Poster: Detecting Missing Checks for Identifying Insufficient Attack Protections

Lingyun Situ
Nanjing University
Nanjing, China
situlingyun@seg.nju.edu.cn

Liang Zou
Nanyang Technological University
Singapore
liang.d.zou@gmail.com

Linzhang Wang
Nanjing University
Nanjing, China
lzwang@nju.edu.cn

Yang Liu
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

Bing Mao
Nanjing University
Nanjing, China
maobing@nju.edu.cn

Xuandong Li
Nanjing University
Nanjing, China
lxd@nju.edu.cn

ABSTRACT

Missing check for untrusted input used in security-sensitive operations is one of the major causes of various serious vulnerabilities. Thus, efficiently detecting missing checks for realistic software is essential for identify insufficient attack protections. We propose a systematic static approach to detect missing checks in C/C++ programs. An automated and cross-platform tool named *Vanguard* was implemented on top of Clang/LLVM 3.6.0. And experimental results have shown its effectiveness and efficiency.

CCS CONCEPTS

• **Security and privacy → Vulnerability scanners**; **Software security engineering**;

KEYWORDS

Missing Check, Static Analysis, Vulnerability Defense

ACM Reference Format:
Lingyun Situ, Liang Zou, Linzhang Wang, Yang Liu, Bing Mao, and Xuandong Li. 2018. Poster: Detecting Missing Checks for Identifying Insufficient Attack Protections. In *Proceedings of 40th International Conference on Software Engineering Companion, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 2 pages.
https://doi.org/10.1145/3183440.3194949

1 INTRODUCTION

Vulnerability detection plays one of the most important roles in improving the correctness of software code. Unfortunately, an automatic approach to detecting arbitrary types of vulnerabilities accurately does not exist according to Rice's theorem [1]. Thus, state of art security research has focused on digging various specific kinds of vulnerabilities such as buffer overflow, integer overflow and so on [3].

However, it is a common fact that missing checks for manipulable data used in security-sensitive operations is one of the major causes of various specific serious vulnerabilities. Therefore, efficiently detecting missing checks for realistic software is essential for identifying insufficient attack protections.

There are few works focusing on the detection of missing checks. Chucky [4] detects missing checks for security APIs usage based on assumption that missing checks are rare events compared with the correct conditions imposed on security-critical objects. Therefore, it is more suitable for analyzing mature code. RoleCast [2] statically explores missing security authorization checks without explicit policy specifications. But it is tightly bounded to web applications coded in PHP and ASP.

We propose a systematic static approach to detect missing checks for manipulable data used in sensitive operations in C/C++ code aimed at identifying insufficient attack protections. An automated and cross-platform tool named Vanguard was implemented on top of Clang/LLVM 3.6.0. Experimental results have shown its effectiveness and efficiency.

2 APPROACHES

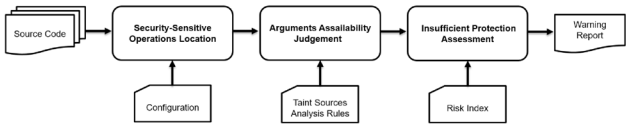

Figure 1: Overview of Vanguard

The overview of Vanguard is illustrated in Figure. 1. The input are source code of target C/C++ programs, configuration file. The output are missing check warnings.

First of all, Vanguard identifies locations of customized security-sensitive operations (SSO) with lightweight static analysis based on abstract syntax tree of target programs. Then, sensitive data used in SSO (i.e. dividend in division arithmetic, modulus in modular operation, index of array access, and arguments of security-sensitive function calls) are obtained to judge whether they are assailable by outside attack input, that is examining whether they are tainted using static taint analysis. Next, if a sensitive data is tainted, then a backward data-flow analysis is applied to explore whether there are attack protection checks for tainted data or related variables. If not, then a missing check is identified. Furthermore, Vanguard will extract its context features and estimate its risk degree, i.e. complexity. At last, Vanguard will generate and export warning report for the missing checks in high-risk context.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3194949

Table 1: Effectiveness and Efficiency of Vanguard

Project	AST	Func	Loc	Time	Sp(M)	missing checks warnings(false positive warnings)			Average False Positive
						divide/mod-zero	array-index-bound	argument-constraint	
Php-5.6.16	634	8499	497602	619.93	2793.2	43(14)	34(2)	196(83)	36%
Openssl-1.1.0	589	5692	284518	448.23	858.4	7(1)	32(6)	28(6)	19%
Pidgin-2.11.0	38	966	328153	37.57	471.7	27(3)	16(4)	63(11)	16%
Libtiff-4.0.6	83	629	66855	15.52	152.8	57(5)	5(1)	4(1)	10%
Libpng-1.2.44	60	337	24621	17.69	176.9	3(0)	4(0)	13(3)	15%

3 IMPLEMENTATION

An automated and cross-platform tool named Vanguard was implemented based on Clang/LLVM 3.6.0. It consists of four modules: (1) *Preprocessor*, which is used to obtain abstract syntax tree, control flow graph, and call graph of target programs; (2) *TaintAnalyzer*, which is in charge of establishing taint data pool using static intra-procedural and inter-procedural taint analysis; (3) *Detector*, which will identify missing checks via lightweight static analysis. (4) *RiskEstimator*, which estimates risk degree of detected missing checks in their contexts by computing context complexity.

