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 [2, 21, 36], Cross-Site Request Forgery [1, 20, 11], Remote Code Execution [3], SQL injection [4, 15], and timing attacks [13, 14]. 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 [22, 38, 17].

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 [26] and to remove unnecessary code from the Linux kernel [25, 24]) and more recently on shared libraries [27, 28] and compiled binary applications [16].

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 as follows:

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. 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) [32]. On the other hand, similar to our work, we observe approaches that target the actual features that are unused under certain circumstances. By defining the bloat as unused code in the applications, as we show later in the results section for web applications, one can mitigate the vulnerable pieces of code and reduce the attack surface of applications.

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, high level manual specifications and usage profiles with dynamic application instrumentation to detect unused code. Contrastingly, Delta Debugging is used. this approach is different in that it starts by removing lines of source code and then verifies if the reduction is valid [39]. TODO: Add the references

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., Lines of code), reduction in the number of gadgets available for attackers and removed historical vulnerabilities are measured and reported. TODO: complete the list

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 [34]. 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 [31]. 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) [12]. 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.

2.0.2 Debloating in other platforms

Regehr et al. developed *C-Reduce* which is a tool that works at the source code level [30]. 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 [35]. 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 [16].

All three aforementioned approaches are founded on Delta debugging [39]. 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 [33]. 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* [18]. Jiang et al. also designed *RedDroid* to reduce the size of Android applications with program transformations [19]. Quach et al. adopted a different approach by bringing dead-code elimination benefits of static linking to dynamic linking [28].

Rastogi et al. looked at debloating a container by partitioning it into smaller and more secure ones [29]. 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 [23]. 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.

Chapter 3

Design & Methodology

In this section, we start by looking at the 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 [37]. Similarly, PyPI, the package manager for Python, reports more than a billion a month [9], while

Packagist, the main repository for Composer package manager for PHP, reports the download of 500 million packages each month [8].

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 [7]) 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 [5]. 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 [6] reveals that the vulnerable lines were derived from yet another framework called Zend [10]. 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.

Chapter 4

Results & Conclusion

In this work, we explain how to use the puthesis.cls class file and the accompanying template.

4.1 Future Work

Future work should include options in the template for a masters thesis or an undergraduate senior thesis. It should also support running headings in the headers using the 'headings' pagestyle. The print mode and proquest mode included in the template might also be candidates to include in the class itself.

Appendix A

Implementation Details

Appendices are just chapters, included after the *appendix* command.

A.1 Switching Formats

When switching `printmode` on and off (see Section **??**), you may need to delete the output .aux files to get the document code to compile correctly. This is because the hyperref package is switched off for `printmode`, but this package inserts extra tags into the contents lines in the auxiliary files for PDF links, and these can cause errors when the package is not used.

A.2 Long Tables

Long tables span multiple pages. By default they are treated like body text, but we want them to be single spaced all the time. The class therefore defines a new command, *tablespacing*, that is placed before a long table to switch to single spacing when the rest of the document is in double spacing mode. Another command, *bodyspacing*, is placed after the long table to switch back to double spacing. Normal

tables using `tabular` automatically use single spacing and do not require the extra commands.

When the documentclass is defined with the 'singlespace' option, these commands are automatically adjusted to stay in single spacing after the long table.

Make sure there is always at least one blank line after the *\bodyspacing* command before the end of the file.

Some times long tables do not format correctly on the first pass. If the column widths are wrong, try running the LaTeXcompiler one or two extra times to allow it to better calculate the column widths.

If you want your long table to break pages at a specific point, you can insert the command *\pagebreak*[4], to tell LaTeXthat it really should put a page break there. *\pagebreak*[2] gives it a hint that this is a good place for a page break, if needed. If there's a row that really should not be broken across a page, use \\∗, which will usually prevent a pagebreak.

A.3 Booktabs

The booktabs package is included to print nicer tables. See the package documentation [**?**] for more details and motivation. Generally, all vertical lines are removed from the tables for a better visual appearance (so don't put them in), and better spacing and line thicknesses are used for the horizontal rules. The rules are defined as *\toprule* at the top of the table, *\midrule* in between the heading and the body of the table (or between sections of the table), and *\bottomrule* at the end of the table. *\cmidrule* can be used with the appropriate options to have a rule that spans only certain columns of the table.

A.4 Bibliography and Footnotes

The bibliography and any footnotes can also be single spaced, even for the electronic copy. The template is already setup to do this.

Bibliography entries go in the .bib file. As usual, be sure to compile the LaTeXcode, then run BibTeX, and then run LaTeXagain.

To cite websites and other electronically accessed materials, you can use the '@electronic' type of BibTeX entry, and use the 'howpublished' field to include the URL of the source material.

The formatting of bibliography entries will be done automatically. Usually the titles are changed to have only the first word capitalized. If you'd prefer to have your original formatting preserved, place the title in an extra set of curly braces, i.e., "title = {{My title has an AcroNyM that should stay unchanged}},".

A.5 Figures and Tables

The captions of figures and tables take an optional parameter in square brackets, specifying the caption text to be used in the Table of Contents. The regular caption in curly braces is used for the table itself.

Generally captions for tables are placed above the table, while captions for figures are placed below the figure.

Bibliography

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

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

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

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

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

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

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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 (ISSRE18)*, 2018.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

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

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

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