Web Application Attack Surface Reduction Through Software Debloating

Babak Amin Azad

Department of Computer Science

Adviser: Nick Nikiforakis

July 2019

Abstract

As software becomes increasingly complex, its attack surface expands enabling the exploitation of a wide range of vulnerabilities. Web applications are no exception since modern HTML5 standards and the ever-increasing capabilities of JavaScript are utilized to build rich web applications, often subsuming the need for traditional desktop applications. One possible way of handling this increased complexity is through the process of software debloating, i.e., the removal not only of dead code but also of code corresponding to features that a specific set of users do not require. Even though debloating has been successfully applied on operating systems, libraries, and compiled programs, its applicability on web applications has not yet been investigated.

In this report, we present the first analysis of the security benefits of debloating web applications. We focus on four popular PHP applications and we dynamically exercise them to obtain information about the server-side code that executes as a result of client-side requests. We evaluate two different debloating strategies (file-level debloating and function-level debloating) and we show that we can produce functional web applications that are 46% smaller than their original versions and exhibit half their original cyclomatic complexity. Moreover, our results show that the process of debloating removes code associated with tens of historical vulnerabilities and further shrinks a web application's attack surface by removing unnecessary external packages and abusable PHP gadgets.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Despite its humble beginnings, the web has evolved into a full-fledged software delivery platform where users increasingly rely on web applications to replace software that traditionally used to be downloaded and installed on their devices. Modern HTML5 standards and the constant evolution of JavaScript enable the development and delivery of office suites, photo-editing software, collaboration tools, and a wide range of other complex applications, all using HTML, CSS, and JavaScript and all delivered and rendered through the user's browser.

This increase in capabilities requires more and more complex server-side and client-side code to be able to deliver the features that users have come to expect. However, as the code and code complexity of an application expands, so does its attack surface. Web applications are vulnerable to a wide range of client-side and server-side attacks including Cross-Site Scripting [5, 49, 75], Cross-Site Request Forgery [4, 48, 35], Remote Code Execution [20], SQL injection [21, 43], and timing attacks [37, 42]. All of these attacks have been abused numerous times to compromise web servers, steal user data, move laterally behind a company's firewall, and infect users with malware and cryptojacking scripts [51, 77, 45].

One possible strategy of dealing with ever-increasing software complexity is to customize software according to the environment where it is used. This idea, known as *attack-surface reduction* and *software debloating*, is based on the assumption that not all users require the same features from the same piece of software. By removing the features of different deployments of the same software according to what the users of each deployment require, one can reduce the attack surface of the program by maintaining only the features that users utilize and deem necessary. The principle of software debloating has been successfully tried on operating systems (both to build unikernel OSs [55] and to remove unnecessary code from the Linux kernel [54, 53]) and more recently on shared libraries [58, 63] and compiled binary applications [44].

In this report, we present the first evaluation of the applicability of software debloating for web applications. We focus on four popular open-source PHP applications (phpMyAdmin, MediaWiki, Magento, and WordPress) and we map the CVEs of 69 reported vulnerabilities to the source code of each web application. We utilize a combination of tutorials (encoded as Selenium scripts), monkey testing, web crawling, and vulnerability scanning to get an *objective* and *unbiased* usage profile for each application. By using these methods to stimulate the evaluated web applications in combination with dynamically profiling the execution of server-side code, we can precisely identify the code that was executed during this stimulation and therefore the code that should be retained during the process of debloating.

Equipped with these server-side execution traces, we evaluate two different debloating strategies (file-level debloating and function-level debloating) which we use to remove unnecessary code from the web applications and quantify the security benefits of this procedure. Among others, we discover an average reduction of the codebase of the evaluated web application of 33.1% for file-level debloating and 46.8% for function-level debloating, with comparable levels of reduction in the applications' cyclomatic complexity. In terms of known vulnerabilities, we remove up to 60% of

known CVEs and the vast majority of PHP gadgets that could be used in Property Oriented Programming attacks (the equivalent of Return-Oriented Programming attacks for PHP applications).

Overall, our contributions are the following:

- We encode a large number of application tutorials as Selenium scripts which, in combination with monkey testing, crawling, and vulnerability scanning, can be used to objectively exercise a web application. Similarly, we map 69 CVEs to their precise location in the applications' source code to be able to quantify whether the vulnerable code could be removed during the process of debloating.

- We design and develop an end-to-end analysis pipeline using Docker containers which can execute client-side, application stimulation, while dynamically profiling the executing server-side code.

- We use this pipeline to precisely quantify the security benefits of debloating web applications, finding that debloating pays large dividends in terms of security, by reducing a web application's source code, cyclomatic complexity, and vulnerability to known attacks.

To motivate further research into debloating web applications and to ensure the reproducibility of our findings, we are releasing *all* data and software artifacts.

Chapter 2

Related Work

Over the years, different approaches that target very different parts of the software stack have been studied in the context of software debloating. The key characteristics that distinguish debloating approaches are the following:

1. **What is being removed and the definition of bloat:** On one hand removing dead code from included libraries and external dependencies has been studied [46, 47, 63]. While this approach does not target the actual vulnerable parts of the code (e.g., the source of a buffer overflow), it will raise the bar for attackers to exploit such systems by removing gadgets used in Return Oriented Programming (ROP) [69]. On the other hand, similar to our work, there are approaches that target the actual features that are unused under certain circumstances [36, 44, 65, 72, 74]. By defining the bloat as unused code in the applications, as we show later in the results section for web applications, one can remove the vulnerable pieces of code and reduce the attack surface of applications significantly.

2. **Identification of bloat:** The other difference arises from the underlying mechanism used to identify the bloat. Depending on target application and the platform, static analysis might need to be augmented by dynamic analysis to

detect unused code. In addition to that, the literature has incorporated application configuration files [52], high level manual specifications [44, 70] and usage profiles [36] with dynamic application instrumentation to detect unused code. Contrastingly, Delta Debugging is used [79]. This category of algorithms are different in that they start by removing lines of source code and then verify if the reduction is valid [44, 65, 74].

3. **Evaluation & metrics:** Based on the context where debloating is introduced (e.g., To ease the debugging process, reduce the software footprint or reduce the attack surface) different metrics has been used to measure the success of debloating strategies. In the context of security, reduction in size of the target application (i.e., Size of the output binary or number of lines of code), reduction in the number of gadgets available for attackers and the number of removed historical vulnerabilities or exploits are measured and reported.

2.0.1 Debloating for the web

Despite the importance of the web platform, there has been very little work that attempts to apply debloating to it. Snyder et al. investigated the costs and benefits of giving websites access to all available browser features through JavaScript [73]. The authors evaluated the use of different JavaScript APIs in the wild and proposed the use of a client-side extension which controls which APIs any given website would get access to, depending on that website's level of trust. Schwarz et al. similarly utilize a browser extension to limit the attack surface of Chrome and show that they are able to protect users against microarchitectural and side-channel attacks [68]. These studies are orthogonal to our work since they both focus on the client-side of the web platform, whereas we focus on the server-side web applications.

Boomsma et al. performed dynamic profiling of a custom web application (a PHP application from an industry partner) [36]. The authors measured the time it takes

for their dynamic profile system to get complete coverage and the percentage of files that they could remove. Since the application was a custom one, the authors were not able to report specifics in terms of the reduction of the programs attack surface, as that relates to CVEs. Contrastingly, by focusing on popular web applications, and utilizing function-level as well as file-level debloating, we were able to precisely quantify the reduction of vulnerabilities, both in terms of known CVEs as well as gadgets for PHP object-injection attacks.

Next, we will discuss two intances of related work that are more inline with our work, in more detail.

Most Websites Don't Need to Vibrate

This work targets the HTML Standards and browser APIs exposed to JavaScript [73].

Definition of bloat: Bloat is defined as the set of browser standards that expose high cost and low benefit. Cost of a standard is defined as a function of historic CVEs affecting the standard, academic papers introducing attacks abusing the standard and number of lines of code in C++, used exclusively to implement the standard. Conversely, the benefit is a function of number of websites that require that feature to function correctly. The judgement of correct functionionality is as perceived by end users while casually browsing through websites.

Identification of bloat: Based on the cost of each feature, and the number of websites that use that feature, the decision is made to remove individual standards. Through a set of user studies, two users browser top Alexa websites and report if removal of each feature breaks the functionality of these websites.

Mapping the JavaScript API to its backend C++ code is a challenging task. A call graph is generated to extract that information. By following the path from JavaScript APIs to auto-generated bindings and then to the C++ implementation

of each feature, the authors find functions within the source code that exclusively implement each functionality.

Evaluation & metrics: CVE reduction and removed lines of code are the two metrics reported by the authors of this work. Moreover, they propose two modes of debloating: "Conservative" and "Aggressive". The former tries to keep most number of websites functional and the latter focuses on attack surface reduction. Overall, the results of both modes are promising. They break less websites in comparison with NoScript browser extension [11] and have a comparable website break rate to Tor browser.

After identifying the risky and less used features, the JavaScript APIs are removed to prevent websites from accessing them. The APIs are removed through a browser extension that injects a JavaScript proxy object to modify target APIs and their properties such that calls to removed code fails gracefully. Additionaly, to prevent introducing new vulnerabilities due to gracefully blocked security related functions, WebCrypto APIs are whitelisted.

By mapping 1,544 CVEs to 175 web standards within Firefox browser, the authors show that a reduction of 66% of standards is possible with no noticeable effect on users experience. By preventing untrusted websites from accessing such features the attack surface can be reduced significantly.

Dead Code Elimination for Web Systems Written in PHP: Lessons Learned from an Industry Case

This work focuses on PHP applications and proposes a system that dynamically detects dead code and eliminates it. The author's goal is to increase the maintainability of the software by eliminating unused files and reducing the total number of files that the developers have to maintain.

Definition of bloat: Bloat is defined as files that are not executed by users during their normal use of the application.

Identification of bloat: Using their system, the authors are able to detect unused files over time. Based on factors such as "First usage", "Number of invocations", "Last time used" and "Version control last update", potentially dead files are dynamically discovered and removed. Their approach, that is based on dynamic code coverage faces several inherent challenges. First, some files are rarely used despite not being totally useless. Examples of such files would be scheduled tasks that run monthly or products that are purchased very rarely. In addition to that, error handlers and files under development also receive minimal code coverage from users of the application. As such, a whitelisting mechanism is used in addition to version control system timestamps to detect the last time a file was updated to prevent the removal of features under active development.

Evaluation & metrics: With the main forcus of this paper being software engineering metrics, only the file coverage is reported and security implications of using this type of debloating has not been discussed. As the first step in our work, we implement file level debloating and also report on the security effects of this method.

After gathering dynamic code coverage on real deployments of multiple PHP applications used by an industry partner, a reduction of 27% to 64% in the number of files is achieved. For different subsystems, it took 1 to 5 months for the code coverage to stabilize, meaning no new files were covered during execution of the application after that period.

2.0.2 Debloating in other platforms

Regehr et al. developed *C-Reduce* which is a tool that works at the source code level [65]. It performs reduction of C/C++ files by applying very specific program

transformation rules. Sun et al. designed a framework called *Perses* that utilizes the grammar of any programming language to guide reduction [74]. Its advantage is that it does not generate syntactically invalid variants during reduction so that the whole process is made faster.

Heo et al. worked on *Chisel* whose distinguishing feature is that it performs fine-grained debloating by removing code even on the functions that are executed, using reinforcement learning to identify the best reduced program [44].

All three aforementioned approaches are founded on Delta debugging [79]. They reduce the size of an application progressively and verify at each step if the created variant still satisfies the desired properties.

Sharif et al. proposed *Trimmer*, a system that goes further than simple static analysis [70]. It propagates the constants that are defined in program arguments and configuration files so that it can remove code that is not used in that particular execution context. However, their system is not particularly well suited for web applications where we remove complete features. Our framework goes beyond this contextual analysis by mapping what is actually executed by the application.

Other works include research that revolves mainly around static analysis to remove dead code. Jiang et al. looked at reducing the bloat of Java applications with a tool called *JRed* [46]. Jiang et al. also designed *RedDroid* to reduce the size of Android applications with program transformations [47]. Quach et al. adopted a different approach by bringing dead-code elimination benefits of static linking to dynamic linking [63].

Rastogi et al. looked at debloating a container by partitioning it into smaller and more secure ones [64]. They perform dynamic analysis on system-call logs to determine which components and executables are used in a container, in order to keep them. Koo et al. proposed configuration-driven debloating [52]. Their system removes unused libraries loaded by applications under a specific configuration. They

test their system on Nginx, VSFTPD, and OpenSSH and show a reduction of 78% of code from Nginx libraries is possible based on specific configurations.

Next, we discuss three instances of related work that are representative of the categories of debloating approaches.

Simplifying and Isolating Failure-Inducing Input

One line of work in the literature of software debloating is based on Delta Debugging algorithm. This idea was first introduced by Zeller et. al. in 2002 [79]. Given a failure inducing piece of code as input, Delta Debugging automatically minimizes the size of the test case by searching the program space for smaller pieces of code that would still cause the target application to run into error.

