloated web applications.

We categorize the sensitive sinks from RIPS into four main groups. **Code Execution** APIs are those which can be used to directly, or indirectly run code or change the control flow of the application (e.g., call an arbitrary function). Next on this list are **File System APIs** which enable interactions with the file system, such as, deletion and file manipulation and can be abused to take over an application by overwriting files, overriding credentials, and removing sensitive configuration files. **Information Disclosure** functions can be abused by attackers to expose sensitive information from the web application or its host operating system. Lastly, for APIs that do not clearly belong to one of the aforementioned categories, we list them under **Other**. This group of critical APIs may allow attackers to conduct malicious actions, such as, sending spam emails, or evading authentication by changing environment variables.

PHP Object Injection Gadgets: Object injection is a vulnerability where user-controlled values reach an `unserialize` API without proper sanitization. In such cases, attackers can assemble a list of PHP classes that already exist in an application and mount an exploit which commonly leads to arbitrary file writes and even remote code execution.

In the case of object-injection attacks, the attacker can abuse the code in the existing classes in their target web applications. Since attackers cannot divert the control flow directly by calling arbitrary functions from the injected classes, they rely on specific functions (e.g., class destructors) to piece-together exploit payloads which are called "gadgets."

In this paper, we incorporate the list of publicly reported gadgets by PHPGGC [35]. This tool includes a repository of gadget chains within popular third-party packages. Therefore, if a vulnerable application makes use of any of these packages, attackers can inject one of the gadget chains from PHPGGC to gain RCE (Remote Code Execution), write to arbitrary files or interact with the database. PHPGGC does not provide a comprehensive list of gadgets for all packages and all classes in our target web applications. Nevertheless, this approach which is also incorporated in the literature [2] provides us with a quantitative measure for the reduction in gadget chains after debloating.

For each of the web applications in our dataset and their third-party packages, we check for the existence of gadget chains based on the PHPGGC dataset. We then check whether debloating removes these gadgets. For debloated instances where we have removed the gadgets, even if attackers find an object injection vulnerability, they cannot abuse any of the public gadget chains in their exploits.

3 USER-STUDY

To empirically evaluate the behavior of users of popular web applications and understand the patterns of their interactions, we conduct a user study. The main goal of this user study is to understand whether administrators use different subsets of the overall features available to them and would therefore benefit from differentially-debloated web applications. Moreover, through this study, we analyze which features in the web applications in our dataset are commonly used among developers and administrators, and which features are relatively unpopular, used by only a fraction of administrators.

We hire the user-study participants by advertising paid projects on popular freelancing platforms, such as, Upwork and Fiverr [10, 39]. We interview freelancers with 2-10 years of expertise on web development and system administration. We specifically interviewed candidates who mentioned phpMyAdmin, WordPress, or Magento on their resume. We focused on these web applications since they were used in the "Less is More" study of Amin Azad et al. [2], allowing us to compare and contrast our findings with theirs.

3.1 User-Study Deliverables

The task description for each project consists of an overview of the user-study, the background required to participate, and the expected deliverables. Moreover, we include the information about the consent to participate in our study and describe the information that we collect (i.e., server-side logs and code-coverage information).

After interviewing the participants and reviewing their resumes, we hire 20 experts for each web application for a total of 60 experts on phpMyAdmin, WordPress, and Magento. We compensate the participants at the rate of $15 per hour. During the pilot experiments, we realized that not every freelancer is familiar with the concept of a user study. More importantly, to avoid future disputes, freelancers preferred to work on a predefined list of deliverables.

Based on these observations, we define two milestones for our user study. First, we ask our participants to provide a list of web application features that they commonly use in their daily tasks and projects. Most of our participants listed both maintenance and administration tasks. Among the common tasks, we observed

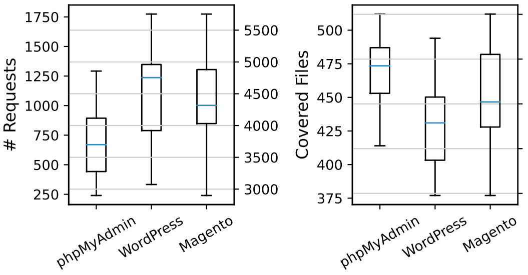

Figure 1: Distribution of requests and covered files for user-study participants. For box plots of phpMyAdmin and WordPress refer the left y-axis and for Magento, refer to the right y-axis.

verification of the functionality of the website (e.g., registering as new customers, submitting orders, etc.), maintenance tasks (e.g., backups, importing data, etc.) and even search-engine optimization.

For the second milestone, we ask our participants to spend one hour of their time on our instrumented web applications and perform the tasks that they listed earlier. This process provides them with the list of deliverables and expectations, and also enables us to validate their effort on this project.

For freelancing platforms that provide a time-tracking utility, we use this feature to verify the participation of users together with cross-validating their task report with our code-coverage traces. For submissions that did not follow our guidelines (e.g., did not spend enough time, skipped the majority of tasks in the reports, etc.) we asked the participants to revise their submission.

IRB Approval Since our experiments involved the assistance of real users, we obtained an Institutional Review Board (IRB) approval for our user study. Upon providing thorough details of our tasks and the human interactions, along with the information that we collect from the users, we obtained IRB approval on May 27, 2021.

Throughout this user study, we interviewed over 110 individuals, some of whom decided not to participate in our study due to reasons such as non-recurring and short-term nature of our tasks, their busy schedule, etc. Overall, we spent numerous weeks interviewing our participants and following up with them to ensure the timely delivery of their tasks. The cost of this experiment was approximately $1,000, most of which was used to pay the administrators in our user study and the remainder to pay for domain names and the hosting of virtual machines on public clouds.

3.2 Setup of Web Applications

To facilitate the setup of web applications for our user-study participants, we prepared the following environments:

- **phpMyAdmin** (version 5.1.0), with multiple pre-populated databases including the ones from WordPress and Magento web applications.
- **WordPress** (version 5.8) with an admin account, over 20 blog posts, multiple pages, and comments.
- **Magento** (version 2.3.5) configured with an inventory of over 1,000 products.