Memory optimizing: In order to avoid crash while analyzing large-scale projects with *Vanguard* in limited memory environment, a memory optimizing mechanism was integrated. The key idea is to preserve latest used ASTs in memory with an AST queue.

Taint analysis optimizing: In order to accelerate the speed of static taint analysis. A tainted data pool consists of each variable expression's taint types is established and stored with format of 32bit unsigned int type array. It turns taint propagation analysis into bit computation of two bit arrays.

The core source code of *Vanguard* is available for download from https://github.com/stuartly/MissingCheck.

4 EVALUATION

Experimental evaluation is performed on a computer with 64-bit Ubuntu 16.04 LTS system, a processor of Intel(R) Xeon(R) CPU E5-1650 v3 @3.5Hz and 8GB RAM.

4.1 Effectiveness of Vanguard

We used Vanguard to analyze some open source projects and asked third party to count the false positive, the result illustrated in Table 1 shows that Vanguard is able to identify missing checks accurately with low false positive, i.e. 19% in average.

Furthermore, Vanguard's ability to identify missing checks lead us to uncover some vulnerabilities posted in national vulnerability database as illustrated in Table 2.

Table 2: Discovery of Vulnerabilities

Project	File	Function	Vulnerability
Openssl-1.1.0	stalen_dtls.c	BUF_MEM_grow_clean	CVE-2016-6308
Pidgin-2.10.11	protocol.c	mxit_send_invite	CVE-2016-2368
Libpng-1.5.21	pngrutil.c	png_read_IDAT_data	CVE-2015-0973
Libtiff-4.0.6	tif_fax3.c	_TIFFFax3Fillruns	CVE-2016-5323
Libtiff-4.0.6	tif_packbits.c	TIFFGetField	CVE-2016-5319

Based on above observations, we can know that Vanguard is able to detect various missing checks effectively with low false positive, and its ability to identify missing checks can help to uncover vulnerabilities.

4.2 Efficiency of Vanguard

As we can see from Table 1, Vanguard finishes analyzing PHP-5.6.16 using 619.93s. We count the time-overhead of Vanguard on project PHP-5.6.16 with increment of AST les, code lines, and functions. All the plots in Figure 2 have shown Vanguard's complexity of is nearly linear, which is scalable on large size of projects.

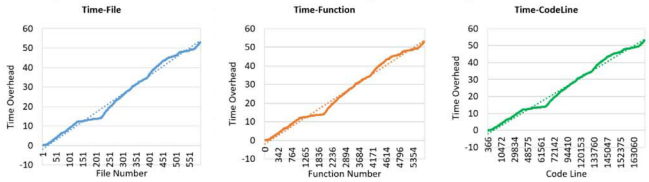

Figure 2: Time overhead with growth of scale

The effect of our memory optimizing is evaluated by analyzing PHP-5.6.16 with setting different size of AST queue. The result in Figure 3 indicates that Vanguard is capable of analyzing PHP-5.6.16 with lower space-cost when size of AST queue is smaller.

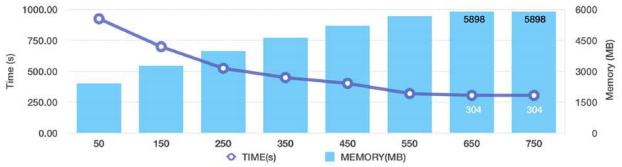

Figure 3: Result of memory optimizing

Based on above observations, we can know that *Vanguard* is capable of dealing with large-scale projects with low space-time overhead, and its complexity is nearly linear. Meanwhile, our memory optimizing technique allows Vanguard to be used in environments with limited memory resources adaptively.

5 CONCLUSION

Vanguard, an automatic static detection system for missing checks in C/C++ programs is designed and implemented on top of Clang/LLVM 3.6.0. It is able to identify missing checks by (1) locating security sensitive operations by lightweight static analysis; (2) judging assailability of sensitive data used in security-sensitive operations using static taint analysis; (3) assessing risk degree of detected missing checks in its context. Experimental results have shown *Vanguard*'s effectiveness and efficiency.

REFERENCES

[1] Michael Sipser. 2006. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston.

[2] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2011. Rolecast: finding missing security checks when you do not know what checks are. *ACM SIGPLAN Notices* 46, 10 (2011), 1069–1084.

[3] Hao Sun, Xiangyu Zhang, Chao Su, and Qingkai Zeng. 2015. Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 483–494.

[4] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 499–510.