Definition of bloat: Under Delta Debugging, bloat is defined as extra code in a failure inducing input, such that their removal does not affect the outcome of the execution of the program, which is to throw an error in this case.

Identification of bloat: The input is split into chunks of an arbitrary size n. Then the algorithm verifies if the first chunk or the n-1 chunks would cause the target application to crash. If neither would satisfy this goal, n is increased and the algorithm iterates using smaller chunks until we can isolate the part where the cause of the error lies. This way, we will end up with a minimal piece of code that still breaks the target test case.

Evaluation & metrics: Minimizing the number of lines of input is the target of this algorithm. The performance of this system is measured as the execution time required to find the minimal input.

Without any prior knowledge of the input program and syntax, Delta Debugging has shown to effectively find minimal inputs on real bug reports. In their first case

study, DD is able to reduce an input that would crash Firefox browser from 95 actions to 3 actions (openning the print dialogue and mouse down and up events on print button), and a further reduction from 896 to 1 line for the html input to be printed. Ultimately, the bug report is summarized as "Printing a page with a SELECT tag would cause the browser to crash". The execution time to minimize this bug report is 35 minutes on a 500 MHz PC requiring 139 rounds. Overall, DD is able to reduce failure inducing programs in polynomial time with respect to size of the input.

Effective Program Debloating via Reinforcement Learning

Since the introduction of Delta Debugging, researchers have proposed optimizations that will enable this algorithms to terminate faster by taking into account the characteristics of the target language such as program's grammar. In this paper, Hoe et. al. introduced reinforcement learning as a means to intelligently and efficiently interpret input code's characteristics and reduce it respectively [44]. In contrast to original Delta Debugging paper, this work introduces Chisel, a tool that applies software minimization techniques with the goal of attack surface reduction.

Definition of bloat: Given a high level specification of the required functionalities of an application, Chisel aims at removing unnecessary features. This configuration is defined as a set of test cases with inputs and desired outputs for the application after each execution.

Identification of bloat: By applying Delta Debugging and rerunning the test cases, Chisel searches for the minimal program that satisfies the test requirements.

Evaluation & metrics: Lines of code reduction as well as removed known vulnerabilities are reported.

The main contribution of Chisel is to intelligently drop chunks of code from the program rather than doing a blind binary search. When compared to C-Reduce and Perses, Chisel is able to debloat real world applications (Linux binaries) in a timely manner whereas the other algorithms would not terminate. Overall, for ten GNU CoreUtil binaries, Chisel was able to reduce 90% of lines of code while keeping the functionality similar to BusyBox, which is an optimized version of GNU CoreUtils with minimal options targetted for embedded systems.

Chisel comes with two inherent drawbacks. First, by randomly removing parts of real applications, we will remove code that does not add to the functionality of the program, but is handling tasks such as error handling and security checks. For example, Chisel can remove a boundary check or a stack canary, as the target application would still function correctly without them. To address this issue, the authors relied on fuzzing and static code analysis techniques to pinpoint some of these security checks and prevent them from being removed.

Secondly, the generation of high level program specifications is a non-trivial task. In the example of "tar" archive software discussed in the paper, only a set of simple compression and decompression tests are performed. This approach can potentially open the door to new and hidden vulnerabilities, introduced after debloating.

Shredder: Breaking Exploits through API Specialization

Mishra et. al. presented an API specialization scheme for closed source binaries [58]. The main idea is to characterize how legitimate applications use sensitive APIs compared to exploit attempts. By generating policies that whitelist legitimate use of these APIs, they can break more than 86% of existing shell codes for 10 Windows applications in their dataset and stop all ROP exploits for 8/10 applications.

Definition of bloat: Instead of only focusing on unused code, they define policies to limit the set of parameters passed to the APIs.

Identification of bloat: Based on static analysis and backward propagation from sensitive APIs, the authors are able to find call sites and move backwards to extract possible values for parameters passed to these functions wherever possible. This approach is not able to identify values passed to every API call across the application due to limitations of static analysis. But this best effort approach can be coupled with control flow integrity to prevent calls to the sensitive APIs from unseen locations during an exploit attempt.

Evaluation & metrics: Current implementation of API specialization does not actually remove unused code, rather, by enforcing policies, it limits possible invocation of sensitive APIs. This method can be coupled with other debloating strategies to remove the corresponding code. As such, the authors report the number of broken shell codes before and after applying the policies.

API specialization tries to distinguish the different characteristics of benign and malicious execution of the program. The same idea can be applied to web applications, but static analysis is more limited on dynamic web programming languages such as PHP. Therefore, a combination of dynamic usage profiling and API specialization can be used as an extra step to further debloat a web application after source code debloating.

Chapter 3

Design & Methodology

Debloating has been applied to different parts of the software stack ranging from operating systems to containers and even executable binaries. Likewise, we also expect the web applications to benefit from this method of attack surface reduction. To quantify the extend to which various web applications are affected by debloating is the main research question of this study. The general architecture of web applications and their attack vectors are different than binaries. Based on this observation, debloating mechanisms that remove dead code and make exploitation harder by reducing the number of available gadgets only provide marginal benefit. By removing the actual vulnerabilities, we can cover a wider range of attacks.

Furtheremore, web applications are inherently more dynamic than compiled binaries. Decisions are made on the fly and its typical to dynamically load code at runtime. This makes static analysis even more complicated and less accurate. As a result, we rely on dynamic analysis and usage profiling to detect bloat. In the remainder of this section, we take a look at code reuse and external dependencies as the main source of bloat for web applications, and then describe the design and implementation details of our debloating pipeline.

3.1 Background

We briefly describe the effect of package managers on software bloat and provide a motivating example for debloating web applications.

3.1.1 Package managers and software bloat

To ease the development of software, developers reuse third-party libraries, external packages, and frameworks for their applications. This approach enables developers to focus on their applications while relying on proven and tested components. Statistics from popular package managers show that reliance on external packages is a widely adopted practice across many different languages. NPM, the registry hosting NodeJS packages, reports more than 10 billion package downloads a month [76]. Similarly, PyPI, the package manager for Python, reports more than a billion a month [32], while Packagist, the main repository for Composer package manager for PHP, reports the download of 500 million packages each month [31].

At the same time, it is doubtful that *all* the code and features obtained through these packages and frameworks are actually used by the applications that rely on them. For the most part, when developers rely on external dependencies, they include entire packages with no effective way of disabling and/or removing the parts of these packages and frameworks that their applications do not require.

3.1.2 Motivating web-application debloating

In this study, we look at the bloat of web applications and quantify how debloating can provide concrete security benefits. Even though debloating has been successfully applied in other contexts, we argue that the idiosyncrasies of the web platform (e.g. the ambient authority of cookies and the client/server model which is standard for the

web but atypical for operating systems and compiled software) require a dedicated analysis of the applicability of debloating for web applications.

To understand how the bloat of a web application can lead to a critical vulnerability, we use a recent vulnerability of the Symfony web framework (CVE-2018-14773 [30]) as a motivating example. Specifically, the Symfony web framework supported a legacy IIS header that could be abused to have Symfony return a different URL than the one in the request header, allowing the bypassing of web application firewalls and server-side access-control mechanisms. If this type of header was never used by the server, debloating the application would have removed support for it, which ultimately would have prevented anyone from exploiting the vulnerability. Drupal, a popular PHP Content Management System (CMS), was also affected by the same vulnerability since it uses libraries from the Symfony framework to handle parts of its internal logic [28]. Even if Drupal developers were not responsible for the code that leads to the vulnerability, their application could still be exploited since Symfony was an external dependency. Even more interestingly, an analysis of the official Symfony patch on GitHub [29] reveals that the vulnerable lines were derived from yet another framework called Zend [33]. This shows that the structure of web applications can be very complex with code reuse originating from many different sources. Even if developers take all possible precautions to minimize vulnerabilities in their own code, flaws from external dependencies can cascade and lead to a critical entry point for an attacker.

Overall, there are clear benefits that debloating could have on web applications. Assuming that we are able to pinpoint all the code that is required by the users of a given software deployment, all other code (including the code containing vulnerabilities) can be removed from that deployment.

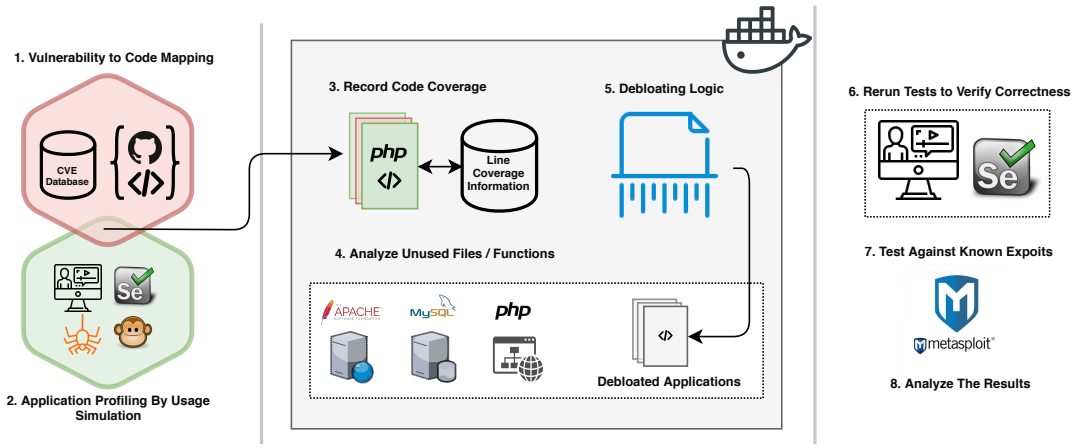


Figure 3.1: Overview of the architecture of our pipeline for debloating web applications and assessing the effects of different debloating strategies.

3.2 Setup

In this section, we describe the process of gathering information regarding known vulnerabilities (in the form of CVEs) for web applications, designing and executing tests against web applications of interest, and identifying the server-side code that was executed as a result of client-side actions.

3.2.1 Overview

The setup for our framework is depicted in Figure 3.1. To debloat target applications, we first collect information about the vulnerabilities of the applications that we analyze in our study. This information includes the files, functions, and line numbers where each vulnerability resides (Step 1, Section 3.2.3). Then, we simulate usage of the application through a combination of different techniques (Step 2, Section 3.2.4). Using a PHP profiler tool (`XDebug`), we record the lines, functions, and files, that are triggered during the simulation (Step 3, Section 3.2.5).

In the middle part of our pipeline, the debloating engine takes both the target applications and coverage information to perform debloating at different levels of granularity, and rewrite parts of the application to remove unused pieces of code based on

the debloating strategy being evaluated (Steps 4 and 5, Section 3.3). Our framework also provides a complete reporting panel to assist human analysts in understanding which vulnerabilities can be removed by the present debloating strategies.

Last, we verify the correctness of our debloating process by running a set of tests against the debloated web applications, and verifying that no removed piece of code is triggered (Step 5). To this end, we utilize assertions in place of the removed code blocks. An absence of error messages from these assertions means that all tests were successfully completed without triggering any missing server-side code. As an final step of verification, we also test the debloated applications against a series of exploits and verify that exploits which abuse any of the vulnerabilities that were removed as part of the debloating process, do not succeed (Step 6, Section 4.1.6).

To ease integration and facilitate the analysis of new web applications, we adopted a modular architecture that relies on three Docker containers. The *Application* container hosts our web applications. The profiler enabled on its web server is responsible for collecting code coverage information. The *Database* container runs a MySQL server that stores the code coverage information along with the databases of the tested applications. Lastly, the *Debloating* container which includes our debloating logic, analyzes the coverage information and generates debloated versions of applications. It also provides a reporting panel that indicates which vulnerabilities are removed in each application after debloating. To add a new vulnerability, a user simply has to provide the details of the vulnerable file(s) and line(s).

3.2.2 Analyzed web applications

To understand how the process of debloating increases the security of web applications, we decided against using toy-like web applications. Instead, we focused on established open-source applications with millions of users, and the presence of a sufficient number of known historical vulnerabilities (in the form of CVEs) to allow us

Table 3.1: Analyzed open-source web applications.

Web Application	**Version**	**Known CVEs** (≥2013)
Magento	1.9.0, 2.0.5	10
MediaWiki	1.19.1, 1.21.1, 1.24.0, 1.28.0	111
phpMyAdmin	4.0.0, 4.4.0, 4.6.0, 4.7.0	130
WordPress	3.9.0, 4.0, 4.2.3, 4.6, 4.7, 4.7.1	131

to generalize from them. To this end, we selected phpMyAdmin [62], MediaWiki [61], Magento [60], and WordPress [78], which are representative samples of four different types of web applications namely web-administration tools, wikis, online shops, and blogging software. Table 3.1 shows the versions of these web applications that we utilized, in order to map CVEs to the location of the vulnerability in the source code of each application.

3.2.3 Vulnerability to source-code mapping

To determine whether debloating web applications can actually remove vulnerabilities, we performed a mapping of known CVEs to the vulnerable lines, functions, and files, that they exploit in each application. This way, by looking at an application after debloating, we can determine if the files, functions, or lines responsible for the vulnerability, are still present or were removed during the debloating process.