Each participant received their own instance with the admin credentials on a unique subdomain. We use a PHP code profiler to collect the usage traces from user interactions in the form of file and line-coverage data.

3.3 Web Application Roles and Usage Patterns

Looking at the usage traces (i.e., file and function coverage) of our user-study participants, we observe several patterns. Figure 1 shows the distribution of PHP files invoked as the result of the requests of each user. We observe that for different web applications, administrators sent a range of requests. For instance, looking at phpMyAdmin, we observe that the majority of administrators sent between 500-800 requests with some outliers who sent as little as 240 and as many as 1,292 requests which invoked 388-512 distinct PHP files. This variance in the number of requests and the invoked files indicates the difference in the usage patterns of our participants.

In this step, through the analysis of submitted reports and the code-coverage, we manually extracted common and unique access patterns. phpMyAdmin relies on its database backend (i.e., MySQL) to authentication users and enforce access-control. As a result, all users of phpMyAdmin have access to all features at the web application level (e.g., file upload, form submission, etc.), and access-control is only enforced when running SQL queries directly or indirectly through the UI. From our user-study logs, we observe a list of tasks that virtually all users performed. Creating new databases and tables, executing SQL queries, and using the import/export functionality of phpMyAdmin to backup and restore databases are among the commonly used features. On the contrary, only a subset of users changed the structure of existing tables and databases, or deleted data. From the export functionality, only a few users exported files with file extension other than SQL (e.g., CSV), and a few individuals used the provided filters to limit the query results.

WordPress on the other hand ships with six hard-coded roles and the default role (i.e., Administrator) has the highest permission. WordPress administrators must rely on third-party plugins to customize the roles. By analyzing the usage traces of our WordPress user-study participants, we observe a group of users that focused on customizing themes and installing plugins. Another group focused on the website content, and their tasks included creation of new blog posts, along with adding tags and keywords to existing posts to enhance the SEO. Interestingly, only a few individuals used the import/export functionality of WordPress to backup blog post content, or setup the RSS/WXR feeds.

Lastly, Magento provides the finest level of control over user permissions and roles. In this web application, administrators can define custom roles and assign permissions for individual sections of the administration panel. From our logs, we observe that the majority of users created new products, managed product inventory, and modified prices. Conversely, only a subset of users enabled sales promotions, gifts and shopping cart rules. Similarly, only *some* users customized the front-end UI of the website.

Based on our observations, we determine that web applications must provide a baseline functionality to all of their users. Beyond this baseline, only a subset of the provided features are required by some and not all of their users (e.g., different export file formats), and certain features are left unused (e.g., phpMyAdmin GIS visualization). Previous work on debloating web applications only focused on the latter (i.e., debloating features that are not required by *any* of the web application users). We identify the opportunity to provide customized debloating matching the needs of groups of

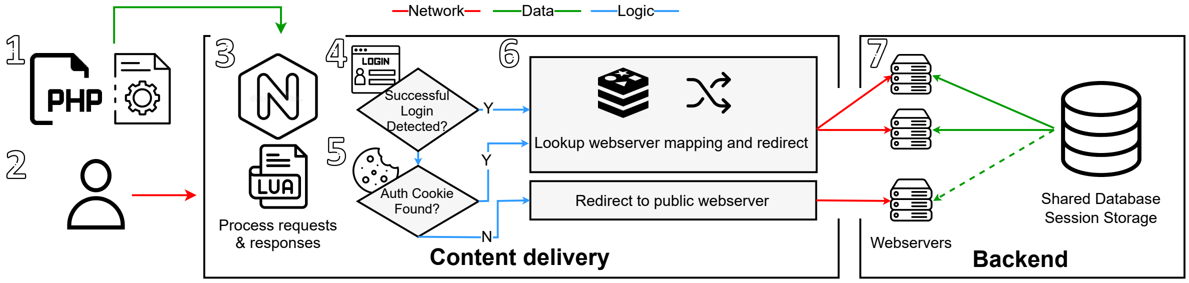

Figure 2: System Architecture of* DBLTR*. In Step 1, we provide the debloated web applications and user to cluster mappings to* DBLTR*'s content delivery module. User requests (2) are processed by* DBLTR*'s reverse-proxy (Step 3). After identifying the identity of the user (Steps 4-6),* DBLTR *internally routes the requests to the custom debloated web applications (Step 7).

users within a web application. Motivated by this observation, we discuss the design of our role-based debloating system in the next section.

4 SYSTEM DESIGN

In this section, we discuss the design of *DBLTR*, our role-based debloating system. *DBLTR* consists of data processing, code rewriting (i.e., debloating), and content-delivery steps, making up a full debloating pipeline. We evaluate the performance of *DBLTR* on the usage traces collected through our user study. Initially, *DBLTR*'s data-analysis step processes the code-coverage traces from web application users and identifies clusters of users with similar behavioral patterns. It then produces debloated copies of web applications customized to the needs of each cluster.

Figure 2 shows the end-to-end architecture of *DBLTR*. First, we review Step 1 in Section 4.1, which consists of analyzing the behavior of users, clustering them into roles and producing the debloated applications for each group of users with similar behavior. Then we discuss the design of *DBLTR*'s content delivery modules (Steps 3-7) in Section 4.2.

4.1 Processing the code-coverage information and debloating

We extract the list of file and line coverage information for all users of each web application. Next, we identify clusters of users that performed similar tasks. In order to cluster similar users together, we train an unsupervised clustering model based on source code features from the users' code-coverage.

4.1.1 Data preparation and clean up. Some web applications (e.g., Magento) create temporary PHP files for caching purposes. Other interactions with the web applications such as installing a new module can also result in the introduction of new PHP code. For our debloating scheme, we consider an application in a stable state (i.e., we assume that all the required plugins and modules are already installed prior to debloating). Therefore, we perform a cleanup step through which we remove references from code-coverage traces that point to non-existing files in the original version of the applications. Our cleanup step will effectively remove newly installed modules during the user study experiments, but will keep the code in the original application that enables users to install new modules.

