Inferring and Securing Software Configurations using Automated Reasoning

Paul Gazzillo
paul.gazzillo@ucf.edu
University of Central Florida
Orlando, FL, USA

ABSTRACT

Software configurability opens the door to misconfiguration vulnerabilities, invalid settings that expose software weaknesses. Misconfiguration is one the top ten most critical security risks and the most common. This paper envisions a world without misconfiguration vulnerabilities through the use of automated reasoning techniques to infer and secure software configurations. Real-world software, however, often lacks an explicit specification of secure configurations, relying on hand-validation by users. Real-world systems comprise many individual highly-configurable software components, making the space of possible configurations for the whole system enormous. To realize our vision and overcome these challenges, we aim to create a rigorous definition of configuration specifications, use formal methods to mechanize the inference and generation of valid configurations, and develop algorithms to automatically secure against misconfiguration.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**.

KEYWORDS

software configuration, formal logic, configuration sampling

ACM Reference Format:
Paul Gazzillo. 2020. Inferring and Securing Software Configurations using Automated Reasoning. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3368089.3417041

1 INTRODUCTION

Highly-configurable software forms the basis of much of our computing infrastructure, because configurability enables reuse. Virtually endless variations of the software are possible with little or no modification to the source code. The Linux kernel and the Apache webserver are examples of highly-configurable software, which have thousands of configuration options. The Linux kernel is used by about 70% of mobile devices [26], by 70% of IoT developers [10], and over 40% of servers [28]. The Apache webserver is used by nearly 40% of web servers [27]. These alone are used in billions of computing devices. Their extreme configurability makes this widespread use possible.

Configurability opens the door to *misconfiguration vulnerabilities* that expose software bugs [1, 9, 18, 19] and security holes [21, 22, 30]. In fact, security misconfigurations are number six in the OWASP top ten list of the most critical security risks [23] and are the most common risk [22]. Misconfiguration is possible, because only certain configuration settings are valid: the misconfiguration bug or insecurity would not exist if the configuration settings were used correctly. Current recommendations for preventing misconfigurations include finding a *hardened*, or "locked-down" configuration and using it on all deployments, preferably automatically [23]. Hardening is typically done manually by system administrations following hardening guides [29], using detectors for known misconfigurations [7], or cloning a known secure configuration. Existing hardening practices are untenable, because they are subject to human error [8, 21, 24, 30], they are time-consuming and inflexible, and guides go out-of-date as software changes [29].

Program analysis and verification have had much success catching vulnerabilities caused by software weaknesses introduced during development. But misconfiguration vulnerabilities fall outside the scope of traditional program analysis, which focuses on source or object code. Misconfiguration vulnerabilities, in contrast, are rooted in *software configuration management* rather than the programming language. Software configuration management, in a nutshell, is the control of *change* to a software system [14]. While in general it includes change at every phase of the life cycle, misconfiguration vulnerabilities arise from the ability to change software *after* development. Developers include machinery to automate such post-development changes so that users and system administrators can tailor software without additional programming effort. But the virtually endless combinations of configuration settings within and across software systems make misconfiguration all but a certainty.

Our vision is a world without misconfiguration vulnerabilities, made possible with automated reasoning that produces reliable and secure software configurations. Automation both reduces the risk of misconfiguration and eases the burden of configuration management on developers, system administrators, and users. The key to this vision is the use of formal logic: by modeling the configuration specifications in formal logic, *the validity of a configuration is equivalent to Boolean satisfiability.* Automated reasoning tools such as Satisfiability Modulo Theories (SMT) solvers can then be used to both enforce correct usage and help discover desirable configurations. While formal logic has been used for software configuration in prior work [2, 11], there remain two key challenges standing in the way of our vision.

```
<Limit PUT DELTE BIND>
</Limit>
```

(a) The .htaccess file.

```
./configure --enable-dav
```

(b) The build option that compiles the WebDAV module.

```
a2enmod dav
```

(c) The tool that enables the WebDAV module.

Figure 1: The three separate configuration mechanisms involved in Optionsbleed. Unless both (b) and (c) are configured, (a)'s use of BIND will expose Optionsbleed.

