LETTER

Detecting Violations of Security Requirements for Vulnerability Discovery in Source Code*

Hongzhe LI[†], **Jaesang OH**[†], ***Nonmembers***, ***and*** **Heejo LEE**[†a)], ***Member***

SUMMARY Finding software vulnerabilities in source code before the program gets deployed is crucial to ensure the software quality. Existing source code auditing tools for vulnerability detection generate too many false positives, and only limited types of vulnerability can be detected automatically. In this paper, we propose an extendable mechanism to reveal vulnerabilities in source code with low false positives by specifying security requirements and detecting requirement violations of the potential vulnerable sinks. The experimental results show that the proposed mechanism can detect vulnerabilities with zero false positives and indicate the extendability of the mechanism to cover more types of vulnerabilities.

key words: *software vulnerability, security sinks, security requirements*

1. Introduction

Despite the best efforts made by security communities and experts, the number of software vulnerabilities is still increasing rapidly on a yearly basis, leaving great threats to the safety of software systems. According to the Common Vulnerabilities and Exposures (CVE) database [1], the number of CVE entries has increased from around 1000 CVEs yearly in 2000 to over 8000 yearly in 2015. The discovery and removal of vulnerabilities from software projects have become critical issue in computer security. Nowadays, because of enormous amount of code being developed as well as limited human resources, it is becoming harder and harder to audit the entire code and accurately address the target vulnerability.

Security researchers have devoted themselves into developing static analysis tools to find vulnerabilities [2], [3]. The large coverage of code and access to the internal structures makes these approaches very efficient to find potential warnings of vulnerabilities. However, they often approximate or even ignore runtime conditions, which leaves them a great amount of false positives.

Recently, more advanced static analysis methods have been proposed [4]–[6]. They either encode insecure coding properties such as missing checks, un-sanitized variables and improper conditions into the analyzer for vulnerability discovery, or they model the known vulnerability properties and generate search patterns to detect unknown vulnerabilities. Even though these approaches can find vulnerabilities using search patterns and exclude the majority of code needed to be inspected, they still require security-specific manual efforts to verify the vulnerability at the very end. Recently, an automated approach is proposed [7] but it is limited to buffer overflow vulnerabilities. In other words, the previous methods cannot be extended to detect new vulnerability types.

In this paper, we propose an extendable mechanism to reveal typical types of vulnerabilities in source code with low false positives by specifying security requirements and detecting requirements' violations of the potential vulnerable sinks. In the beginning, a 3-element model {Sink, Argument(s), Requirement} is proposed to specify the security requirements of potential vulnerable sinks. Subsequently, we set security probes at each security sink in source code based on the specification table (see Table 1) and get ready to monitor the program execution and detect a vulnerability when the security requirement of a sink is being violated. Finally, the instrumented program source is passed to the concolic (CONCrete + symbOLIC) testing engine [8], [9] to detect any violation of security requirements and verify the existence of an actual vulnerable sink, which dramatically reduce the false positives. In this work, four popular types (buffer overflow, integer overflow, format string and divide-by-zero) of potential vulnerable sinks are discussed and specified and the mechanism can be extended to cover more types of vulnerabilities by adding {Sink, Argument(s), Requirement} entries in the specification table. We perform experiments with 400 test cases (100 cases for each type of vulnerability) from Juliet Test Suite [10]. The results show that our mechanism gets a detection result with *Precision* = 100% and *Recall* = 97.3% with zero false positives. A real program case (Cpio) is also discussed to show practicability of the proposed mechanism.

2. Proposed Mechanism

Discovery of vulnerabilities in a program is a key process to the development and management of secure systems. Current static analysis techniques can detect vulnerabilities in a fast and scalable way but they generate too many false positives. What's more, most of current methods can only detect limited types of vulnerabilities. In this work, we propose an extendable mechanism to reveal typical types of vulnerabilities in source code with low false positives by specifying

Manuscript received February 1, 2016.
Manuscript revised May 22, 2016.
Manuscript publicized June 13, 2016.

[†]The authors are with the Korea University, South Korea.