Even though there exist multiple databases listing the current and historical CVEs of popular software (including the web applications in question) [38, 39], locating the actual source code containing the vulnerability described in a CVE, is a non-trivial process which requires careful investigation. In some cases, the right patch can be discovered because of a direct reference to a CVE in a commit message, or in a bug report on official public repositories of web applications. For others, the fix is included within numerous commits that have to be carefully analyzed to locate the appropriate lines of code. Since a vulnerability can span over multiple lines, functions, and even

multiple files, we record all affected locations in a database so that this information can be later correlated with each evaluated application.

Given the time-consuming nature of mapping CVEs to existing code, for this study, we limited ourselves to, at most, 20 CVEs per application of interest. The complete list of CVEs we mapped for this study can be found in Table 7 in the Appendix. To select these CVEs, we ordered existing vulnerabilities by their CVSS score (thereby selecting the ones that are the most critical) and we did not consider vulnerabilities that were reported before 2013. This focus on fairly recent vulnerabilities (i.e. in the last five years) makes our results more generalizable to the current state of web applications, as opposed to quantifying vulnerabilities in source-code which has since dramatically evolved. Note that, because not all versions of a web application are vulnerable to all evaluated CVEs, we had to map vulnerabilities across a number of different versions, as shown in Table 3.1.

3.2.4 Application usage profiling

Modern web applications provide an incredibly wide range of features and options to their users. Even though, from a functional perspective, more features are desirable, from a security perspective, the code that implements new features may contain new vulnerabilities thereby further expanding a program's attack surface. In order for a system to be able to remove code related to unnecessary features, one must first identify which features are necessary for a target set of users.

Given a usage profile, the goal of our framework is to produce debloated versions of web applications which maintain the code and features that are part of that profile but remove the rest. To be as objective as possible with what features are considered "necessary," we utilize four independent sources of web application usage: i) online tutorials describing how to use the applications of interest, ii) web crawlers that autonomously navigate the application, iii) vulnerability scanners that feed malicious

content to the application, and iv) monkey testing tools that click on random parts of webpages and type random keystrokes. The combination of all four gives our profiles both breadth (through the crawler and monkey testing) as well as depth (through the user following complicated paths while providing expected inputs and the vulnerability scanner which provides large amounts of malicious inputs trying to exploit the web application).

Tutorials

To simulate common interactions with an application, we use a popular search engine to search for the application's name followed by the word "tutorials" (e.g. "phpMyAdmin tutorials") and follow the tutorials from the first two pages of search results.

Specifically, we map each tutorial to a Selenium script that allows us to both execute the same tutorial multiple times and also assess the correctness of the results (e.g. encode that when we delete a database using phpMyAdmin, the deleted database is no-longer shown in the list of databases). Note that this mapping of tutorials to Selenium scripts is yet another time-consuming process which, occasionally, has to be repeated for different versions of the same web application. One change in a form field or in a selector can break the complete flow of a test suite and we observed a significant number of cases with slight interface changes between two consecutive versions of the same application.

Overall, after fine-tuning the scripts for all our tested versions, we obtained 46 tutorials which translated into 302 use cases scripted as Selenium tests requiring 16,025 lines of code. Given our desire for complete reproducibility of our results, we include the complete list of tutorials in the Appendix (Table **??**) along with WebArchive links that will remain available despite potential future domain expirations and linkrot of the original URLs [50].

Below, we provide a non-exhaustive list of actions that were part of the followed tutorials of each web application. Full details are available in the actual tutorials and in the Selenium scripts which we will release together with this paper.

Actions covered by phpMyAdmin tutorials: As a web administration tool, all phpMyAdmin functionality is protected by an authentication mechanism. We followed the actions described by tutorials when logged in as a root user account with full application access. The Selenium-encoded tutorials cover database operations including creating and dropping databases, filling tables with data, querying, table indexes, and importing/exporting data. They also include administration tasks such as adding new user accounts, optimizing databases, checking database server status, obtaining performance metrics, and accessing server settings such as variables, charsets, and engines.

Actions covered by MediaWiki tutorials: MediaWiki provides different features depending on the privileges of the user. Unauthenticated users can only visit and search pages. Registered ones can post and edit content while administrators can perform moderation and management operations. The tutorials that we followed cover all these different use cases. More specifically, actions coded in our tutorials include authentication, creating and renaming pages, importing and exporting content from the wiki, as well as changing settings such as skins, styles, and formatting options.

Actions covered by WordPress tutorials: As a blogging software, WordPress has two distinct entry points, one for normal unauthenticated users to read blogs and post comments, and a separate administration panel accessible to privileged and authenticated users. WordPress tutorials mostly focus on administrative tasks since normal users have limited abilities. The Selenium-encoded tutorials include actions such as creating a new post using HTML for the content, modifying most post options (ranging from visibility and tags to setting featured images), as well

as downloading and changing WordPress themes. For the administration panel, the tutorials include exporting content, setting up user accounts, and uploading media. Finally, the tutorials include the visiting of posts and the posting of comments as well as the management of comments, such as approving them, marking them as spam, and deleting them.

Actions covered by Magento tutorials: Magento is the largest evaluated web application in terms of source code and has the most features compared to the other applications. Similar to WordPress, the tutorials mostly target administration tasks which include store settings, advanced product search options, order notification via RSS, product pricing, currencies and tax rules, delivery and payment methods, emails and notifications, reviews and ratings and cache control. Some tutorials go in even more details by covering product and stock management, managing customers and groups configurations, modifying the UI, creating pages, and using widgets. On the customer side, we followed tutorials that included registration of a new account, authentication actions, and purchasing products until checkout.

Monkey testing

Monkey testing is a method for testing software where the simulated user sends random clicks and keystrokes to the target application. This unpredictable behavior can uncover bugs in an application as it can trigger paths and actions that were not anticipated by developers. In our case, we use such a technique to trigger additional code, not covered by tutorials. We observe that this approach adds breadth to the code coverage by reaching easy to access features. In addition, by feeding random key strokes into forms, monkey testing can bring the application in an error state thus exercising error-handling pieces of code.

We rely on the stress-testing library called `gremlins.js` [8] in conjunction with the GreaseMonkey browser extension [7] to inject the library into web application pages.

Since this kind of testing can occasionally trigger unwanted actions, we have to take necessary steps to stop them, e.g., prevent the test from leaving the web application and visiting external websites. We also want to prevent `gremlins.js` from getting trapped on a single page as an unexpected JavaScript dialogue box or a dead end page can pause our test execution. An additional issue is that of accidentally logging out a web application by clicking on a logout link. Given that we run monkey-testing under three different usage profiles (public user, logged-in user, and administrator) we took steps to avoid accidental logouts. Overall, we perform the following modifications: i) we remove all links that lead to external pages, ii) we remove logout buttons for applications that require authentication, iii) we override the aforementioned JavaScript functions and iv) we set a timeout to detect when the monkey is stuck and reset it to a known good state. All these actions are done using injected JavaScript on target pages prior to starting the `gremlins.js` library.

To cover a large set of pages from a web application, we run `gremlins.js` for 12 hours for each of the test profiles. To guarantee the reproducibility of our experiment, we choose a fixed seed for each run that will generate the same sequence of pseudo-random actions.

Crawling

Web spiders (also known as crawlers) are a type of bot that follows the links of a web application and optionally submits forms with predefined content. Each newly crawled page is added to a database of the application that the crawler uses to prevent repeated visits to the same pages. For our study, we use BurpSuite Spider v2.0.14beta [3] to crawl our web applications. As a result, we augment the applica-

tion coverage with code paths that were not triggered, either through the followed tutorials or through monkey testing.

Running vulnerability scanners

Vulnerability scanners are tools that try to detect security flaws in web applications. We use BurpSuite Scanner v2.0.14beta [3] based on the URLs extracted by the spider to look for vulnerabilities in headers, URLs and forms. Notably, the scanner tries different injection mechanisms like SQL injection, XSS, PHP file injection, and path traversal, to trigger errors and reach unwanted states in the application. The vulnerability scanner goes beyond what the crawler and the monkey cover by modifying headers and URL parameters. By inspecting the resulting coverage, we observe that each of these four methods result in exercising server-side code that would not have been exercised through the other methods. We quantify this relationship in Section 4.1.

3.2.5 Recording server-side code coverage

Regardless of the method that is used to interact with a web application, in order to be able to successfully remove unused code (i.e. debloat the web application), we must be able to associate client-side requests with server-side code. To record the files and lines of code that are triggered by user requests, we make use of PHP profilers.

PHP profilers are available as PHP extensions that modify the PHP engine to collect code-coverage information. There exist a number of different profilers, such as, `XDebug` [25], `phpdbg` [18], and `xhprof` [26] all of which require a similar setup to record code coverage. For our framework, we decided to use `XDebug` as it is the most mature profiler and is actively maintained.

Adding coverage support in a web application

Connecting a web application to XDebug. To be able to perform dynamic analysis and record lines of code that are triggered by user requests, our framework must add calls to specific `XDebug` functions in every PHP file of a web application. Specifically, both `xdebug_start_code_coverage()` and `xdebug_get_code_coverage()` functions are called to, respectively, start and receive coverage information. If the "get" function is never called, the coverage information is lost. In the following paragraphs, we describe challenges related to obtaining the code coverage from `XDebug` and how we overcame them.

The case of unrecorded lines. Boomsma and Gross reported on the possibility of removing unused code in a custom PHP application [36]. By performing dynamic analysis, they observed which files were not used and removed them from their application. The authors utilized their own profiler and took advantage of the `auto_append` built-in function of PHP to add the necessary log functions at the very end of all PHP files [1].

For our study, we initially attempted to use the same approach and ran preliminary tests by appending `XDebug` function calls at the end of our tested files. However, we discovered that the coverage was incomplete and that some lines were not properly recorded. Given that any PHP file can call the *exit()* or *die()* function at any time to terminate the current script, our `XDebug` calls which were located at the end of each file, were not always executed thus leading to under-reported code coverage.

Main challenges for getting full coverage

Avoiding early exits. To overcome the coverage problems due to calls to exit functions, we utilized a specific type of PHP callback functions, called *shutdown* functions. When registered, these functions are triggered after all the code on the

page has finished running or after either *exit()* or *die()* functions are called. This way, we are able to obtain the desired coverage information even if a PHP script used one of the aforementioned functions. Interestingly, we also discovered that calls to *exit()* inside a shutdown function prevent the execution of other shutdown functions including the call to collect our own code-coverage information. To correct this issue, we statically analyzed the evaluated applications and automatically added calls to collect code coverage before these exit calls (e.g. Line 7 in Listing 3.1).

Getting correct coverage information of shutdown functions. Another challenge, in terms of recording correct code-coverage information, is to properly record the executed lines of code inside shutdown functions. As mentioned by the PHP manual [14], shutdown functions are called in the order they were registered. This means that if our own shutdown function is registered first, it will also be triggered first, thereby missing any calls to subsequent shutdown functions present in the same PHP file. To get full coverage, we use the following approach: our own shutdown function will perform a late registration of a final shutdown function that will be added at the very end of the execution queue. This way, we can be certain that the very last shutdown function that will be executed in a script will be our own, providing us with the desired coverage information.

Getting correct coverage information of destructors. The final challenge that we faced was to properly record covered lines for all class destructors. PHP uses garbage collection and reference counting to remove objects from memory, whenever they are no longer necessary. However, there is no real way to anticipate when the garbage collector will effectively remove objects during program execution. If objects are destroyed *before* the shutdown functions are executed, our framework has no issue recording them. However, if they are destroyed after, our shutdown functions are incapable of registering the execution of these destructors.

```
1  <?php
2  register_shutdown_function("PMA_Response::resp");
3  class PMA_Response {
4    public static function resp() {
5      $buffer->flush();
6      // Prepend original call to exit:
7      collect_code_coverage();
8      exit;
9    }
10 }
11 class TCPDF {
12   public function __destruct() {
13     // If called after shutdown_functions
14     // start recording code coverage
15     ...
16     // If called after shutdown_functions
17     // stop coverage
18   }
19 }
20 ?>
```

Listing 3.1: Code rewritten by the debloating framework to ensure correct code coverage of corner cases.

To handle this special case, we rewrote class destructors so that they register themselves while they are executing. Every time a destructor is called, we query the `XDebug` engine to check whether code-coverage recording is currently in progress. This way, we can determine whether the destructor is called before or after shutdown functions. If the destructor is called after shutdown functions, we dynamically decide to start recording all executed lines within the destructor and save the coverage information when it finishes executing.

Summary. As witnessed through the above use cases, collecting the correct code coverage information for a web application is significantly more complicated than one would initially expect. Through the preprocessing of code, and the use of destructors and shutdown functions, we solve the issues that were not even mentioned in prior work and get a precise view of the code that executes at the server side, as a result of user requests. Listing 3.1 provides an example of concrete modifications in a PHP file. On line 7, we added a code-coverage call before an `exit` which happens inside a shutdown functions to prevent information loss due to early exits. On lines 14 and 17, we wrapped the destructor with code-coverage calls.

3.3 Debloating web applications

In this section, we briefly describe the evaluated debloating strategies and the steps we took to ensure that the debloated applications remain functional.