The first challenge is that *real-world software systems do not have complete, explicit configuration specifications* defining what configurations are valid [4, 11, 25]—one reason for hardening guides. The second challenge is that *the space of possible configurations is enormous* [11, 20]. For instance, v4.19.50 of the x86 Linux kernel has 13,381 build-time options. With most being Boolean options, that is $2^{13,381}$ configurations, more than the estimated number of atoms in the universe! With the kernel being only one component of a complete software system. Scaling to massive systems is critical for the adoption of formal reasoning.

To realize our vision, we aim to (1) create a rigorous definition of configuration specifications, (2) use formal methods to mechanize the generation of valid configurations, and (3) develop algorithms to automatically secure against misconfiguration. This effort benefits all users of computing infrastructure, because the most critical software is some of the most configurable. System developers and administrators will benefit by being able to securely configure their infrastructure more accurately and quickly. Security misconfigurations are already the most common security risk, and as our world increasingly depends on infrastructure that combines more and more configurable software, this risk will only continue to increase.

2 MOTIVATING EXAMPLE: OPTIONSBLEED

The Optionsbleed vulnerability illustrates how a lack of explicit specification of what configurations are valid leads to a silent security misconfiguration. Optionsbleed was an exploit in the Apache webserver that "bleeds" arbitrary memory contents to a remote attacker [6] due to a use-after-free bug in its HTTP method handling code. It manifests only if a user has written an invalid configuration file, which is possible because of the lack of automatic validation of configuration file usage before execution.

Optionsbleed involves the misconfiguration of configuration options that span the Apache webserver's many configuration mechanisms. The first is the user-specified, and attacker-controllable, `.htaccess` file, shown in Figure 1a. The `Limit` directive restricts permissions to the specified HTTP methods, a useful setting for securing a server. But an invalid HTTP method name exposes the Optionsbleed vulnerability. For instance, while `PUT` is a valid method, `DELTE` is a misspelling of `DELETE`, which triggers the bleed.

$$\begin{aligned}&\texttt{limit.method} = \texttt{PUT}\ \textbf{or}\ \texttt{limit.method} = \texttt{DELETE}\\ \textbf{or}\ &(\texttt{build.enable-dav} = \text{True}\ \textbf{and}\ \texttt{module.dav} = \text{True}\\ &\textbf{and}\ \texttt{limit.method} = \texttt{BIND})\end{aligned}$$

Figure 2: Formalized constraints that prevent Optionsbleed.

While string matching can prevent a misspelling, i.e., input validation, Optionsbleed misconfigurations are more subtle due to interactions between configuration mechanisms. `BIND` in Figure 1a is only a valid HTTP method for the WebDAV extension, which adds filesystem-like methods to HTTP. The inclusion of WebDAV support in the webserver is configurable, which means that the use of `BIND` is only a misconfiguration under certain conditions, i.e., when WebDAV has *not* been enabled. This shows that there are global constraints on valid uses of the Limit directive that are outside the control of `.htaccess`. *Two more* configuration mechanisms affect the WebDAV extension, adding further constraints on the valid usage of Limit. A `configure` script (Figure 1b) controls compilation of the extension and a runtime module system controls its inclusion in the webserver at runtime (Figure 1c).

Since many other server extensions add new HTTP methods, trying to prevent such misconfiguration by only detecting known ones leaves openings for the same misconfiguration vulnerability. Ironically, security best practices recommend disabling unneeded features to reduce the attack surface [23], which in this case *increases* the opportunities to trigger Optionsbleed. Validating the `.htaccess` file is impossible without considering all configuration specifications. These specifications are fragmented across multiple configuration mechanisms, yet there are implicit, global constraints among them that no individual mechanism validates alone.

3 SOLUTION APPROACH

Automatically validating a configuration requires a unified, global view of all configuration specifications for a system. Intuitively, a configuration option is a long-lived value, global to an entire software system and (typically) only set once at the beginning of execution. In this sense, an option is effectively a program variable, with two important distinctions. First, configurations options are defined outside any particular program and exist across potentially all programs comprising a system. Second, they typically do not change during program execution except perhaps via a clearly delineated settings menu. Given the configuration options available for software system, we can formalize a specification of valid configurations as a set of logical constraints among options, so that the validity of a configuration is equivalent to Boolean satisfiability.