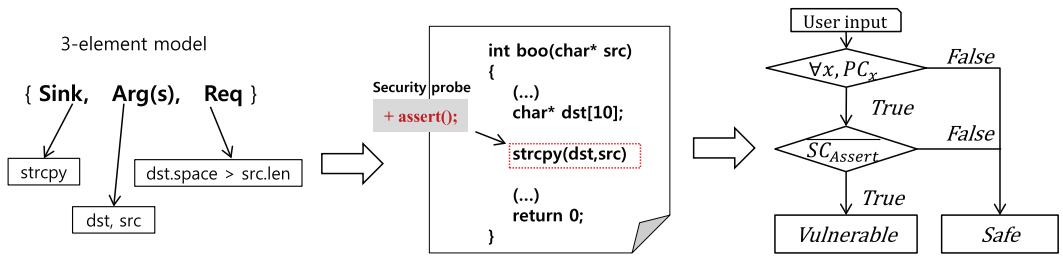
*This work was supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-16-2011, Development of Vulnerability Discovery Technologies for IoT Software Security).

a) E-mail: heejo@korea.ac.kr

DOI: 10.1587/transinf.2016EDL8035

Table 1 The specification of security requirements.

Type	3-element model			Description
	Sink	Argument(s)	Requirement	
Buffer overflow	strcpy(dst,src)	dst, src	dst.space > src.strlen	Space of dst should be bigger than the string length of src
	memcpy(dst,src,n)	dst, n	(dst.space ≥ n) ∧ (n ≥ 0)	Space of dst should be bigger or equal to the positive integer n
Integer overflow	unsigned addition x + y	x, y	(LONG64)x+y < 0x00000000ffffffff	The result value should be less than max value represented
	unsigned multiply x * y	x, y	(LONG64)x*y < 0x00000000ffffffff	The result value should be less than max value represented
Format string	printf(format, . . .)	formats, params	# formats = # params-1	Number of formats should be equal to the number of params-1
	sprintf(dst,format, . . .)	formats, params	# formats = # params-2	Number of formats should be equal to the number of params-2
Divide-by-zero	Division x / y	y	y <> 0	The value of y should not be zero



(1) Sink specification (2) Probes insertion (3) Violation detection

Fig. 1 General overview of our approach.

security requirements and detecting requirement violations of the potential vulnerable sinks.

Before the detailed description of the proposed mechanism, the general process is illustrated in Fig. 1. Our mechanism mainly consists of three phases which are **Sink specification**, **Probes insertion**, and **Violation detection**. In the phase of sink specification, we classify the security sinks which are considered to be potential vulnerabilities into the four categories: buffer overflow, integer overflow, format string and divide-by-zero. These types are reported as the most dangerous vulnerabilities according to SANS†. A 3-element model {Sink, Argument(s), Requirement} is proposed to specify the security requirement of each sink. In the second phase, we set security probes at each security sink in source code based on the specification table (see Table 1) and get ready to monitor the program execution and detect a vulnerability when the security requirement of a sink is being violated. In the last phase, we leverage concolic execution technique to detect the violation of security requirements and verify the existence of an actual vulnerable sink, which dramatically reduce the false positives.

2.1 Sink Specification

Since most of vulnerabilities are caused by attacker controlled data falling into a security sink [11], it is crucial to first understand security sinks. Here, we classify the security sinks according to typical types of vulnerabilities, i.e., buffer overflow, integer overflow, format string and divide-by-zero respectively.

- Sinks for buffer overflow: In security sensitive functions, the sensitive data is used as argument to be copied in a destination buffer. When destination buffer cannot hold the sensitive data, buffer overflow occurs. These functions include *strcpy*, *strcat*, *strncpy* and *memcpy*.
- Sinks for integer overflow: Arithmetic operations such as: unsigned add32 and multiply32. If the operation results in a value larger than the maximum value to be represented. The integer overflow occurs.
- Sinks for format string: In format string functions, the number of formats doesn't match the number of arguments to be formatted. Attacker can take use of this vulnerability to take control of the system. These functions include *printf, scanf and sprintf*.
- Sinks for divide-by-zero: The division operations. If the denominator argument is zero, this operation becomes into a divide-by-zero vulnerability.

Security sinks become into vulnerabilities when inappropriate data being used in the sinks. In order to use these sinks in a secure way, we must follow the security requirements of each sink. In this paper, we provide a 3-element model {Sink, Argument(s), Requirement} to specify the se-

†SANS: CWE/SANS TOP 25 Most Dangerous Software Errors, https://www.sans.org/top25-software-errors/.