3.3.1 Debloating strategies

By combining the simulated usage of a web application (achieved through tutorials encoded in Selenium scripts, web crawlers, monkey testing, and vulnerability scanning) with server-side code profiling, we can identify the code that was executed as part of handling web requests. Consequently, code whose execution was not triggered by any client-side request can presumably be removed since it is not necessary for any of the functionality that is desired by users (as quantified by the utilized usage profiles). In this work, we evaluate the following debloating strategies:

- **File-level debloating:** Given that the source code of web applications spans tens or hundreds of different files, we can completely remove a file, when none of the lines of code in that file were executed during the stimulation of the web application.

- **Function-level debloating:** In function-level debloating, not only can we remove entire files but we can also selectively remove some of the functions contained in other files. This is a more fine-grained approach which allows us to remove more code, than the more conservative, file-level debloating strategy.

More fine-grained approaches are possible, such as, the removal of specific code statements from retained functions which were not exercised during stimulation. However, such changes essentially modify the logic of a function (e.g. removing conditional code blocks) thereby increasing the probability of breaking the resulting program when a minute change of a client-side request would lead the execution into these blocks of code.

3.3.2 Detecting the execution of removed code

We replace all removed functions and files with placeholders which, if executed, have the following tasks:

- **Exit the application:** If a placeholder happens to be triggered, the PHP application will start its shutdown procedures. This way, the application does not enter an unexpected state that was not planned by the debloating process.
- **Record information about the missing function:** In order to better understand which missing placeholders were triggered and how, our framework logs several pieces of information, such as, the URL that triggered the execution of the removed code, the name of the class and function of the removed code, and the corresponding line numbers.

To ensure that the debloating process has preserved the functionality of the debloated web application, we rerun all the Selenium-mapped tutorials and monkey scripts after the debloating stage. If our placeholder code for removed files and functions executes during this stage, this means that this code should not have been removed.

This feedback mechanism proved invaluable during the development of our framework since it helped us identify problems with our coverage logic which in turn revealed the challenges that we described in Section 3.2.5.

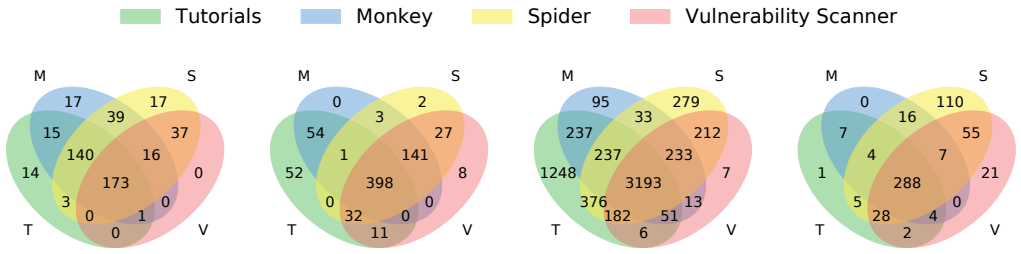
Chapter 4

Results & Conclusion

In this chapter, we present the results of our study.

4.1 Results

To assess the impact of debloating web applications, we analyze our results from a number of different perspectives. First, we show the contributions of different application-profiling methods and then compute different metrics to understand the effectiveness of debloating in terms of reducing the attack surface of our tested applications. Next, we focus on CVEs to determine whether debloating can actually remove critical vulnerabilities. Then, we take a closer look at the bloat introduced by external packages along with the security implications that come with using this specific development practice. Finally, we look at what has effectively been removed in debloated applications and test a number of exploits against the original and debloated versions of the evaluated web applications.



(a) phpMyAdmin 4.7.0 (b) MediaWiki 1.28.0 (c) Magento 2.0.5 (d) WordPress 4.7.1

Figure 4.1: Venn Diagrams showing covered files during the execution of Tutorials, Crawler, Monkey testing and Vulnerability scanner

4.1.1 Tutorials vs. Monkey Testing vs. Crawling vs. Vulnerability Scanning

As described in Section 3.2.4, to ensure that we exercise web applications in an objective and repeatable way, we utilized tutorials, monkey testing, crawlers, and vulnerability scanners. Figure 4.1 shows the coverage, in terms of server-side files, that each method obtained on the latest version of each web application in our testbed. We can clearly see that all four methods are required, with each method contributing differently for different web applications. For example, tutorials trigger more files in Magento compared to other applications, while Spider covers most unique files in WordPress.

4.1.2 Debloating by the numbers

To evaluate the effectiveness of our two debloating strategies, we computed different metrics that provide insights into what has actually been removed during the debloating process.

Logical lines of code

The size of a program positively correlates with the number of programming errors (i.e. bugs). According to McConnel [57], the industry average, at least in 2004, was to have between 1 and 25 bugs for every one thousands lines of code. Given the importance of the size of an application to its overall security, we start by estimating the reduction of the attack surface by looking at the Logical Lines Of Code (LLOC, sometimes also called Effective Lines Of Code). LLOC is intended to measure lines of code without comments, empty lines and syntactic structure required by the programming language. LLOC reduction is a robust and precise indicator of how much the volume of the code was reduced. Figure 4.2 reports on the LLOC for all versions of the applications we debloated.

Number of logical lines over time. Looking at the number of LLOC of the original applications, we can observe two different evolution behaviors. For WordPress, the amount of code is stable and there is even a small decrease of 2% of LLOC between versions 4.7 and 4.7.1. For the other applications, we observe the opposite where the source code in the latest versions spikes, compared to the ones released just before them: 82% LLOC increase for phpMyAdmin, 99% for MediaWiki, and 171% for Magento. By analyzing the code of these newer versions in an attempt to understand their sudden expansion in size, we discovered that these spikes can be attributed to a change in development practices, namely the reliance on external packages. As WordPress does not rely on external packages, it does not exhibit this kind of behavior. We discuss the issue of relying on external packages in more detail in Section 4.1.4.

File-level debloating. Overall, file-level debloating, the most conservative of the two evaluated debloating strategies, is already effective in reducing the number of

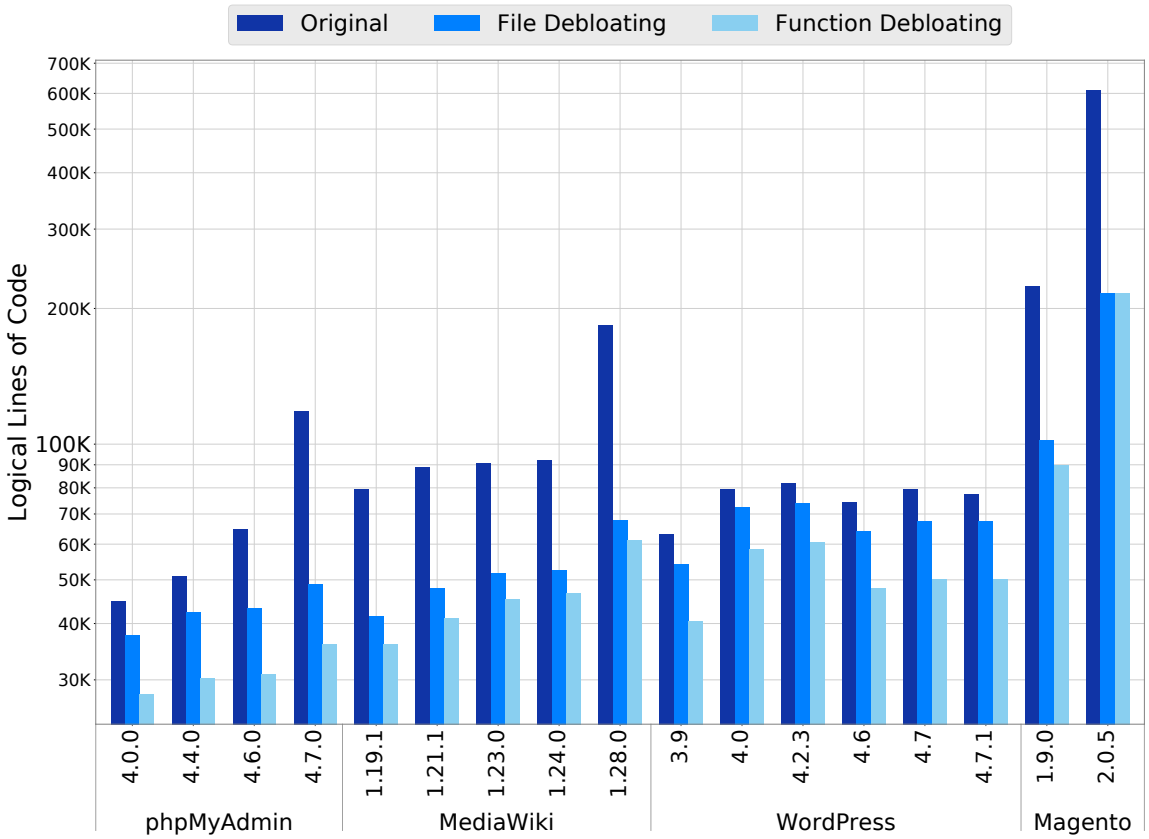


Figure 4.2: Logical Lines of Code before and after debloating

LLOC with an average of 33.1% reduction. The minimum observed in our experiment is 9.2% for WordPress v.4.0 and a maximum of 64.5% for Magento v.2.0.5. For Magento, this reduction represents a removal of 393K lines of code. This number is a clear sign that large web applications encompass many different features that may not be used by all users and therefore result in bloated applications with an unnecessarily large attack surface. At the same time, it is worthwhile repeating that all debloating results presented in this section are conditional to how web applications are used. Therefore, these large levels of debloating cannot be guaranteed for all possible deployments of web applications. We discuss this issue in Section 4.3.

Function-level debloating. On average, function-level debloating is able to remove 46.8% of lines of code. For both Magento and MediaWiki, it can remove up

to 7% more code over file-level debloating. For phpMyAdmin and WordPress, we observe an increase of debloating capability of up to 24%. This larger reduction (compared to MediaWiki and Magento) is mainly due to the differences in software development practices.

Compared to the other tested applications, phpMyAdmin and WordPress are more monolithic with a smaller number of large source-code files. Since file-level debloating only removes files when none of their functions were executed, the monolithic nature of these two applications resists this kind of coarse-level debloating. Contrastingly, Magento and MediaWiki are developed in a much more modular fashion (many small files each responsible for a small number of well-defined tasks) and therefore lend themselves better to file-level debloating. The more fine-grained, function-level debloating bypasses this issue and can therefore reduce the attack surface of a web application, even for more monolithic web applications.

Cyclomatic complexity

Next, we look at the evolution of cyclomatic complexity (CC). CC is defined as the number of linearly independent paths through the code of an application [56]. A high CC for a single class implies complicated code that is difficult to debug and maintain [41] and therefore more prone to contain vulnerabilities when compared to code with low CC [71, 54].

Figure 4.3 reports on the evolution of the overall CC for each tested version in our experiment. File-level debloating decreases CC between 5.9% to 74.3% with an average of 32.5%. Function-level debloating decreases the program complexity between 23.8% and 80.2% with an average of 50.3%. These statistics demonstrate that debloating can remove complex instructions and execution paths in addition to simple ones. Moreover, the difference between file-level and function-level debloating shows that code removal through function-level debloating is much more suited to

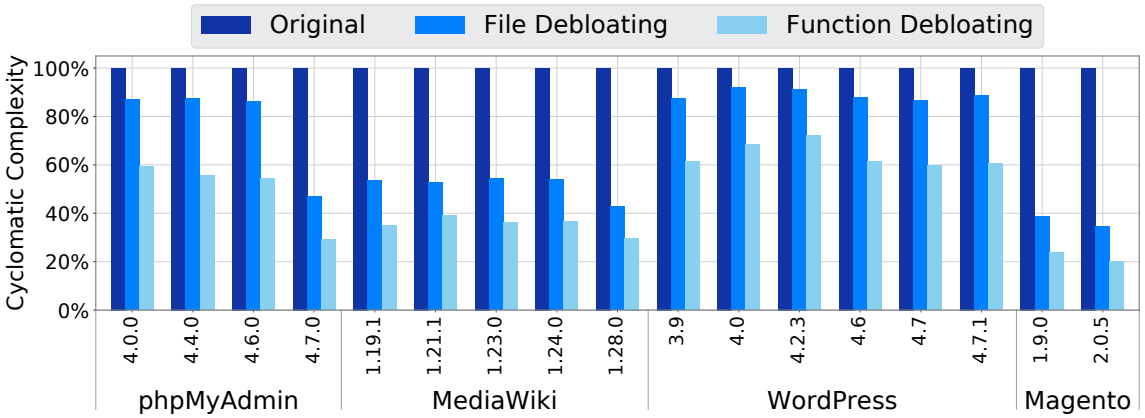


Figure 4.3: Evolution of cyclomatic complexity before and after debloating

all kinds of web applications as shown earlier through LLOC reduction achieved via function-level debloating.

4.1.3 Analysis of CVEs

In this section, we investigate the number of removed CVEs after debloating along with the effects of debloating on different vulnerability categories.