For example, Figure 2 shows the constraints relevant to Optionsbleed, where the options from all three configuration mechanisms are represented with a name: `build` for the configure script, `module` for the module loader, and `limit` for the `.htaccess` file's Limit directive. The logical connectives represent the global constraints on these options. The satisfying assignments of these options are the valid configurations that prevent triggering Optionsbleed. Using formal logic not only makes the notion of a valid configuration rigorous, but it enables automatic inference and generation of configurations for use in testing and security. We propose four initial

```
 1 namespace limit {
 2   config method : string
 3   constraint method="GET"
 4           or method="POST"
 5           or method="DELETE"
 6           or method="BIND" and build.dav
 7 }
 8 namespace build {
 9   config dav : bool
10   config dav_fs : bool
11   constraint dav <-> dav_fs
12 }
```

Figure 3: An example of intermediate configuration language relevant to Optionsbleed

tasks to realize our vision. Evaluating these tasks involves taking multiple large, prevalent, highly-configurable open-source codebases to determine whether the formalization of configurability and the proposed solution approach can successfully infer and generate valid configurations correctly and efficiently at scale.

3.1 An Intermediate Configuration Language

We propose creating an intermediate configuration specification language to serve as both a target language for extracting specifications from configuration mechanisms and a source language for generating logical formulas. Figure 3 is a preliminary example of such a language. It describes valid Apache webserver configurations, which prevents the Optionsbleed vulnerability. Lines 2, 9, and 10 declare configuration options. Namespaces delineate configuration mechanisms, e.g., `method` describes the Limit directive while the WebDAV options are in `build`. Boolean expressions define the constraints on the Limit directive (lines 3–6), which eliminates the misspelling misconfiguration. Line 6 makes explicit the interaction between the WebDAV extension and parameters to Limit, ensuring Optionsbleed is not part of any valid configuration.

While these specifications could be written by hand, we propose bootstrapping their creation by automatically extracting or "decompiling" them from build system code. We have demonstrated such extraction directly to formal logic on Linux's Kbuild [11, 20] and plan to retarget the extraction to the intermediate language. When software has no explicit description of valid configurations, we can infer implicit specifications from configurations that "work", i.e., those that build and run without breaking the software.

3.2 Formal Modeling and Analysis

In order to automatically reason about configurations, we can define the semantics of the intermediate language in terms of formal logic, then automatically compile the language to logical formulas. Enforcement of valid configurations is then equivalent to satisfiability, with satisfiability checks discharged to existing SAT and SMT solvers. Both static and dynamic analyses of the configuration constraints are useful for optimizing the compilation as well as supporting the testing and security of configuration specifications. For instance, namespacing in our intermediate language provides a priori information for opportunistic partitioning, which helps make SMT solving faster [15].

Sampling configurations [5, 13, 16], i.e., generating valid concrete configurations from logical constraints, supports securing configurations automatically. Real-world configuration specifications, however, are large, and constrained sampling remains an open problem [17]. To improve over existing sampling techniques, we propose using information from the intermediate configuration specification to help direct the sampling effort and scale to large configuration specifications. For instance, each configuration mechanism defines many configuration options that only have constraints among its own options, rather than across other configuration mechanisms. It is likely that only a fraction of configuration options are constrained with other parts of the system-wide specification. In the Linux kernel, for example, configuration options are often tied to a specific kernel subsystem, where a large portion of options are used only within their own subdirectory. We can use such information to reduce the burden on solvers and samplers.

3.3 Testing and Bug-Finding

Misconfiguration vulnerabilities fall into two categories. The first are *invalid configurations that expose a defect*. The Optionsbleed vulnerability described in Section 2 is an example. The second are *valid configurations that result in undesirable security properties*, which will be addressed in Section 3.4. We propose tackling the first kind of misconfiguration by both testing software across many configurations to find invalid configurations, i.e., those that expose software bugs, and combining logical constraints with static analyses for static bug-finding.