4.1.2 Vectorizing code-coverage. *DBLTR* extracts a list of features representing the usage profiles from the code-coverage traces. Most commonly, web application source files are partitioned under directories that indicate the feature they implement. Moreover, for external dependencies (i.e., composer packages), the file-path includes the name of the module that the files belong to. The same holds true for namespaces, class names and function names in that they usually represent the underlying feature that they implement. *DBLTR*'s clustering does not need the naming conventions of the web application modules to be meaningful, as long as the naming scheme can uniquely represent the underlying feature that is being used.

DBLTR extracts file names, active namespaces, used classes, and invoked functions from the code-coverage traces. These features are effective indicators of the functionality corresponding to each user's code-coverage. We then map each feature to a unique representation in a binary feature space. Based on these features, we cluster the code-coverage of users using unsupervised clustering algorithms, and optimize the number of clusters (i.e., roles) to produce the best debloating (i.e., highest number of removed functions across roles). *DBLTR* then produces the following artifacts:

- *N* debloated variants of web applications, where *N* is the number of roles that optimizes the debloating by grouping users with similar behavior together.
- Configuration files for the reverse-proxy and the web servers to host the debloated web applications in a containerized environment.
- Database with the mapping of users to roles.

4.1.3 Measuring code-coverage similarity. Unsupervised clustering requires a measure for similarity between the data points (i.e., code-coverage of users). We use the Jaccard similarity coefficient to measure how dissimilar the code-coverage of two users are, and use it as the distance metric in our clustering. Under this scheme, we assign the distance of zero to users with identical code-coverage. Similarly, we assign a distance of one to users with no overlap in their code-coverage.

4.1.4 Clustering code-coverage information. We experimented with three different unsupervised clustering algorithms namely, K-means [17], Spectral clustering [24] and DBSCAN [9]. The output of the clustering step is a list of roles and the mapping of users to those roles.

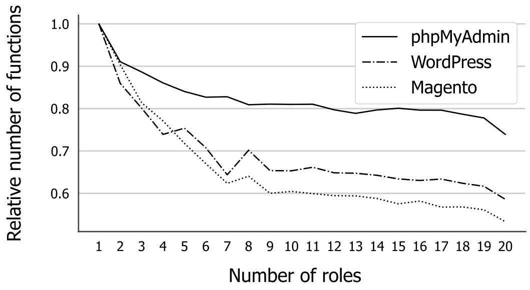

Figure 3: Reduction in number of functions after debloating for different number of roles compared to "Global" debloating (i.e., one role).

On one end, we build a single role which acts as the baseline for our debloating measurements. This role which contains the code-coverage of all users resembles the "global" debloating structure of Less is More [2]. On the other end, we can assign a unique role to each user. By doing so, each user receives their own uniquely debloated web application which ensures the maximum tailoring of features. We argue that the optimal number of roles lies in between these two extremes, such that the debloating metrics are maximized while minimizing the total number of roles.

After comparing the debloating statistics of different clustering algorithms, we observed that Spectral clustering outperformed the other algorithms. Therefore, we use Spectral clustering in *DBLTR* and report its debloating statistics in the remainder of this paper. Table 1 lists the LLOC reduction statistics for all the evaluated clustering algorithms.

4.1.5 Determining the optimal number of roles. *DBLTR* determines the optimal number of roles for each web application by using the elbow method [3]. By measuring the decrease in the average number of functions in the debloated copy of the web applications, *DBLTR* chooses the smallest number of roles that produce the highest reduction.

Figure 3 depicts the reduction in the average number of functions remaining in the web applications after debloating based on the total number of roles. Given the ways that the 60 administrators used the evaluated web applications during our user study, the optimal number of roles are as follows: six roles for phpMyAdmin, seven for WordPress, and seven for Magento.

4.1.6 Debloating the applications. For each role, we merge the code-coverage information for all the users of that role, and then use the aggregate file and line coverage information to identify unused files/functions and debloat them. In this step, we first perform file debloating and then debloat functions within the remaining files.

The process of debloating consists of neutralizing unused files and functions by replacing them with a routine that blocks the further execution of the code. By neutralizing unused files and functions, we immediately stop the execution of code paths that invoke the debloated code. This immediate termination would prevent the application from executing paths with debloated functions and potentially introducing new bugs. Moreover, this allows us to display an error message to the user and log this event for the administrators for further analysis (e.g., in the case of potential exploitation attempts).

Table 1: Minimum, median, and maximum size of debloated applications for optimal number of roles reported as thousands of LLOC.

Web Application	Clustering Algorithm	K-LLOC		
		Min	Median	Max
phpMyAdmin	Spectral	32	39	42
	DBSCAN	34	39	41
	K-means	32	40	42
WordPress	Spectral	44	54	64
	DBSCAN	47	59	64
	K-means	44	52	64
Magento	Spectral	241	270	326
	DBSCAN	251	275	310
	K-means	251	283	316

This debloating procedure provides us with N copies of the original application, with N being the optimal number of roles determined by *DBLTR*. Debloated applications for each role cater to the use cases of all their underlying users. Lastly, we measure the debloating metrics such as size, CVE, and CAC reduction across the debloated copies of our web applications. We discuss these results in more detail in Section 5.

4.2 Content delivery

The second stage of the *DBLTR* pipeline is responsible for serving the debloated web applications and seamlessly routing users to their underlying debloated web applications. *DBLTR* implements a reverse-proxy module based on OpenResty, which is a popular high performance scalable web server that extends NGINX and provides content manipulation APIs through Lua code [26]. This is depicted in Figure 2 as step 3.

We implement the login-detection logic as a Lua module for OpenResty. This module is responsible for detecting successful login requests, extracting username and session cookie information, as well as storing and retrieving the user-to-debloated-application mappings from the data store.