CVE reduction after debloating

One practical way to measure the security benefits of debloating web applications is to study the effects of debloating on known historical vulnerabilities. If vulnerabilities were part of the core functionality of the program, the evaluated debloating strategies will not be able to remove the code associated with them. However, if some vulnerabilities reside in parts of a web application that are not commonly used, the process of debloating can effectively remove them.

Table 4.1 compares the effectiveness of debloating strategies by listing the fractions of removed CVEs. We consider a vulnerability to have been successfully removed if all the lines of code and functions associated with that vulnerability were removed during the stage of debloating. This is a conservative approach as one modification

Table 4.1: Number of CVEs removed after application debloating

Application	**Strategy**	**Total Removed CVEs**		**Removed Exploitable CVEs**	
phpMyAdmin	File Debloating	4/20	20 %	3/19	15.7 %
	Function Debloating	12/20	60 %	11/19	57.8 %
MediaWiki	File Debloating	8/21	38 %	3/16	18.7 %
	Function Debloating	10/21	47.6 %	5/16	31.2 %
WordPress	File Debloating	0/20	0 %	0/20	0 %
	Function Debloating	2/20	10 %	2/20	10 %
Magento	File Debloating	1/8	12.5 %	1/8	12.5 %
	Function Debloating	3/8	37.5 %	3/8	37.5 %

performed on a single line could thwart a complete attack. As such, the numbers we report in this section can be interpreted as lower bounds of the actual number of removed CVEs.

In terms of configuration, we selected the default one for each application. However, certain vulnerabilities may not be exploitable under this configuration. For example, there exists 5 CVEs in our dataset for MediaWiki which require file upload functionality to be enabled. Since this option is disabled by default, we make an explicit distinction in the table. "Total Removed CVEs" is the total number of CVEs removed by debloating regardless of whether the vulnerable code is enabled or disabled through a configuration option. "Removed Exploitable CVEs" reports on the CVEs that are reachable under default configurations of target web applications.

On average, we discovered that up to 38 % of vulnerabilities are removed by file debloating whereas 10-60 % are removed by function debloating. As shown in Table 4.1, function-level debloating can triple (in the case of phpMyAdmin and Magento) the number of removed CVEs, compared to file-level debloating. This behavior can be generalized to web applications that do not have CVE information and demonstrates that the reduction of a web application's LLOC (Section 4.1.2) and its cyclomatic complexity (Section 4.1.2) translates to a reduction of concrete vulnerabilities. Wordpress is a clear negative outlier with only 10% CVE reduction, even through the more

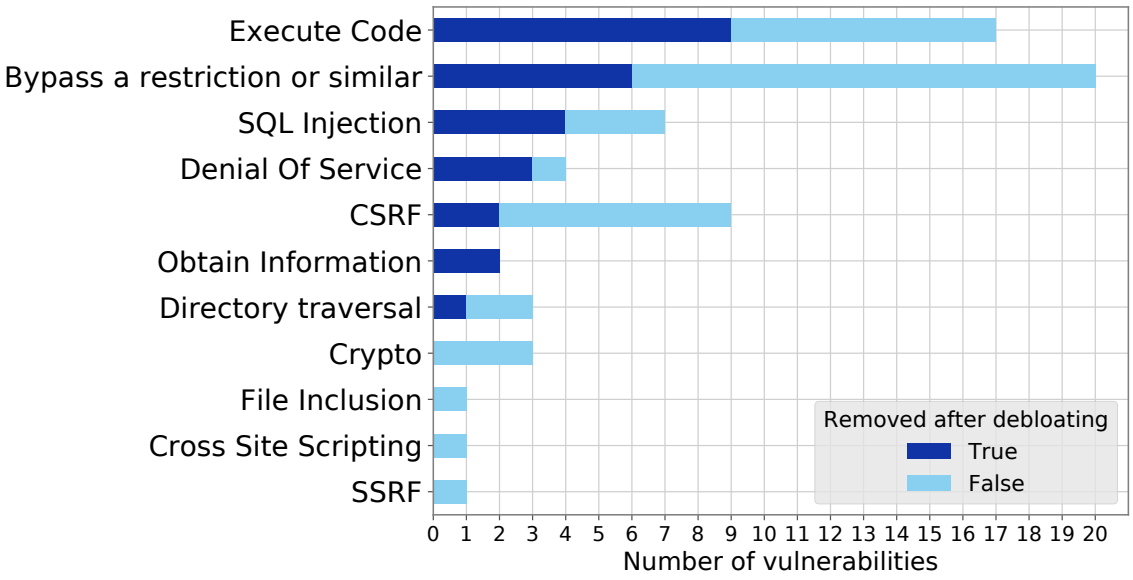


Figure 4.4: Vulnerability Categories

flexible function-debloating strategy. As mentioned earlier, WordPress is a relatively monolithic application and most of our mapped CVEs are located in core WordPress code (e.g., Authentication, CSRF tokens, and post/comment-related actions) which cannot be removed by our debloating framework.

Types of CVEs in analyzed web applications

Even though our results demonstrate the ability to remove vulnerabilities from web applications through the use of debloating, one may wonder whether debloating is better suited for some types of vulnerabilities over others. Figure 4.4 provides details on the categories of the CVEs we removed through debloating.

One can observe that for certain classes of vulnerabilities, such as, Denial-of-Service attacks and Information-Revealing vulnerabilities, debloating can almost completely remove them. For others, such as, restriction bypassing, command execution, and SQL injection, debloating can substantially reduce them. Our interpretation of these findings has to do with the maturity of the evaluated web applications. Specifically, all four web applications have been available for a long period of time, allowing

Table 4.2: Statistics on the external packages included in web applications and the effects of debloating in terms of reducing their LLOC.

Application	Before debloating			After function-level debloating					
	# lines in main application	*# lines in packages*	*# packages*	*# lines in main application*	*# lines in packages*	*# packages completely removed*	*# packages where a given % lines were removed*		
							> 70%	*< 70% and > 30%*	*< 30%*
phpMyAdmin 4.7.0	35,739	82,604	45	26,377 (-26.2%)	9,653 (-88.3%)	38 (84.4%)	2	1	4
MediaWiki 1.28.0	133,019	50,898	40	54,827 (-58.8%)	6,285 (-87.7%)	24 (60.0%)	2	2	12
Magento 2.0.5	396,448	212,906	71	181,696 (-54.2%)	34,038 (-84.0%)	58 (81.7%)	6	5	2

many shallow vulnerabilities to have already been discovered and corrected. The remaining vulnerabilities are likely to be situated in parts of a web application that are less commonly exercised. For example, the code-execution vulnerabilities that can be removed for phpMyAdmin are inside very specific features, such as, the ability to export PHP arrays (CVE-2016-6609), the support of the ZIP extension while importing data (CVE-2016-6633), and the abilities to copy table definitions (CVE-2013-3238) and perform Regex search and replace over table columns (CVE-2016-5734).

Contrastingly, the three cryptography-related vulnerabilities we analyzed are still present in the debloated versions of web applications. One of the CVEs related to this category is about a flaw in the cookie encryption algorithm in phpMyAdmin (CVE-2016-6606). Since every page interacts with user cookies to, at the very least, verify them, vulnerable code cannot be removed. Another vulnerability in this category relates to an insecure random number generator used in cryptographic operations by Magento (CVE-2016-6485). This vulnerability exists in a constructor of the main encryption classes which is widely used throughout the application. When considered together, these findings suggest that cryptography-related vulnerabilities are a core part of web applications and thus unlikely to be removed through the process of debloating.

4.1.4 External packages

Quantifying the bloat from external packages

In our testbed, phpMyAdmin v.4.7.0, MediaWiki v.1.28.0 and Magento v.2.0.5 rely on external dependencies that can be downloaded via Composer (WordPress does not rely on external packages). As described in Section **??**, Composer is a package manager for PHP (similar to the NPM manager for NodeJS applications) which allows web applications to specify which external packages they rely on and have these packages be tracked and updated.

As we briefly discussed in Section 4.1.2, the number of LLOC of these three specific versions dramatically increases (compared to prior versions) because of this dependency on external packages. Table 4.2 provides statistics on the number of packages pulled by these applications and how much bloat they provide against our usage profiles.

First, one can observe that external packages introduce a large amount of unused code. For all three debloated applications, more than 84% of their code was removed from them. This means that the attack surface is unnecessarily large through the dependency on external packages. The number of removed lines from external packages for Magento is particularly noteworthy with more than 178,000 lines of code removed. Moreover, the number of packages that can be completely removed is also quite large: 84% for phpMyAdmin, 60% for MediaWiki and 81% for Magento. This confirms that most packages are unnecessary for the usage profiles that we recorded. Finally, focusing exclusively on the lines of code, phpMyAdmin is the only application where external packages have more lines than the main application. However, after debloating, this relationship is reversed with the codebase of phpMyAdmin being three times the size of the introduced external packages.

Despite the advantages of using package managers (e.g. the ability to track dependencies and update vulnerable libraries without the need to update the main application), our findings show that these advantages come at a considerable cost in terms of unnecessarily expanding the attack surface of a web application with code that is seldomly executed. As such, developers must take special care to include the bare minimum of external packages, knowing the unwanted side-effects that each external package brings.

Removing POI gadgets

What are POI gadgets? Property Oriented Programing (POP) is an exploitation technique in PHP which works similarly to Return Oriented Programming (ROP) [69] and is used to exploit PHP Object Injection (POI) vulnerabilities [13]. In this technique, the attacker creates exploit gadgets from available code in the applications. By chaining multiple gadgets within the application, an attacker can usually run arbitrary code, write to arbitrary files, or interact with a database. Dahse et al. have studied the automatic generation of such gadget chains for PHP applications [40].

PHP unsafe deserialization. The PHP language gives developers the ability to serialize arbitrary objects in order to store them as text, or transfer them over the network. Deserialization reverses this process, generating PHP objects from serialized data. This mechanism can be abused by an attacker to load specific classes in the application and build a gadget chain. Practical examples of this vulnerability are when `unserialize` is called on a database field or value of a field within a cookie that can be manipulated by the users.

Historically, this attack was very difficult to successfully execute. Attackers could only build gadgets with the classes that were present in the context of the vulnerable file. They needed insights into how the application was built in order to know which

classes could be abused for gadgets. However, starting from PHP 5, the `__autoload()` magic function [12] was introduced and unintentionally made exploitation of deserialization vulnerabilities easier. This new loading feature was beneficial for PHP developers who did not have to manually include all the files they wanted to use at the very top of each of their PHP files. It also helped the adoption of package managers like Composer, as any external dependency could be easily called from anywhere in the application. The downside of this new function was that it also allowed attackers to instantiate any PHP class across the entire application thereby enabling the easier construction of gadget chains.

In order to build a chain, attackers use these so-called "magic" functions [15] that form the basis of their gadget chain. One of the functions that is widely used in POI exploits is the *destruct* function. In Section 3.2.5, we detailed the challenges in getting complete coverage of destructors in our tested applications. Accurate coverage of destructors also allows us to precisely analyze the impact of debloating on gadget creation.

Can debloating remove gadgets from external packages? Given the increased footprint of web applications due to their reliance on package managers and external dependencies, one may wonder about the possibility of abuse of these packages for the creation of gadgets. To measure the effect of debloating on Property-Oriented-Programming (POP) gadgets, we utilized the PHPGGC [19] library. PHPGGC (which stands for PHP Generic Gadget Chains) contains a list of known gadgets in popular PHP packages such as Doctrine, Symfony, Laravel, Yii and ZendFramework. When a vulnerable PHP application includes any of the packages listed in PHPGGC, the attackers can generate gadget chains to achieve RCE, arbitrary file writes, and SQL injections.

Table 4.3: List of packages with known POP gadget chains

Application	**Package**	**Removed by Debloating**	
		File	*Function*
phpMyAdmin 4.7.0	Doctrine	✔	✔
	Guzzle	✔	✔
MediaWiki 1.28.0	Monolog	✔	✔
Magento 2.0.5	Doctrine	✔	✔
	Monolog	✖	✔
	Zendframework	✖	✔

We analyzed the available gadget chains in PHPGGC and checked whether any of our tested PHP applications included these chains. Table 4.3 summarizes the presence of each gadget and whether debloating removes them or not. WordPress is not included in this table because it does not rely on external packages. This does not make WordPress immune to POI attacks, but universally known gadget chains in popular external packages can not be used to exploit WordPress. For the affected applications, file-level debloating removes 4/6 gadgets while function debloating removes 6/6 available gadget chains. This again demonstrates the power of debloating which can not only remove some fraction of vulnerabilities but also make the exploitation of the remaining ones harder by removing the gadgets that attackers could abuse during a POI attack.

Utilizing development packages in production

During our analysis of external packages, we identified yet another source of bloat in new versions of web applications. When declaring external dependencies through Composer, two options are available: "require" and "require-dev". The first option indicates packages that are mandatory for the application to run properly. The second lists packages that should only be used in development environments, such as, packages providing support for unit testing, performance analysis, and profiling. We discovered that applications downloaded from official websites often include these

development packages. As such, when these packages are used to deploy web applications in production mode, they will contain unnecessary development libraries. This does not only increase the attack surface by having unnecessary code bloating the application, but can also lead to exploitation for misconfigured applications.