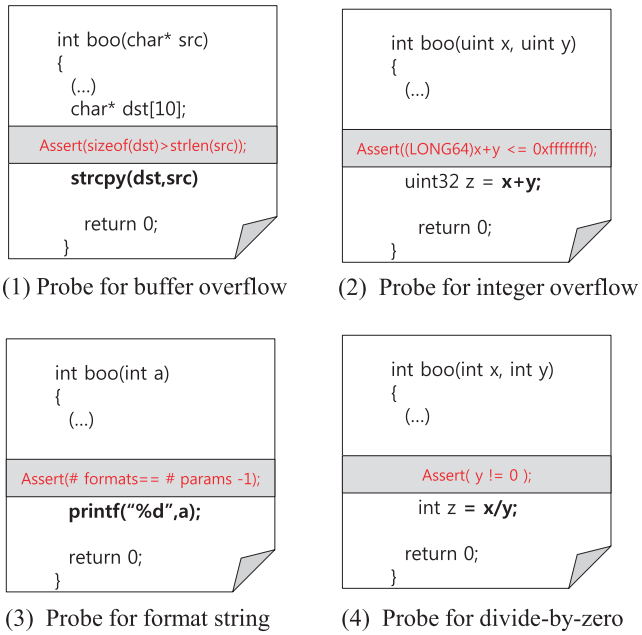


Fig. 2 Probes insertion for different types of sinks.

curity requirements of sinks. In the model, Sink(s) are the statements or instructions in the source code described and classified above. Arguments are data holding entries which are related to the sink's safety. Requirements are rules applied to Argument(s). The rule defines the appropriate assignment of data to the Argument(s) used in the sink to ensure the sink safety. On the contrary, if the security requirement rules are being violated (inappropriate data assigned), a sink becomes into a vulnerability. For example, the buffer overflow sink *strcpy(dst,src)*; Arguments are *dst* and *src*; the Requirement is: *dst.space* > *strlen(src)*. Table 1 shows the 3-element specification of security requirements regarding to different types of sinks.

Due to the space limitation, we do not list all the sinks regarding to each type in the table. Nevertheless, the specification table can be extended to cover more types of vulnerabilities by adding more sink specification items according to our 3-element model. After the classification and specification of security sinks, we identify the security sinks as potential vulnerabilities using a fast pattern matching approach and prepare to instrument the source code to set the security probes.

2.2 Setting Security Probes in Source Code

In this part, we set the security probes which is used to detect vulnerabilities in source code by instrumenting the source code. Based on our specification table, for each sink in the source code, we insert security probes (assertions) right before the sink. The condition of the assertion depends on the *Requirement* column in the table. After setting up these probes, the program is able to detect a violation of the security requirement by reporting "assertion failure" message when the program is executed with abnormal input. Figure 2 (1) to (4) illustrate the insertion of security probes for each type of sink in the source code respectively. As we can see, the probes are actually security constraints (SCs) which ensure the safety usage of security sinks. Any invalid usage of the sink will be caught by the probes and reported as a vulnerability.

2.3 Violation Detection (Vulnerability Verification)

After we set security probes in the source code, we have prepared a program equipped with vulnerability detection probes. We apply concolic (concrete + symbolic) execution technique in our mechanism execute the instrumented program and generate inputs to make the violation of security constraints of sinks. The general principle of the verification is to find inputs which satisfy all the program constraints (PCs) but violate the security constraints (SCs) as shown in Fig. 1.

Symbolic execution and concolic execution have been widely used in software testing and some have shown good practical impact, such as KLEE [9] and CUTE [12]. Our approach for concolic testing to verify potential vulnerable sinks can be described as follows. The program is first executed with concrete input data. During the execution, symbolic path constraints over the symbolic values at each conditional branch are collected. When the execution terminates, a new symbolic path formula is generated by negating one of the collected constraints. Then, the new set of constraints are sent to a SMT constraint solver [13] to solve the constraints and the solver generates the next input value to traverse another execution path. The program is iteratively executed with newly generated input in order to traverse different paths of the program. A vulnerability is verified when the false branch of the assertion statement is executed and the "assertion failure" message pops up at a certain execution.