The login procedure for web applications consists of a request containing the credentials followed by the server response assigning the authentication cookie to the users in the case of successful login. *DBLTR* inspects the request-response pairs for successful login attempts. For instance in phpMyAdmin, a successful login comprises a POST request towards the login endpoint "/" or "index.php" that receives a 302 HTTP response code which redirects the user to the administration page of the application. We extract the username from the POST request with the field name of "pma_username" and then verify through the HTTP response code that we detected a successful login. Finally, we extract the session cookie named "phpmyadmin" and store this user-to-session-cookie mapping in the Redis data store for subsequent requests.

During the debloating stage, *DBLTR* produces mappings that directs our OpenResty module to redirect users to specific instances of debloated web applications. This information is stored in a Redis data store along with a hashed copy of the session cookie and username mappings extracted by the Lua module. Producing these Lua modules is a simple process for anyone with a basic understanding of web applications who can use approaches such as the Developer Tools of modern browsers to identify the right requests, and form fields. Once authored, these short modules (typically under 20 lines of Lua code and mostly made up of boilerplate code) will be valid for *all* deployments of that web application and will only need to be updated, if the web application changes its authentication logic in a later version.

For subsequent requests containing a valid session cookie, *DBLTR* first queries the username from the data store and then determines the target web server based on the username mapping. Steps 4, 5, and 6 in Figure 2 depict this process. Once the upstream web server is determined, traffic is routed to the corresponding web server. Any subsequent log-out requests and timeouts invalidate the session-cookie mapping.

5 DEBLOATING RESULTS

In this section, we measure the debloating performance of our system and demonstrate its ability to reduce the attack surface of web applications beyond the previous work. Next, we compare the debloating results of *DBLTR* and quantify its improvements over the baseline model through source code metrics such as LLOC reduction, as well as security metrics, namely CVE, gadget chain, and CAC reductions.

5.1 Debloating results

5.1.1 LLOC Reduction. We measure the reduction in size of web applications in terms of logical lines of code (LLOC), which counts the number of statements in the application source code. By reporting the size of the debloated applications in LLOC, we reduce the effects of various syntax and coding styles (i.e., the debloating process changing the code style after rewriting).

Table 2 shows the LLOC reduction results of our debloating scheme. The numbers are reported in terms of thousands of LLOC. The column marked as "Baseline" lists the debloating results of combining the code-coverage data of all users together and assigning them to a single role, which is equivalent to prior debloating approaches, such as the one by "Less is More" of Amin Azad et al. [2]. "*DBLTR*" shows the LLOC statistics with the optimal number of roles chosen by *DBLTR* for each web application, as discussed in Section 4.1.5. For *DBLTR* column, we report the size of the role with highest debloated LLOC (Max) and the smallest role with minimum removed LLOC (Min) along with the median. The number in the parenthesis for Baseline and *DBLTR* denotes the percentage of LLOC reduction with respect to the size of the original application.

By comparing the LLOC reduction of Baseline to *DBLTR*, it becomes evident that a one-size-fits-all debloating (i.e., Baseline), exposes certain users and roles to a much larger code-base than they actually require. This unnecessary bloat for Baseline debloating can be as high as 30% of extra LLOC for applications. This unnecessary exposure is most significant for larger web applications such as Magento where certain users inherit up to 90,000 unnecessary LLOC which is 36% larger—compared to the smallest *DBLTR* role—than what they actually need.

Moreover, we observe that *all DBLTR* roles (including the one with maximum remaining LLOCs) are strictly smaller than the Baseline, meaning that the globally debloated application is still bloated as far as individual users are concerned.

Table 2: LLOC reduction of debloated clusters: Numbers are reported in terms of thousands of LLOC. For Baseline, and* DBLTR*, we report the percentage of LLOC reduction compared to the Original application.* DBLTR *columns show the roles with maximum, median and minimum number of debloated LLOC.

Web Application	Original	Baseline	DBLTR		
			Max	Median	Min
phpMyAdmin	155	44 (▼72%)	32 (▼79%)	39 (▼75%)	42 (▼73%)
WordPress	103	66 (▼36%)	44 (▼57%)	54 (▼48%)	64 (▼38%)
Magento	1,050	330 (▼69%)	240 (▼77%)	270 (▼74%)	326 (▼69%)

5.1.2 CVE Reduction. One of the metrics to model the effects of debloating on the security of applications is removal of actual vulnerabilities. Historic CVEs provide a good source of information on vulnerabilities and we incorporate a mapping of CVE to source code to identify whether a debloated variant of applications includes the vulnerability or not.

One of the challenges of this approach is the availability of patch information. In order to map a CVE to the vulnerable parts of the source code, we use the data that is available in the form of bug report analysis, Git diffs, and security patches. The goal of this step is to identify the files and functions responsible for the vulnerability.

In our user study, we focused on the latest versions of web applications, so we opted to map all CVEs to these versions. In practice, this translates to mapping the location of a CVE to a specific function in a specific file, even if that function is currently patched. The purpose of this step is to identify whether the code that contained the vulnerability would have been retained by a debloating approach because it was part of the functionality that the administrators in our user study relied upon.

Some of the web applications in our dataset maintained a stable structure of their source code over time (e.g., WordPress) which makes the process of mapping CVEs from older versions to the recent one straightforward. Conversely, phpMyAdmin and Magento changed drastically since their older versions (i.e., Magento version 1.x vs 2.x) and it is not always possible to find the same PHP file/class to perform the mapping. Moreover, the developers of phpMyAdmin and WordPress usually acknowledge CVEs in their patches and GitHub commits, whereas for Magento, CVEs are only discussed with minimal details and the patches are released in the form of Major and Minor updates ranging from 500 to 4,000 modifications.

As a result, we mapped 20 CVEs to the source code of phpMyAdmin and WordPress, and mapped 10 recent CVEs to source code of Magento, for a total of 50 mapped CVEs. We selected the CVEs with the highest CVSS score, and skipped the ones where the vulnerability or patch information was unavailable.