CVE-2017-9841 presents one example of such a vulnerability [27]. Specifically, this CVE refers to an RCE attack in specific versions of the PHPUnit library, which is a popular unit testing library for PHP. By default, Composer places all external packages under "vendor" directory. If this specific directory happens to be accessible through a misconfiguration of the server, PHPUnit files are then accessible and can be exploited to conduct an RCE attack.

The four web applications that we evaluated for this study, present different behaviors with respect to development packages. WordPress does not rely on external packages downloaded through Composer. MediaWiki never included development packages in its releases. phpMyAdmin had them in version 4.7.0 but stopped including them in version 4.8.3 (the latest at the time of writing). Magento started including them from version 2.0 and still includes them today. We have reached out to Magento and informed them about this issue.

4.1.5 Qualitative analysis of the removed code

In the previous sections, we analyzed the effects of debloating on the source code of applications from a software-engineering perspective (i.e. LLOC and Cyclomatic Complexity reduction) as well as from a security standpoint (i.e. number of CVEs and gadgets removed). At the same time, one may wonder what exactly was removed from each application during the process of debloating.

Given that thousands of files were removed, manually analyzing each file does not scale. As such, we turn to NLP techniques that allow us to cluster the removed files together and provide us with hints about the nature of each cluster. Specifically, we

use the k-means clustering algorithm based on text vectors extracted from removed file names and file paths. Each file path includes directories that indicate which library or package, the file belongs to. For most modern web applications, this allows for a reasonable separation of files across different application plugins and modules. To end up with meaningful clusters, we tuned TFIDF vectorizer parameters along with the number of k-means clusters. We used the TFIDF maximum frequency limit to ignore common terms appearing in more than 50% of the files. Depending on the size and modularity of the application, 10 to 20 clusters yielded the most instructive grouping of files.

Table 4.4 shows the categories of the three largest removed clusters from each web application. Across all four applications, we observe the removal of source code related to external packages (e.g. Symfony for phpMyAdmin, Elastica for MediaWiki, and Zendframework1 for Magento), followed by localization/theme files (e.g. twentyfourteen theme for WordPress), and unused database drivers. We provide more application-specific details of removed features in the next paragraphs.

phpMyAdmin's removed features include the uploading of plugins, GIS visualizations, and unused file formats used in import/export (such as, Dia, EPS, PDF, SVG, and ZIP). In addition, debloating removed unused plugins and external packages which make up the top 3 features removed from this web application as shown in Table 4.4. phpMyAdmin version 4.6.0 and 4.7.0 include unit tests which are also removed by our system. The LLOC for the removed test files is less than 2% of the whole code base of the application.

MediaWiki provides an API to interact with the wiki which is separate from the regular web interface that users interact with. Most actions within this API, including queries, file upload, and non-default output formats for this API were removed. Top categories of removed files consist of localization of messages and language files in addition to external dependencies (Lines 2 and 3) as listed in Table 4.4. The de-

Table 4.4: Features and external packages with the most removed files after file debloating (removed features are marked in italic). Entries marked with $*$ are packages that are indirectly pulled by other "require-dev" packages (not used by core application) for the purpose of test coverage reporting and coding standard enforcement.

Applications	**Features/Packages with most files removed**
phpMyAdmin 4.7.0	1) Guzzle [9]: "Generating API HTTP response" * 2) Symfony [22]: "Parsing configuration files" * 3) PHP_CodeSniffer [17]: "Enforcing coding standards" *
MediaWiki 1.28.0	1) *Messages & Languages* 2) Less.php [10]: "Generating CSS code" 3) Elastica [6]: "Elastic search interface used by extensions"
WordPress 4.7.1	1) Twentyfourteen theme [23] 2) Twentytwelve theme [24] 3-4) Also theme related 5) *Multi-site administration*
Magento 2.0.5	1) Zendframework1 [16]: "Generating web pages and database operations" 2) *Sales, Orders & Credit Memo* 3) *Internal framework filters & Views*

bloating process also removes file-upload modules which are disabled, by default, in MediaWiki. It is important to note that even if a module is "disabled," the code still resides on the server and could be abused by specific types of attacks. For example, in a recent attack against a WordPress plugin, the vulnerability could be exploited even if that plugin was disabled [34]. Debloating *removes* the source code of disabled and unused features and therefore does not suffer from this type of attack. Finally, the process of debloating, removed unused extensions of Mediawiki (e.g. citation, input box, pdf handler, poem and syntax highlighting). Mediawiki 1.19.1 and 1.28.0 include unit tests, and they measure less than 1.5% of LLOC in the whole code base of their respective versions.

WordPress takes a slightly different approach where the core functionality is concentrated in a relatively small number of large PHP files. The removed features of WordPress include installation files, unused modules (FTP, multi-site, user registration), disabled themes and update files (note that we could not exercise update files

during our tests because this would change the version of the evaluated web application and create inconsistencies in our analysis of removed CVEs). In terms of testing, the installation files that we obtained from the WordPress website do not contain any unit tests.

Magento consists of both external packages and internal modules. We observed that various internal modules were removed, including an XML API for mobile, wishlists, ratings, and specific payment modules (such as, Paypal). Since many packages and internal modules include the terms "sales," "orders," and "tax," these individual files across multiple modules were clustered into the same category by k-means. Finally, Magento 1.9.0 does not include unit tests while the test files included in Magento 2.0.5 and its external packages measure up to 15% of its code base. For Magento 2.0.5, Zendframework1 which is an external dependency has most of its files removed by debloating.

4.1.6 Testing debloated web applications against real exploits

To ensure the correct mapping of CVEs to source code and the ability of debloating to stop real attacks, we collected 4 exploits available in the Metasploit framework and augmented them with 4 POCs that we developed based on public bug-tracker records and vulnerability details. After verifying that we can successfully exploit the original versions of the evaluated web applications, we tested the same exploits on the debloated versions. Half of the previously successful exploits failed because the vulnerable code was removed during the process of debloating. Table 4.5 lists the tested exploits against original and debloated applications.

As before, this demonstrates that while debloating is not a panacea against all possible issues, it can substantially improve the security of web applications. Finally, we present a demonstration of CVE-2016-4010 on Magento 2.0.5 in the following video: `https://vimeo.com/328225679`.

Table 4.5: Verifying exploitability of vulnerabilities by testing exploits against original & debloated web applications

CVE	Target Software	Exploit Successful?	
		Original	Debloated
CVE-2013-3238	phpMyAdmin 4.0.0	✔	✔
CVE-2016-5734	phpMyAdmin 4.4.0	✔	✖
CVE-2014-1610	MediaWiki 1.21.1	✔	✔
CVE-2017-0362	MediaWiki 1.28.0	✔	✖
CVE-2018-20714	WordPress 3.9	✔	✔
CVE-2015-5731	WordPress 4.2.3	✔	✔
CVE-2016-4010	Magento 2.0.5	✔	✖
CVE-2018-5301	Magento 2.0.5	✔	✖

Table 4.6: Measurements of the execution time, the CPU and memory consumption for the tested web applications with XDebug and Code Coverage (CC) and without XDebug. The reported values for the CPU and memory correspond to the average for each application.

Application		Execution (s)	CPU (%)	Memory (%)
Magento 2.0.5	*Without XDebug*	317	21.7	10.7
	With CC	584 (x1.85)	56.9 (x2.62)	11.82 (x1.10)
MediaWiki 1.2.8	*Without XDebug*	36	30.7	5.2
	With CC	121 (x3.38)	79.3 (x2.58)	6.9 (x1.31)
phpMyAdmin 4.7.0	*Without XDebug*	102	3.7	5.7
	With CC	116 (x1.14)	31.5 (x8.47)	5.6 (x0.97)
WordPress 4.7.1	*Without XDebug*	68	8.2	8.2
	With CC	170 (x2.50)	42.6 (x5.22)	12.5 (x1.53)

4.2 Performance analysis

It is known that code-coverage tools impose a non-negligible overhead on web applications [67]. In this section, we report on the results of conducting all the Selenium tests with and without `XDebug` (our chosen PHP profiler) while measuring execution time, and recording server-side CPU usage and memory consumption. Table 4.6 presents the overall results and Figure 4.5 focuses on CPU consumption.

First, looking at the execution time, we can see that code coverage has a varying impact on the tested web applications. On one hand, phpMyAdmin is lightly affected with a 14% increase. On the other hand, the time it takes to run all tests

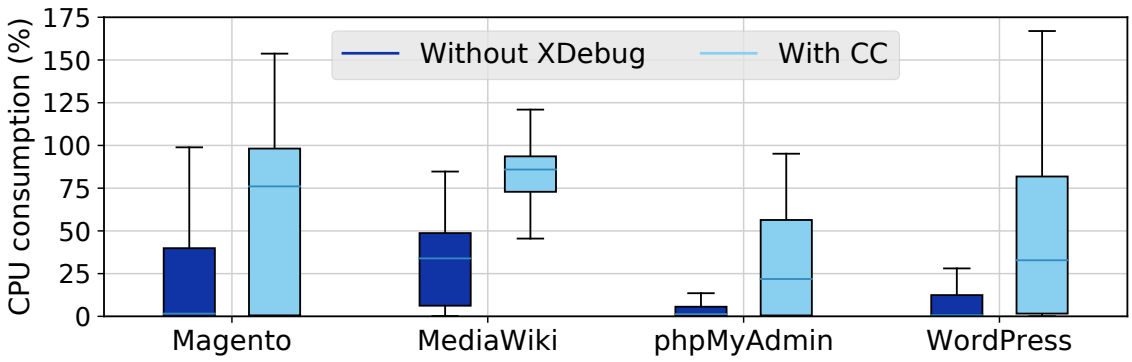


Figure 4.5: Measurement of the CPU consumption for the tested web applications. 100% corresponds to the use of a single CPU core.

for MediaWiki has tripled. For CPU consumption, the overhead is noticeable and all applications at least double their use of resources when code coverage is active. phpMyAdmin is exhibiting the biggest performance hit with a reported average almost 9 times higher than the one from the base application. Figure 4.5 shows that all median values are higher for applications with `XDebug` and most applications, at some point, require a second core with values above 100%. Finally, in terms of memory consumption, the server-side code profiler incurs a relatively modest increase for most applications. The worst overhead is observed when evaluating WordPress with an increase of 4.3% of the total device memory (16GB), i.e., an additional 700MB of RAM.

Even though our results show that the overall overhead is substantial, it is important to note that this overhead is not the overhead of the debloated web applications. Debloated web applications do not require code-coverage statistics and will therefore execute in the exact same environment as the original application (i.e. without `XDebug`). Depending on how code-coverage information is obtained, this overhead may or may not be an issue. For example, if the coverage is calculated in an offline fashion where traces of application usage are replayed against a testing system, this overhead will have no impact on the real production systems. To allow for the online computation of code coverage (using real-time user traffic), we need more op-

timized code profilers. For example, `XDebug` currently overloads 43 opcodes to obtain line-level code-coverage information that is more fine-grained than required by our debloating techniques and incurs an unnecessary performance overhead [66]. We leave the development and evaluation of faster code profilers for future work.

4.3 Limitations

In this study, we set out to precisely quantify the security benefits of debloating, when applied to web applications. Through a series of experiments, we demonstrated that debloating web applications has a number of very concrete advantages. We showed that debloating can, on average, decrease an application's code base by removing hundreds of thousands of lines of code, reduce its cyclomatic complexity by 30-50% and remove code associated with up to half of historical CVEs. Moreover, even for vulnerabilities that could not be removed, debloating can remove gadgets that makes their exploitation significantly harder. Next, we discuss some of the inherent and technical limitations of our approach and future direction.

Lack of available exploits: The number of exploits publicly available compared to the total number of registered CVEs is low. At the same time, the effort to study vulnerability reports, find the relevant patch or bug report, and track the actual vulnerability down to source code level takes a non-negligible amount of manual labor. This lack of available exploits limits our ability to test the exploitability of vulnerabilities before debloating since certain vulnerabilities might only be exploitable under specific configurations. For example the set of five file-upload-related vulnerabilities in our MediaWiki dataset (marked as gray in Table 7) require access to file upload functionality which is disabled by default. A maintained set of automated, replayable exploits against popular web applications similar to "BugBox" introduced by Nilson et al. in 2013, could substantially help researchers at this step[59].

To address this issue, we mapped the CVEs to features within those applications. This is done by studying the architecture of target applications based on documentation within the code and available on their websites. We marked a CVE as unexploitable if the underlying feature is disabled by default, and online tutorials in our dataset do not require users to enable that functionality. This limitation only applies to reported numbers on removed CVEs and does not affect our results on POI gadgets since their mere existence is enough for them to be used in gadget chains.

Our approach results in lower bounds for CVE removal since disabling modules through application configuration does not guarantee removal of all code paths that trigger those modules. Taking CVE-2019-6703 as an example, a vulnerability was discovered in the WordPress "Total Donations" plugin [34] and disabling this plugin did not prevent attackers from invoking the vulnerable end point and running their exploits.