A fundamental challenge to fixing bugs is localizing the cause of the failure. It is not feasible to create a single test suite that exercises all possible configurations, but new defects may appear in untested configurations. The problem of *localizing configurations* [12] is to find what configurations affect each specific part of the software. Optionsbleed, for instance, was caused by a use-after-free. Once the bug is found, a configuration localization algorithm would determine what configuration settings lead to it. The formalization described in Section 3.2 can be used to statically localize configurations by combining constraints from the configuration specification with configuration options that control relevant portions of source code. Armed with these constraints, we can also use configuration sampling techniques to search the constrained space of configurations for software defects and use localization to focus testing on only relevant configurations.

3.4 Security and Prevention

Security misconfigurations are a broad category of security risks that have a wide variety of attack models, ranging from software bugs like the Optionsbleed's use-after-free, to insecure settings like default passwords and open permissions. Confidentiality, integrity, and availability can all be affected, because misconfigurations are not one particular software weakness but a vehicle for exposing weaknesses. A misconfiguration attack occurs when an existing configuration leads to exploitable software or an attacker can influence the choice of configuration. Because misconfigurations expose weaknesses indirectly, a misconfiguration vulnerability inserted by an insider may provide plausible deniability compared to implementing a backdoor in a well-audited source code repository.

While an invalid configuration can reveal a software defect, a valid configuration is not necessarily secure. For example, webservers can be configured to log user interactions. While innocuous in some use-cases, e.g., a personal webserver, such logging needs careful hardening for third-party services. Millions of user's passwords have been leaked due to such misconfiguration [21, 30]. Secure configuration policies can be expressed in the configuration specification intermediate language to restrict valid configurations to those that meet the policy. The analysis algorithms from Section 3.2 can also help automatically create secure configuration policies. For instance, if the security policy can be checked via a test, automation can build, run, and check the software to discover the range of secure configurations.

4 RELATED WORK

Our work draws inspiration from the use of propositional logic for feature modeling [3] in the feature-oriented software design (FOSD) paradigm. Feature modeling defines software as combinations of software components. Most open-source system software, however, does not follow the FOSD paradigm and often has no explicit definition of valid configurations. Our goal is to model and infer configurability beyond software components alone, including password policies, logging, permissions, etc, while employing the same spirit of applying formal logic to software to tackle the challenges of producing safe and secure software configurations.

ACKNOWLEDGMENTS

The author would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is supported by the National Science Foundation under CCF-1941816.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2642937.2642990

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated.

[3] Don Batory. 2005. *Feature Models, Grammars, and Propositional Formulas.* Technical Report TR-18-02. The University of Texas at Austin, Department of Computer Science.

[4] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *Software Engineering, IEEE Transactions on* 39 (12 2013), 1611–1640. https://doi.org/10.1109/TSE.2013.34

[5] Myra Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2008. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *Software Engineering, IEEE Transactions on* 34 (09 2008), 633–650. https://doi.org/10.1109/TSE.2008.50

[6] CVE. 2017. CVE-2017-9798. https://nvd.nist.gov/vuln/detail/CVE-2017-9798. Accessed: 2020-06-10.

[7] detectify blog. 2016. OWASP TOP 10: Security Misconfiguration. https://blog.detectify.com/2016/06/17/owasp-top-10-security-misconfiguration-5/. Accessed: 2020-06-10.

[8] detectify blog. 2017. AWS S3 Misconfiguration Explained – And How To Fix It. https://blog.detectify.com/2017/07/13/aws-s3-misconfiguration-explained-fix/. Accessed: 2020-06-10.

[9] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do #Ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference* (Beijing, China) *(SPLC '16)*. ACM, New York, NY, USA, 65–73. https://doi.org/10.1145/2934466.2934467

[10] Eclipse Foundation. 2018. IoT Developer Survey Results. https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2018.pdf. Accessed: 2020-06-10.