Figure 4 depicts the number of CVEs remaining after debloating for each web application. The "*DBLTR*" bar shows the median CVE reduction across roles, while "*DBLTR*-Max" shows the roles with highest CVE reduction representing the maximum protection provided by *DBLTR* for users in those roles. For example, for phpMyAdmin we discover that 3/6 roles (accounting for 65% of users) exposed to the fewest remaining vulnerabilities. These roles contained 5 historic CVEs corresponding to 45% reduction of CVEs compared to the Baseline debloating and 75% reduction compared to the non-debloated application. Similarly, the median number of

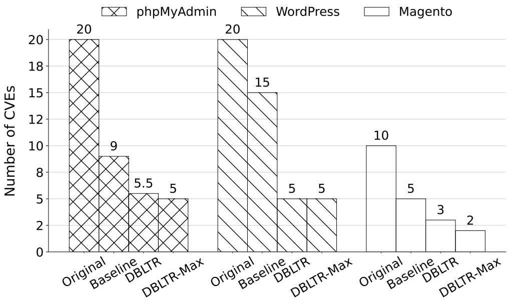

Figure 4: CVE reduction statistics of debloated web applications. "Original" represents the total number of mapped CVEs, "Baseline" represents the reduction of global debloating, "DBLTR" depicts the median reduction across roles, and "DBLTR-Max" represents the roles with the highest CVE reduction.

vulnerabilities among the roles was 5.5 accounting for 40% reduction compared to the Baseline approach. This effect is even more pronounced in WordPress where *DBLTR* reduces the median number of CVEs per role to 4 accounting for a 73% reduction compared to the 15 CVEs remaining after the Baseline debloating approach. Magento exhibits a similar trend of localized debloating gains.

Overall, our results demonstrate that debloating web applications based on clusters of usage-data (i.e., roles) results in significant reduction of severe historic CVEs, compared to prior debloating schemes that could only remove code that was determined to be globally unnecessary for all users of a deployed web application.

5.1.3 Case Study: phpMyAdmin Database Export Local file Inclusion Vulnerability. phpMyAdmin version 4.0 is vulnerable to CVE-2013-3240 which resides in the database export functionality. This vulnerability allows the attackers to bypass the `checkParameters` function by sending a specially-crafted variable. phpMyAdmin uses this variable to determine the database export file type (e.g., .sql, or .zip) and load the corresponding plugin. Malicious users can abuse this flaw to load and execute arbitrary PHP files from the server.

The export feature in phpMyAdmin is commonly used to backup existing databases and therefore is highly unlikely to be removed by prior global-debloating mechanisms. Nevertheless, we observed that one of our roles produced by *DBLTR* included four users who did not exercise the export functionality. As a result, *DBLTR* is able to remove this feature from the source code of that specific role, protecting the web application from abuse by these four specific users (including from attackers who compromise their accounts).

5.1.4 Critical API Calls Reduction. Another security metric that we analyze is the reduction in Critical API Calls (CACs). Figure 5 depicts the average CAC reduction across all roles. The first bar of each group shows the total number of CACs for the Baseline debloating. Baseline indicates the reductions in the global debloating scheme where all users are grouped into a single role. *DBLTR* bars show the average reduction of CACs based on the optimal number of roles for each web application and *DBLTR*-Max represents the maximum reduction in CACs for a subset of users.

Across all CAC categories and all web applications, we observe a reduction of 10% up to 70% for *DBLTR* debloating. This reduction indicates that a sizable number of CACs are in unused parts of the applications. Upon closer inspection of phpMyAdmin results, it becomes evident that 85% of code execution APIs reside in the external dependencies of the web application, out of which, *DBLTR* removed 12%-20%. For the larger applications such as Magento, the reduction in Code Execution APIs is more significant where 32%-53% of these APIs are removed by *DBLTR*. Over all categories of CACs, we observe that *DBLTR* removes tens to hundreds of such API calls beyond the Baseline, further protecting web applications from exploits that target these APIs.

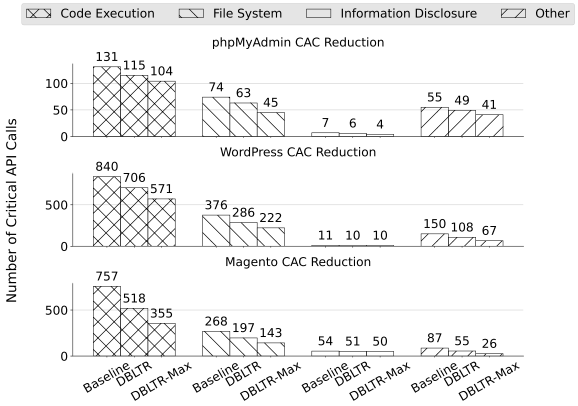

Figure 5: Critical API Call (CAC) reduction after debloating. Baseline represents the global debloating approach where all users are assigned to the same role.* DBLTR *indicates the average reduction across roles.

5.1.5 PHP Object Injection Gadget Reduction. To identify existing object injection gadgets in the applications, we incorporate PHPGGC [35], an open-source project listing available gadget chains for popular composer packages. After mapping the list of vulnerable package with those in phpMyAdmin, WordPress, and Magento, we search for the removal of classes and functions used within the gadget chains after debloating.

Table 3 lists the packages in each of our web applications with a known gadget chain based on PHPGGC. Under the column named "Original" we list the number of gadget chains within each package. As before, the Baseline column lists the number of gadget chains available after the global debloating (i.e., single role), whereas the *DBLTR* column lists the percentage of roles in *DBLTR* exposed to these gadgets.

The TCPDF package in phpMyAdmin includes one known gadget chain that can lead to arbitrary code execution. While global debloating (Baseline) does not remove this gadget chain, *DBLTR* removes it for 17% of the roles, thereby protecting the web application from users belonging to that role. For WordPress, the baseline debloating strategy removes the existing gadget therefore there are no opportunities for any additional gains by *DBLTR*.

More interestingly, Magento contains 10 known gadget chains. For Magento's Guzzle package, we observe that while 2 gadgets are still present in the Baseline debloating, one of the gadget chains is fully removed from all roles. For the gadget chain within the Magento package itself, only (1/7) 14% of roles produced by the *DBLTR* contain this gadget, therefore, the majority of users are protected.