Dynamic code coverage: Given our reliance on dynamic code-coverage techniques, it is clear that the success of debloating a web application is tightly related to its usage profile. Even though we constructed profiles in a way that is reproducible and unbiased (i.e. by relying on external popular tutorials, monkey testing, crawlers, and vulnerability scanners), we cannot claim that real web users would not trigger code that was removed during the stage of debloating, while they are interacting with a debloated web application.

More specifically, our modeled usage profiles do not cover all possible benign states of target web applications as we assume that users do not use all available features. Our intuition behind debloating proves to be successful to a large degree since removing unnecessary features brings clear security improvements. At the same time, our current usage model may not cover deep error states (e.g. logical errors in multi-stage form submissions, or the invalid structure of uploaded files). As such, we intend to follow-up this work with crowd sourcing and user studies to understand how

administrators, developers, and regular users utilize the evaluated web applications and whether their usage profiles would allow for similar levels of debloating.

Due to nature of our approach, we can not take advantage of standard static-analysis techniques, since we aim to remove the features that are not useful for a given set of users, not those that are not reachable by other code. Using static analysis would greatly overestimate the code that needs to be maintained through the process of debloating and the resulting web application would contain code (and therefore vulnerabilities) that is not useful to all users. Going forward, we envision a hybrid approach where dynamic analysis is used as a first step to identify the core features that are useful for a specific set of users. These features can then be used as a starting point for a follow-up static analysis phase to ensure that all code related to these features is maintained when debloating a web application.

Handling requests to removed code: A separate issue is that of handling requests to removed code. Our current prototype utilizes assertions to log these requests so that we can investigate why the corresponding server-side code was not captured by our coverage profiler. When real users utilize debloated web applications, one must decide how these failures (i.e. client-side requests requiring server-side code that was removed) will be handled. Assuming that cleanly exiting the application and showing an error to the user is not sufficient, we need methods to authenticate the user's request, determine whether the request is a benign one (and not a malicious request that aims to exploit the debloated web application) and potentially re-introduce the removed code. The client/server architecture of web applications lends itself well to this model since the web server can decide to re-introduce debloated code and handle the user's request, without any knowledge of this happening from the side of the user. All of this, however, requires server-side systems to introduce the code at the right time and for the appropriate users. We leave the design of such systems for future work.

Metrics to measure debloating effectiveness: In this paper, we use Cyclomatic Complexity (CC), Logical Lines of Code (LLOC), reduction in historical CVEs, and POP gadget reduction as four metrics to measure the effects of debloating on different web applications. However, not every line of code contributes equally to a program's attack surface. For example, 15% of removed files from Magento 2.0.5 are test files for external packages and the core of the application. Such code may not be directly exploitable or used in a POP chain unless there is a misconfiguration (e.g., autoloading including these files, or the directories being publicly accessible). As such, the resulted reduction in source code metrics (CC and LLOC) may also reflect the code that does not contribute to the attack surface. Contrastingly, the reduction of exploitable CVEs draws a more realistic picture of real world attacks. The drawback of this metric is its unavailability for proprietary software and the manual effort required to map CVEs to source code and verify their exploitability.

Debloating effectiveness: Through our debloating experiments we discovered that, in terms of debloating, not all applications are "equal." Modular web applications debloat significantly better than monolithic ones (such as Wordpress). We hope that our findings will inspire different debloating strategies that lend themselves better to monolithic web applications which resist our current function-level and file-level debloating strategies.

4.4 Future work

In this section, we will discuss the future directions and improvements to our current prototype that we are willing to pursue in future.

4.4.1 End-to-end web application debloating

In this work, we showed that web application debloating has the potential to significantly reduce the attack surface despite the challenges involved. Next, we discuss some of these challenges and propose directions for our future work on this topic.

Handling calls to removed code

One of the biggest challenges of using debloating in production applications is handling calls to removed functions. In our current implementation, upon executing a function that leads to debloated code, user's operation is terminated. Then the user is notified that the intended functionality has been removed. This can happen after clicking on a link, or after several steps into filling a multi-step form. To make this more user friendly and enable website administrators to make the final decision, we envision several improvements:

- Adding the functionality to dynamically introduce back the removed code. This decision can be made either through an administration panel, or after a second factor of authentication for privileged users. For that, we need to identify risky actions and enforce further security mechanisms and identity verification steps before continuing. From another point of view, this step can be seen as an anomaly detection system based on history of the user behavior.

- Our current hypothesis is based on the observation that not all users of an application require all features. To go one step further, one can pursue, user and role based debloating. Under this scenario, different set of users for the same web application are provided with their own debloated versions based on their activity history.

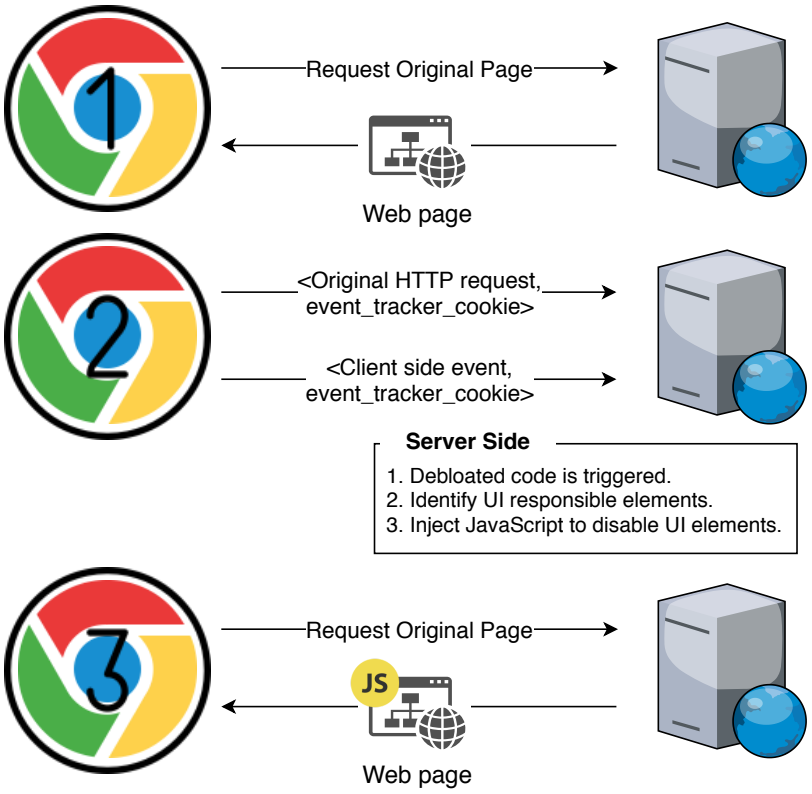


Figure 4.6: Overview of steps required to detect and remove UI elements for debloated actions.

Propagating debloating changes to the UI

Current debloating setup will show error messages to users, when they execute removed code. By disabling UI elements that exercise the removed features we can proactively stop users from running into errors. By tracking web application UI elements back to their implementation and doing a backward traversal on the call graph, we can build this dependency chain and disable, limit functionality or add informative messages to UI elements. This will inform the users about the disabled features before investing any time and effort on that function.

Implementation details: Web applications consist of two main components, server side and client side. Client side code runs on users browsers and generate HTTP requests based on user interactions with HTML and JavaScript components

on the page. From server's point of view, generation of HTML content and events received from the browser are totally disconnected. The goal is to build this bridge and track server side events as the result of client side interactions with minimal reliance on certain application architectures.

Figure 4.6 depicts the browser interactions required to detect and disable debloated actions on client side. Initially, PHP code is injected into all pages of the web application to add tracker cookies that are sent along page events that are reported to the server. During first user interaction with the page at step 2 (e.g., Form submission, click on a link, etc.), by correlating the events and UI elements that triggered the request through the cookie values, we are able to find elements and forms that trigger debloated events. The next time users request the same page, at step 3, the elements that triggered debloated actions are disabled through injected JavaScript.

Code coverage collection in production environment

By shipping the code coverage collection infrastructure to real world users and website owners, we gain an insight into how the applications are used by different types of users.

Optimized code coverage collection: To make code coverage recording less resouce intensive, we need to know the source of this overhead. Our toolset is built on top of XDebug PHP extension. To collect line level code coverage, XDebug hooks into various PHP opcodes. This makes the program execution slower and more resource intensive. Meanwhile, we only need function coverage information in our setup. This information can either be collected by a light-weight code profiler that is optimized for this task, or by developing and including PHP libraries in web applications that collect execution traces from call stack. To that end, we plan to develop optimized code profilers that can be easily installed and integrated with web applications with-

out the need to enable PHP extensions and undergoing significant performance hits. Furtheremore, by relying on function coverage instead of line coverage, our code coverage information will be more resilient to small changes that affect the order of source line numbers.

To further reduce the overhead, one can only record the code coverage for functions that include sensitive API calls (e.g., Filesystem and database interaction, dynamic code execution, network connections, etc.). The first step is to identify such functions, and through static analysis, identify code paths that interact with these APIs. By only tracking the code coverage on these specific functions, we can gain the benefit of debloating with an increased performance.

Attack surface reduction through API specialization

In the context of web application security, Runtime Application Self Protection (RASP) has been the hot technology in the recent years. By adding a defensive layer that detects attacks from within the application, the protection engine has access to execution context. This extra information increases the precision of the protection engine in detecting attacks.

Similar idea can be applied to debloating web applications. When normal users interact with a web application, functions and APIs are invoked with specific set of parameters. By dynamically generating and enforcing whitelists that only allow benign invocation of security critical APIs, we can block exploitation attempts and reduce the attack surface and exploitability of our software.

As a motivating example, we look at the following SQL injection vulnerability in AutoSuggest 0.24 WordPress plugin [2]:

Benign queries in listing 4.1 only target $wpdb.posts table, whereas an exploit will likely target other tables via subqueries and UNION and try to access users table and extract information. Notice that $wpdb.posts variable comes from WordPress

```
1 <?php
2 $wpas_keys = $_GET['wpas_keys'];
3 $pageposts = $wpdb->get_results("SELECT * FROM $wpdb->posts
4 WHERE (post_title LIKE '%$wpas_keys%') AND post_status = 'publish' ORDER BY post_date DESC");
```

Listing 4.1: PHP code from AutoSuggest WordPress plugin with SQL injection vulnerability

configuration file and is static for each installation of WordPress. By running each query under a different user that has the minimal permissions that satisfy the benign invocations of target query (e.g., by limiting the target table for a given query), the exploit attempt will fail. So in this example, the query would run under the automatically generated user named "wp_posts_user" which only has "SELECT" permission to WordPress "posts" table. This would limit the exploit to be able to extract information only from posts table.

To implement this idea, we would start by creating a list of security critical PHP functions. For each critical API, we distinguish benign and malicious invocations at runtime by learning from application usage and create a whitelist that is dynamically generated and enforces conditions to minimize the set of possible invocations of these APIs.

More concretely, we distinguish the following cases:

- **Database Operations:** In the context of database operations, we limit type of operation (e.g., SELECT query) and target tables (e.g., Posts table) and database (WordPress db) for each database interaction. This can be implemented as a proxy between the database and the application or by modifying the database layer within the application (e.g., modifying the ORM). This modified API should dynamically learn and decide how the api is called and what rules should be enforced.

- **File System Interactions:** For these APIs, we enforce target directories, allowed file extensions and their permissions. For example, if the code writes to

the file system, we dynamically learn the target directory and type/permission of files submitted through this API. Another possible variation is to mount virtual directories that are not directly accessible from the web, but are accessible by the PHP code itself. This prevents uploading a PHP backdoor and executing it directly from the web.

- **Limiting Available APIs:** By limiting the set of PHP Apis that are available to each PHP file, we can block potential exploits. For example, the login.php file should not interact with the file system or call "eval" or "exec".

4.5 Conclusion

In this project, we analyzed the impact of removing unnecessary code in modern web applications through a process called *software debloating*. We presented the pipeline details of the end-to-end, modular debloating framework that we designed and implemented, allowing us to record how a PHP application is used and what server-side code is triggered as a result of client-side requests. After retrieving code-coverage information, our debloating framework removes unused parts of an application using file-level and function-level debloating.

By evaluating our framework on four popular PHP applications (phpMyAdmin, MediaWiki, Magento, and WordPress) we witnessed the clear security benefits of debloating web applications. We observed a significant LLOC decrease ranging between 9% to 64% for file-level debloating and up to an additional 24% with function-level debloating. Next, we showed that external packages are one of the primary source of bloat as our debloating framework was able to remove more than 84% of unused code in versions that used Composer, PHP's most popular package manager. By quantifying the removal of code associated with critical CVEs, we observed a reduction of up to 60% of high-impact, historical vulnerabilities. Finally, we showed that the process

of debloating also removes instructions and classes that are the primary sources for attackers to build gadgets and perform POI attacks.

Our results demonstrate that debloating web applications provides tangible security benefits and therefore should be seriously considered as a practical way of reducing the attack surface of web-applications deployments.

Table 7: Comprehensive list of mapped CVEs and whether vulnerable files, functions or lines were triggered based on our usage profiles. Grey rows indicate CVEs located in modules that are, by default, disabled.