[11] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 279–290. https://doi.org/10.1145/3106237.3106283

[12] Paul Gazzillo, Ugur Koc, ThanhVu Nguyen, and Shiyi Wei. 2018. Localizing Configurations in Highly-configurable Systems. In *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1* (Gothenburg, Sweden) *(SPLC '18)*. ACM, New York, NY, USA, 269–273. https://doi.org/10.1145/3233027.3236404

[13] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. 2011. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software* 84 (12 2011), 2208–2221. https://doi.org/10.1016/j.jss.2011.06.026

[14] IEEE. 2012. IEEE Standard for Configuration Management in Systems and Software Engineering.

[15] Matteo Marescotti, Antti E. J. Hyvärinen, and Natasha Sharygina. 2016. Clause Sharing and Partitioning for Cloud-Based SMT Solving. In *Automated Technology for Verification and Analysis*, Cyrille Artho, Axel Legay, and Doron Peled (Eds.). Springer International Publishing, Cham, 428–443.

[16] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)* (Austin, TX). ACM Press, New York, NY, 643–654. https://doi.org/10.1145/2884781.2884793

[17] Kuldeep S. Meel, Moshe Y. Vardi, Supratik Chakraborty, Daniel J. Fremont, Sanjit A. Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. 2015. Constrained Sampling and Counting: Universal Hashing Meets SAT Solving. *CoRR* abs/1512.06633 (2015). arXiv:1512.06633 http://arxiv.org/abs/1512.06633

[18] Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wasowski. 2017. Variability Through the Eyes of the Programmer. In *Proceedings of the 25th International Conference on Program Comprehension* (Buenos Aires, Argentina) *(ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 34–44. https://doi.org/10.1109/ICPC.2017.34

[19] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An Empirical Study of Real-World Variability Bugs Detected by Variability-Oblivious Tools. In *Proceedings of the 2019 11th Joint Meeting on Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 12. https://doi.org/10.1145/3338906.3338967

[20] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. *Uniform Sampling from Kconfig Feature Models.* Technical Report TR-19-02. The University of Texas at Austin, Department of Computer Science.

[21] Krebs on Security. 2019. Facebook Stored Hundreds of Millions of User Passwords in Plain Text for Years. https://krebsonsecurity.com/2019/03/facebook-stored-hundreds-of-millions-of-user-passwords-in-plain-text-for-years/. Accessed: 2020-06-10.

[22] OWASP. [n.d.]. OWASP Top 10 2017 GM Data Analysis. https://github.com/OWASP/Top10/tree/master/2017/datacall/analysis. Accessed: 2020-06-10.

[23] OWASP. [n.d.]. Top 10 -2017: The Ten Most Critical Web Application Security Risks. https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf. Accessed: 2020-06-10.

[24] PCWorld. 2016. Massive smart device botnet highlights the dangers of default passwords. https://www.pcworld.com/article/3127257/iot-botnet-highlights-the-dangers-of-default-passwords.html. Accessed: 2020-06-10.

[25] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. 2007. Is the linux kernel a software product line?. In *Proceedings of the International Workshop on Open Source Software and Product Lines* (Kyoto, Japan) *(SPLC-OSSPL)*. 134–140.

[26] statcounter GlobalStats. 2019. Mobile & Tablet Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share/mobile-tablet/worldwide/#monthly-201801-201801-bar. Accessed: 2020-06-20.

[27] W3Techs World Wide Web Technology Surveys. 2019. Usage statistics of Apache. https://w3techs.com/technologies/details/ws-apache/all/all. Accessed: 2020-06-10.

[28] W3Techs World Wide Web Technology Surveys. 2019. Usage statistics of Unix for websites. https://w3techs.com/technologies/details/os-unix/all/all. Accessed: 2020-06-10.

[29] Sachin Kumar Microsoft TechNet. 2014. Why you should avoid manual 'server hardening'. https://blogs.technet.microsoft.com/mspfe/2014/05/29/why-you-should-avoid-manual-server-hardening/. Accessed: 2020-06-10.

[30] Zack Whittaker. 2018. GitHub says bug exposed some plaintext passwords. https://www.zdnet.com/article/github-says-bug-exposed-account-passwords/. Accessed: 2020-06-10.