Table 3: PHP Object Injection Gadgets statistics after debloating. Listing number of existing gadgets for Baseline and the percentage of roles in* DBLTR *exposed to those gadgets.

Web Application	Package	Original	Baseline	*DBLTR*
phpMyAdmin	Symfony	2	0	0
	TCPDF	1	1	83%
WordPress	Generic	1	0	0
Magento	Guzzle	3	2	100%, 0%
	Magento	1	1	14%
	Monolog	6	0	0

Our results confirm the findings of previous work that debloating is a highly effective defense for removing publicly known object injection gadget chains. Moreover, we observe that by clustering users into multiple groups, we can further breakdown the availability of gadgets and in certain cases, further complicating the exploitation of object injection vulnerabilities. Under a *DBLTR*-protected deployment, attackers not only need to identify the available gadgets in the target web application to build their exploit chain, but they also have to target *specific* victims who have access to the underlying gadgets used in their exploits.

6 DISCUSSION

In this section we discuss and evaluate possible strategies for handling the addition and removal of users from a *DBLTR*-protected web application. We then review the important takeaways, and finally discuss the limitations of our approach.

6.1 Addition and Removal of Users

The process of assigning a role to new users in RBAC web applications is based on an administrator's discretion regarding the required capabilities of the new users. Unlike traditional RBAC roles, *DBLTR* roles are defined dynamically and are unique to each deployment of a web application based on the behavior of its users. As a result, assigning new users to existing *DBLTR* roles requires special treatment.

The conservative approach is to assign new users to a non-debloated web application and record their usage behavior. *DBLTR* can straightforwardly handle this by the mere introduction of a new containerized environment containing that non-debloated web application and a mapping rule assigning the new user to that container. A more aggressive approach is to assign users to a *globally-debloated* web application with the expectation that a newly added user will use features used by at least one existing user of that web application. For both strategies (i.e., conservative and aggressive assignment), administrators can collect usage traces for that new user and eventually invoke *DBLTR* to produce new roles and migrate the new user to a more tailored cluster of users.

To evaluate the latter more aggressive user-assignment strategy, we start by assuming a setup for *DBLTR* where the usage traces for the majority of users have been collected and the roles are produced. This setup resembles an environment (e.g., a company) where the majority of employees were present for the usage-trace collection period and a limited number of new users (e.g., newly-hired employees) are periodically added to this steady-state system. When a new user is added to the system, we use the sum of the code-coverage of all existing users to produce a globally-debloated web application, which still contains a significantly smaller attack surface compared to the original application.

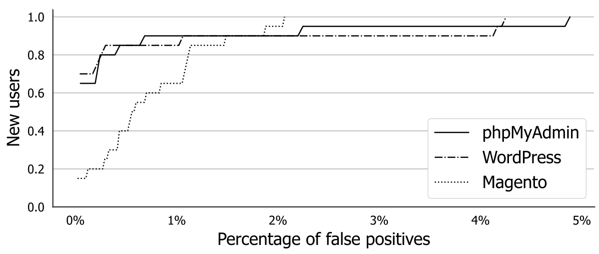

Figure 6: CDF of observed false positives from the perspective of new users. The X axis depicts the percentage of false positives (i.e., ratio of the number of missing files compared to total used files by each new user).

To simulate this, we conduct the following leave-one-out experiment: we remove the code-coverage information of each user from the training dataset and create a debloated copy of the web application based on the code-coverage of remaining users (i.e., 19/20 users). This globally-debloated web application is strictly larger than any of the role-specific copies of the same application and is equivalent to prior dynamic debloating approaches [2]. We then simulate the addition of new users by introducing the code-coverage of the user that we left out and measuring false positives, i.e., the files required by that user that are not present in the globally-debloated web application.

Figure 6 provides the CDF of false positives (i.e., missing files) when adding new users. For phpMyAdmin and WordPress, the majority (60-70%) of users in the leave-one-out experiment could be added to the "globally-debloated" web application without any false positives. Likewise, more than 85% of new users would experience breakage in less than 1% of the overall files that they interact with. For Magento, while most new users assigned to the "globally-debloated" web application would experience some breakage, most still fall below the 1% threshold. Overall, across all three web applications in our dataset, even for users with notably unique behavior, we observe less than 5% of the overall files missing when their usage behavior traces were omitted from the training dataset.

Looking at the unique behavior of users that led to false positives, in phpMyAdmin, we observe enabling multistep authentication modules, using query explainer features, and creating SQL Views to be the underlying cause (i.e., desired functionality that is unavailable to new users). For WordPress, customizing the RSS feed, using specific text blocks in posts such as text art, and quotes, and using less popular embedding protocols led to false positives under this aggressive user-assignment strategy. Finally, for Magento, various users interacted with unique features. These features include third-party integrations (e.g., monitoring, marketing, and automation services), PDF invoices, and wish lists.

Overall, our results show that the modular architecture of *DBLTR* allows it to successfully serve different types of environments. For environments with fixed workloads or where security is prioritized over usability, *DBLTR* can be used to assign new users to a globally-debloated web application. Alternatively, when administrators are unsure about the needs of new users, *DBLTR* can be used to serve the original version of a web application to these users, until enough usage traces are collected to allow *DBLTR* to create new roles and clusters. In either scenario, the security-related debloating benefits of existing users are not in any way compromised when new users are added to a system. In terms of removing users, administrators can merely disable user-role mappings in *DBLTR*'s configurations and optionally delete containers that do not have any roles associated with them.

6.2 Main Takeaways

Static roles in web applications are over authorized and administrators only need a subset of the available features: Throughout our user study, we determined that administrators of the same web application use different subsets of the features available to them. While in theory, administrators have full access to every feature in the administration panel, in practice only 25% and 52% of the lines in the administrative panels of phpMyAdmin and WordPress respectively, were among the commonly exercised features. Evidently, the existing authorization mechanisms of these applications aim to offer access to a large variety of features whereas, in reality, administrators only need a subset of them. *DBLTR* builds an accurate list of required features and enforces the principle of least privilege via debloating.