#	CVE	Ver.	Vulnerability Triggered			Affected Functionality
			Files	Functions	Lines	
phpMyAdmin						
1	CVE-2013-3238	4.0.0	✔	NA	✖	Rename table using Regex
2	CVE-2013-3240	4.0.0	✔	✔	✔	Plugins
3	CVE-2014-8959	4.0.0	✖	✖	✖	GIS Editor
4	CVE-2016-6609	4.0.0	✔	✖	✖	Export as phparray
5	CVE-2016-6619	4.0.0	✔	✖	✖	Recent tables
6	CVE-2016-6620	4.0.0	✖	✖	✖	Table tracking
7	CVE-2016-6628	4.0.0	✖	✖	✖	Create charts
8	CVE-2016-6629	4.0.0	✖	✖	✖	Configuration option
9	CVE-2016-6631	4.0.0	✖	✖	✖	Create transform plugins
10	CVE-2016-6633	4.0.0	✔	✖	✖	Import ESRI shape file
11	CVE-2016-9866	4.0.0	✔	NA	✖	User preferences
12	CVE-2016-5703	4.4.0	✔	✖	✖	Central columns
13	CVE-2016-5734	4.4.0	✔	✖	✖	Table search using Regex
14	CVE-2016-6616	4.4.0	✖	✖	✖	User groups
15	CVE-2017-1000017	4.4.0	✔	✔	✖	Replication
16	CVE-2016-6606	4.6.0	✔	✔	✔	Authentication cookies
17	CVE-2016-6617	4.6.0	✔	✖	✖	Export templates
18	CVE-2016-9849	4.6.0	✔	✔	✔	Authentication
19	CVE-2016-9865	4.6.0	✔	NA	✖	Core deserialization
20	CVE-2017-1000499	4.7.0	✔	✔	✔	Navigation tree
MediaWiki						
21	CVE-2013-2114	1.19.1	✔	✖	✖	File upload from chunks
22	CVE-2013-6453	1.21.1	✔	✖	✖	Verify uploaded file
23	CVE-2014-1610	1.21.1	✔	✖	✖	PDF Upload
24	CVE-2014-2243	1.21.1	✔	✔	✖	User settings
25	CVE-2014-5241	1.21.1	✔	✖	✖	JSON Output formatter
26	CVE-2014-9277	1.21.1	✔	✖	✖	Flash policy output
27	CVE-2014-9276	1.23.0	✔	✔	✔	Expand templates
28	CVE-2015-2936	1.24.0	✔	✔	✔	Authentication
29	CVE-2015-2937	1.24.0	✖	✖	✖	XMP data reader
30	CVE-2015-6728	1.24.0	✔	✖	✖	Get watchlists through API
31	CVE-2015-8002	1.24.0	✔	✖	✖	File upload from chunks
32	CVE-2015-8003	1.24.0	✔	✖	✖	File upload API
33	CVE-2015-8623	1.24.0	✖	✖	✖	User object
34	CVE-2015-8624	1.24.0	✖	✖	✖	User object
35	CVE-2017-0370	1.24.0	✔	✔	✔	Markup parser (blacklist)
36	CVE-2017-0362	1.28.0	✔	✔	✔	Track pages
37	CVE-2017-0363	1.28.0	✔	✔	✔	Search
38	CVE-2017-0364	1.28.0	✔	✔	✔	Search
39	CVE-2017-0367	1.28.0	✔	✔	✔	Localization cache
40	CVE-2017-0368	1.28.0	✔	✔	✔	System messages
41	CVE-2017-8809	1.28.0	✔	✔	✔	APIs and RSS
Magento						
42	CVE-2015-1397	1.9.0	✔	✔	✔	Prepare SQL condition
43	CVE-2015-1398	1.9.0	✔	✔	✖	OAuth & XML modules
44	CVE-2015-1399	1.9.0	✔	✔	✔	Actions predispatch
45	CVE-2015-8707	1.9.0	✔	✖	✖	Password reset
46	CVE-2016-2212	1.9.0	✔	✖	✖	Order status RSS
47	CVE-2016-4010	2.0.5	✔	✔	✔	Shopping cart
48	CVE-2016-6485	2.0.5	✔	✔	✔	Cryptography functions
49	CVE-2018-5301	2.0.5	✖	✖	✖	Delete customer address
WordPress						
50	CVE-2014-5203	3.9	✔	✔	✖	Widget customization
51	CVE-2014-5204	3.9	✔	✔	✔	CSRF token verification
52	CVE-2014-5205	3.9	✔	✔	✔	CSRF token verification
53	CVE-2018-12895	3.9	✔	✔	✔	Delete post thumbnail
54	CVE-2015-2213	4.0	✔	✔	✔	Untrash comment
55	CVE-2017-14723	4.0	✔	✔	✔	Prepared queries
56	CVE-2014-9033	4.0	✔	✔	✖	Password reset
57	CVE-2014-9037	4.0	✔	✔	✔	Password hashing library
58	CVE-2016-6635	4.0	✔	✖	✖	Ajax compression test
59	CVE-2014-9038	4.0	✔	✔	✔	HTTP request API
60	CVE-2015-5731	4.2.3	✔	✔	✖	Admin panel
61	CVE-2016-7169	4.6	✔	✔	✖	Sanitize uploaded file name
62	CVE-2017-17091	4.6	✔	NA	✖	Create new user
63	CVE-2017-5492	4.7	✔	✔	✔	Admin screen API, widgets
64	CVE-2017-9064	4.7	✔	✔	✔	Admin file system operations
65	CVE-2018-10101	4.7	✔	✔	✔	HTTP request API
66	CVE-2018-10100	4.7	✔	NA	✖	Login
67	CVE-2017-6815	4.7	✔	✔	✔	Redirect URL validation
68	CVE-2017-5611	4.7.1	✔	✔	✔	Query helper
69	CVE-2017-16510	4.7.1	✔	✖	✖	Prepared queries

Bibliography

[1] Automatically append or prepend files in a PHP script. https://www.php.net/manual/en/ini.core.php#ini.auto-append-file.

[2] AutoSuggest 0.24 WordPress plugin Exploit. https://www.exploit-db.com/exploits/45977.

[3] Burp Suite web vulnerability scanner. https://portswigger.net/burp.

[4] Cross-Site Request Forgery (CSRF) - OWASP. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

[5] Cross-site Scripting (XSS) - OWASP. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[6] Elastica: Elasticsearch client. https://github.com/ruflin/Elastica.

[7] Greasemonkey. https://www.greasespot.net/.

[8] gremlins.js. https://github.com/marmelab/gremlins.js.

[9] Guzzle: PHP HTTP client. https://github.com/guzzle/guzzle.

[10] less.js ported to PHP. https://github.com/oyejorge/less.php.

[11] NoScript browser extension. https://noscript.net/.

[12] PHP autoload built-in function. http://php.net/manual/en/language.oop5.autoload.php.

[13] PHP Object Injection Vulnerability. https://www.owasp.org/index.php/PHP_Object_Injection.

[14] PHP: register_shutdown_function - Manual. https://secure.php.net/manual/function.register-shutdown-function.php.

[15] PHP wakeup built-in function. http://php.net/manual/en/language.oop5.magic.php#object.wakeup.

[16] PHP Zend Framework 1. https://github.com/zendframework/zf1.

[17] PHP_CodeSniffer is a PHP package that tokenizes PHP, JavaScript and CSS files and detects violations of a defined set of coding standards. https://github.com/squizlabs/PHP_CodeSniffer.

[18] phpdbg PHP Debugger. https://github.com/krakjoe/phpdbg.

[19] PHPGGC: PHP Generic Gadget Chains. https://github.com/ambionics/phpggc.

[20] Remote Code Execution Vulnerability | Netsparker. https://www.netsparker.com/blog/web-security/remote-code-evaluation-execution/.

[21] SQL Injection: OWASP. https://www.owasp.org/index.php/SQL_Injection.

[22] Symfony PHP framework. https://github.com/symfony/symfony.

[23] WordPress Twenty Fourteen theme. https://wordpress.org/themes/twentyfourteen/.

[24] WordPress Twenty Twelve theme. https://wordpress.org/themes/twentytwelve/.

[25] XDebug Debugger and Profiler Tool for PHP. https://xdebug.org/.

[26] xhprof function-level hierarchical profiler for PHP. https://github.com/phacility/xhprof.

[27] NVD - CVE-2017-9841 (PHPUnit vulnerability). https://nvd.nist.gov/vuln/detail/CVE-2017-9841, 2017.

[28] Drupal Core - 3rd-party libraries -SA-CORE-2018-005 | Drupal.org. https://www.drupal.org/SA-CORE-2018-005, 2018.

[29] [HttpFoundation] Remove support for legacy and risky HTTP headers - Symfony framework on GitHub. https://github.com/symfony/symfony/commit/e447e8b92148ddb3d1956b96638600ec95e08f6b#diff-9d63a61ac1b3720a090df6b1015822f2R1694, 2018.

[30] NVD - CVE-2018-14773 (Symfony vulnerability). https://nvd.nist.gov/vuln/detail/CVE-2018-14773, 2018.

[31] Packagist statistics. https://packagist.org/statistics, 2018.

[32] PyPI Stats. https://pypistats.org/packages/__all__, 2018.

[33] Security Advisory: URL Rewrite vulnerability (Zend Framework). https://framework.zend.com/security/advisory/ZF2018-01, 2018.

[34] WordPress sites under attack via zero-day in abandoned plugin | ZDNet. https://www.zdnet.com/article/wordpress-sites-under-attack-via-zero-day-in-abandoned-plugin/, 2019.

[35] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS, NY, USA, 2008. ACM.

[36] H. Boomsma, B. V. Hostnet, and H. Gross. Dead code elimination for web systems written in PHP: Lessons learned from an industry case. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2012.

[37] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, Berkeley, CA, USA, 2003. USENIX Association.

[38] CVE Details: The ultimate security vulnerability datasource. https://www.cvedetails.com/.

[39] NIST: National Vulnerability Database. https://nvd.nist.gov/.

[40] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, New York, NY, USA, 2014. ACM.

[41] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering*, 17(12), 1991.

[42] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The Clock is Still Ticking: Timing Attacks in the Modern Web. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[43] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*. IEEE, 2006.

[44] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.

[45] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. How You Get Shot in the Back: A Systematical Study About Cryptojacking in the Real World. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, 2018.

[46] Y. Jiang, D. Wu, and P. Liu. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1.

[47] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. RedDroid: Android Application Redundancy Customization Based on Static Analysis. In *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering (ISSREâĂŹ18)*, 2018.

[48] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *Securecomm and Workshops*. IEEE, 2006.

[49] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A Client-side Solution for Mitigating Cross-site Scripting Attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, New York, NY, USA, 2006. ACM.

[50] Wallace Koehler. A longitudinal study of Web pages continued: a consideration of document persistence. *Information Research*, 9(2), 2004.

[51] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, 2018.

[52] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, EuroSec '19, New York, NY, USA, 2019. ACM.

[53] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. Attack surface reduction for commodity os kernels: Trimmed garden plants may attract less bugs. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, 2011.

[54] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of Network and Distributed Systems Security (NDSS)*, 2013.

[55] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11), 2013.

[56] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4), 1976.

[57] Steve McConnell. *Code complete*. Pearson Education, 2004.

[58] Shachee Mishra and Michalis Polychronakis. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[59] Gary Nilson, Kent Wills, Jeffrey Stuckman, and James Purtilo. Bugbox: A vulnerability corpus for PHP web applications. In *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test*, Washington, D.C., 2013. USENIX.

[60] Magento: eCommerce Platform. `https://magento.com/`.

[61] MediaWiki: Free and Open Source Software Wiki . `https://www.mediawiki.org/wiki/MediaWiki`.

[62] phpMyAdmin: MySQL web administration. `https://phpmyadmin.net/`.

[63] Anh Quach, Aravind Prakash, and Lok Kwong Yan. Debloating Software through Piece-Wise Compilation and Loading. *Proceedings of USENIX Security*, 2018.

[64] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, New York, NY, USA, 2017. ACM.

[65] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, New York, NY, USA, 2012. ACM.

[66] Derick Rethans. Code Coverage: The Present. `https://derickrethans.nl/code-coverage.html`.

[67] Derick Rethans. Xdebug's Code Coverage speedup. `https://derickrethans.nl/xdebug-codecoverage-speedup.html`.

[68] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. *Ndss*, (February), 2018.

[69] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, 2007.

[70] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, NY, USA, 2018. ACM.

[71] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008.

[72] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. 2017.

[73] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most Websites Don'T Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, New York, NY, USA, 2017. ACM.

[74] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, New York, NY, USA, 2018. ACM.

[75] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, volume 2007, 2007.

[76] Laurie Voss. The State of JavaScript Frameworks. `https://www.npmjs.com/npm/the-state-of-javascript-frameworks-2017-part-2-the-react-ecosystem`, 2018.

[77] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In *European Symposium on Research in Computer Security*. Springer, 2018.

[78] WordPress: OpenSource Content Management System. `https://wordpress.com/`.

[79] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2), February 2002.