One-size-fits-all debloating still produces bloated web applications: As demonstrated by this work and the literature, debloating web applications is highly effective in reducing the attack surface of web applications by removing unused features and their underlying vulnerabilities.

The global debloating approach explored by the prior work produces one debloated web application which as demonstrated by our analysis, can contain as much as 29% extra LLOC compared to what users actually need. In this work, we integrated *DBLTR* with a dynamic debloating scheme and demonstrated that *DBLTR* can provide improvements across all debloating metrics. We demonstrated that in the case of global debloating, users in at least half of the roles would be provided with larger web applications containing 5,000-60,000 more lines of code than required, and exposed to 40%-70% more CVEs.

***DBLTR* provides a content delivery environment for debloated web applications:** One of the main contributions of our work is our content delivery pipeline which reduces the need for modifications and customizations to target web applications, while keeping the debloating platform entirely invisible to web application users.

6.3 Limitations

Usage behavior modeling: Our debloating reports are based on the code-coverage that we collected during our user study. Using our domain knowledge, we have established that the collected dataset is a representative sample of web-application use in the real world. At the same time, different deployments of web applications may be used differently and therefore be debloated differently. Regardless of the degree of use, *DBLTR* can offer concrete security advantages to any existing or future end-to-end debloating strategy by automatically removing code that is not globally required by all users of a given deployment and clustering users to their appropriately-debloated codebase.

Applications with a public interface: As depicted in Figure 2, *DBLTR* routes unauthenticated requests towards a "public" profile of the debloated web application. This profile includes the code for user authentication as well as any other code required for public unauthenticated users.

For administrative applications that are behind an authentication wall, producing a public profile is straightforward. In the example of phpMyAdmin, we only need to perform successful and failed login attempts to generate the baseline code-coverage and debloat the application to produce the public debloated profile. For other applications such as WordPress and Magento, this step is more involved. First, we would remove all the files that are only available to the administrators, for the WordPress and Magento, this constitutes of removing administrator directories (e.g., *wp-admin* for WordPress and *module-backend* directory for Magento). Next, we need to only retain the features and modules that are available to public users. Therefore, we need to record the code-coverage of unauthenticated users with benign behavior.

One of the main benefits of our role-based debloating is the removal of features that are not limited by the authentication and authorization boundaries of web applications. If attackers can somehow taint the code-coverage of unauthenticated profiles to include a vulnerable piece of code, they can force the debloating pipeline to retain that code, and exploit it later. This only applies to the potential vulnerabilities in the public interface of the applications. A possible solution to this problem is to use artificial modeling techniques, such as automated crawlers, to extract the code-coverage of public users.

Changes to the source code: In this work, we studied the debloating of web applications at a stable state. That is, all the required configurations, updates, and plugins were installed and available at the time of debloating. For smaller updates, we need to repeat the debloating to produce new copies of the updated web application while not touching the modified files during the update. For major version updates that include drastic changes to the architecture of the source code and modules, we would need to collect the code-coverage traces again. This limitation is shared by all debloating systems that offer security benefits via late-stage code transformations. Note however that *DBLTR* can be used to stage a move from the old version of a web application to the new version by slowly migrating users from their old containerized environments to the new ones, one role at a time.

Number of users of the web applications: In our user study, we hired a total of 60 participants (20 participants for each web application). While most websites are operated by a small number of administrators, there clearly exist web applications (such as popular social networks) with billions of users and thousands of administrators. Understanding how *DBLTR* could be used in such an environment requires the collaboration of a large operator, something which we do not have access to. *DBLTR*'s current architecture does allow for horizontal scaling of servers, enabling it to serve an arbitrary number of users and roles. As such, we hope that, through the open-sourcing of our system, large organizations will be able to evaluate *DBLTR* in their environments and user populations.

7 RELATED WORK

The idea of software debloating was initially discussed by Zeller et al. [40] as a means to isolate failure-inducing code. This idea was later applied to the context of software security to reduce the attack surface of applications. Ghavamnia et al. and Rastogi et al. explored the idea of debloating containers [12, 33], while Abubakar et al. debloated the kernel [1]. Orthogonally, another line of research explores binary debloating [13, 14, 19, 22, 31, 32, 34, 36], and debloating web applications [2, 5, 15, 18].

At a high level, there are three mainstream approaches to debloating: i) using static analysis to identify unreachable code [13, 18, 32, 34, 36], ii) debloating reachable code which is unused given a set of tests (e.g., automated test cases, or dynamic code-coverage traces) [2, 14, 19, 31], and finally, iii) API specialization, which consists of disabling sensitive APIs or hardening them with respect to the execution context of applications [5, 15, 22, 23].

Our work is mainly motivated by the "Less is More" approach of Amin Azad et al. [2]. By comparing the debloating results of *DBLTR* with the Baseline debloating approach of "Less is More" (Section 5), we demonstrated that role-based debloating outperforms the "Less is More" approach, both in terms of security metrics and reducing concrete vulnerabilities.

In another line of work targeting binaries and web applications, Mishra et al. [21, 23], Bulekov et al. [5], and Jahanshahi et al. [15] provided solutions to reduce the software attack surface of applications through limiting the list of available APIs for each piece of code. By incorporating their defense, attackers are limited in their ability to exploit the application vulnerabilities. These solutions are orthogonal to our work and can be used in combination with *DBLTR* to further protect the applications against attacks.

Koishybayev et al. proposed Mininode, a tool to debloat Node.js applications by focusing on third-party modules [18]. Their approach is based on static analysis which they use to identify unreachable code in third-party modules and the chain dependencies of Node.js applications. While static analysis is helpful in identifying unused code, several categories of common web application vulnerabilities (e.g., SQLi, XSS, CSRF, etc.) reside in reachable parts of the source code. *DBLTR* incorporates dynamic analysis and therefore, is capable of debloating even the reachable but unused parts of the code.

Bocic et al. and Son et al. studied access-control bugs in web applications [4, 37]. They analyzed open-source Ruby on Rails and PHP applications and identified over 100 authorization bugs. Their findings reinforce the motivation for *DBLTR*'s role-based debloating which guarantees the separation of public vs. authenticated users, even in the presence of access-control errors.

8 CONCLUSION

In this paper, we explored the idea of "role-based debloating", which consists of producing multiple versions of debloated web applications, each tailored to a cluster of users with similar usage behaviors (i.e., roles). We started by conducting a user study to understand how experienced developers and administrators interact with web applications. We then used this data in combination with *DBLTR*, our proposed tool that is capable of collecting code-coverage traces of the users of an application, forming clusters of users with similar usage behavior, and producing differently-debloated applications customized to the needs of each group. *DBLTR* also includes a content-delivery pipeline that can transparently route users to their clusters of dedicated debloated web applications without the need to modify the target web applications.

Through our detailed analysis, we quantitatively showed that *DBLTR* can outperform the state-of-the-art in web application debloating. By incorporating the idea of role-based debloating, we can produce debloated web applications that are 30% smaller in size, and contain 80% fewer severe vulnerabilities (i.e., historic CVEs) compared to the "globally" debloated web applications produced by prior work. We also explored the contribution of each user to the code-coverage of roles, in an effort to understand the robustness of clustered debloating compared to the extreme where each users receives their own copy of the debloated applications.

We showed that *DBLTR*'s clustering expands the code-coverage of similar features in each role for up to 38% of all files, which affects more than half of the packages and classes in the web applications. This effect on the code-coverage allows *DBLTR* to retain the code for similar features that role members may use in future. Overall, our results demonstrate that role-based debloating is a superior approach compared to past global-debloating approaches, with tangible benefits, both in terms of security (greater degree of attack-surface reduction) as well as usability (lower likelihood of breakage and support for "live" introduction/migration of users into new and existing debloating clusters).

Acknowledgements: This work was supported by the Office of Naval Research (ONR) under grant N00014-21-1-2159 as well as by the National Science Foundation (NSF) under grants CNS-1813974, CNS-1941617, and CNS-2126654.

9 AVAILABILITY

To ensure full transparency while promoting future work in the space of debloating web applications, we will provide public access to *all* developed code and artifacts upon publication of this paper at https://dbltr.debloating.com.

REFERENCES

[1] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. *SHARD*: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *Proceedings of the 30th USENIX Security Symposium.*

[2] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium.*

[3] Purnima Bholowalia and Arvind Kumar. 2014. EBK-means: A clustering technique based on elbow method and k-means in WSN. *International Journal of Computer Applications* (2014).

[4] Ivan Bocić and Tevfik Bultan. 2016. Finding access control bugs in web applications with CanCheck. In *31st IEEE/ACM International Conference on Automated Software Engineering.*

[5] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. 2021. Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists. In *Proceedings of the 30th USENIX Security Symposium.*

[6] Johannes Dahse and Jörg Schwenk. 2010. RIPS-A static source code analyser for vulnerabilities in PHP scripts. In *Seminar Work. Horst Görtz Institute Ruhr-University Bochum.*

[7] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. 2009. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. (2009).

[8] Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. 2011. Fear the EAR: Discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.*

[9] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96).*

[10] Fiverr. 2022. *The online marketplace for freelance services.* https://fiverr.com

[11] Ivan Fratrić. 2012. ROPGuard: Runtime prevention of return-oriented programming attacks. *Technical report* (2012).

[12] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses.*

[13] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium.*

[14] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.*

[15] Rasoul Jahanshahi, Adam Doupé, and Manuel Egele. 2020. You shall not pass: Mitigating sql injection attacks on legacy web applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security.*

[16] Pawan Jaiswal. 2022. *WordPress File Manager Plugin Unauthenticated RCE Exploit.* https://medium.com/swlh/wordpress-file-manager-plugin-exploit-for-unauthenticated-rce-8053db3512ac

[17] Xin Jin and Jiawei Han. 2010. *K-Means Clustering.* Springer US.

[18] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses.*

[19] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security.*

[20] Steve McConnell. 2004. *Code complete.* Pearson Education.

[21] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking exploits through API specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference.*

[22] Shachee Mishra and Michalis Polychronakis. 2020. Saffire: Context-sensitive Function Specialization and Hardening against Code Reuse Attacks. In *IEEE European Symposium on Security & Privacy.*

[23] Shachee Mishra and Michalis Polychronakis. 2021. SGXPecial: Specializing SGX Interfaces against Code Reuse Attacks. In *Proceedings of the 14th European Workshop on Systems Security.*

[24] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. 2001. On Spectral Clustering: Analysis and an Algorithm. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic* (Vancouver, British Columbia, Canada) *(NIPS'01).* MIT Press, Cambridge, MA, USA, 849–856.

[25] NPM. 2022. *Node Package Manager Statistics.* https://blog.npmjs.org/post/615388323067854848/so-long-and-thanks-for-all-the-packages.html

[26] OpenResty. 2022. *Scalable Web Platform by Extending NGINX with Lua.* https://openresty.org/en/

[27] Packagist. 2022. *The PHP Package Repository.* https://packagist.org/statistics

[28] Packagist. 2022. *Popular PHP Packages.* https://packagist.org/explore/popular

[29] Vasilis Pappas. 2012. kBouncer: Efficient and transparent ROP mitigation. (2012).

[30] PyPI. 2022. *Package Download Statistics.* https://pypistats.org/top

[31] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.*

[32] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium.*

[33] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.*

[34] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. Bintrimmer: Towards static binary debloating through abstract interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.*

[35] Ambionics Security. 2017. *PHPGGC: PHP Generic Gadget Chains.* https://github.com/ambionics/phpggc

[36] Peter Snyder, Cynthia Taylor, and Chris Kanich. 2017. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.*

[37] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing Access-Control Bugs in Web Applications.. In *NDSS.*

[38] Statistica. 2022. *How many websites are there?* https://www.statista.com/chart/19058/number-of-websites-online/

[39] Upwork. 2022. *The marketplace for freelancers.* https://upwork.com

[40] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002).