3. Experimental Results

We have implemented our mechanism by developing a prototype system. We choose CREST-BV [14] as a basic concolic execution engine because of its good performance in test input generation speed.

Dataset: For the experiment, we collected 400 test cases from Juliet Test Suite [10]. Juliet Test Cases were created by US National Security Agency (NSA) and they are widely used for testing the effectiveness of software vulnerability detection and analysis tools. In each test case, there are good sinks and bad sinks which provide us ground truth for our evaluation. For example, in the test case file named CWE121_Buffer_Overflow_CWE131_memcpy_01.c, there are one good sink which is not vulnerable and one bad sink which is actually a vulnerability. The 400 test cases consist of four different types of sinks. Table 2 shows the distribution of test cases and the number of good and bad sinks regarding to each type.

Detection result: The evaluation metrics of our system

Table 2 Test cases from Juliet Test Suite.

Type	# of test cases	# of bad sinks	# of good sinks
Buffer overflow	100	100	138
Integer overflow	100	100	145
Format string	100	100	128
Divide-by-zero	100	100	181
Total	400	400	592

Table 3 Detection result.

Type	# of TPs	# of FPs	# of FNs	execution time(s)
Buffer overflow	93	0	7	26.53
Integer overflow	96	0	4	22.31
Format string	100	0	0	15.24
Divide-by-zero	100	0	0	17.39
Sum	389	0	11	81.47

```
1  static void
2  list_file(struct new_cpio_header* file_hdr, int in_file_des)
3  {
4      if(verbose_flag)
5      [...]                                          Integer overflow
6        char *link_name = NULL;
7        char *link_name = NULL;
8        link_name = (char *) xmalloc ((unsigned int) file_hdr->c_filesize + 1);
9        link_name[file_hdr->c_filesize] = '\0';
10       tape_buffered_read (link_name, in_file_des, file_hdr->c_filesize);
11       long_format (file_hdr, link_name);
12     free (link_name);                                buffer overflow
13     tape_skip_padding (in_file_des, file_hdr->c_filesize);
14    return;
      [...]
```

Fig. 3 CVE-2014-9112 vulnerability from Cpio-2.6.

tested on Juliet Test Cases are shown in Table 3. For the vulnerability detection (detection of bad sinks), we measure the number of true positives (TP), false positives (FP), false negatives (FN), precision and execution time regarding to each type of sink.

According to the table, we get zero false positives on all four types of sinks and we only get a few false negatives (missing vulnerabilities) in buffer overflow and integer overflow types. In our method, not all the vulnerabilities can be recognized because of complex code structure for some cases. This can be solved by improving our system to correctly analyze the complex code structure that we cannot handle now. We then calculate the precision ($\frac{TP}{TP+FP}$) and recall ($\frac{TP}{TP+FN}$) value for the overall detection result. As a result, our system gets the *Precision* of 100% which means zero false positives and the *Recall* of 97.3% on the whole data set.

Even though we only consider four basic types of vulnerability here, the mechanism can be easily extended to cover more types such as null pointer de-reference vulnerability by extending security specifications.

Real case: Cpio-2.6 (CVE-2014-9112). We also demonstrate the effectiveness of our mechanism to detect a vulnerability in a real open source project. Figure 3 shows the vulnerability (CVE-2014-9112) in program Cpio-2.6 which is a program to manage archives of files. This vulnerability is caused by an integer overflow at line 8, the numeric operation can cause an integer overflow and results in zero byte allocation for "link_name". This will again result in a buffer overflow which is at line 10 when the program is trying to write "c_filesize" number of bytes to zero space buffer. We apply our mechanism to detect this integer overflow vulnerability by setting the security probe "assert((unsigned long)c_filesize + 1 < 0x00000000ffffffff)" based on the sink specification table (Table 1) and generating an input which makes "$filesize\_c = 0xffffffff$" to report a requirement violation and trigger this vulnerability.

From the above results, our mechanism detects four popular types of vulnerabilities with zero false positives. The mechanism can be extended to cover more types of vulnerabilities by adding more 3-element sink specification items. Meanwhile, the real case study indicates the practicability of our mechanism to work in real world projects.

4. Conclusion

In this paper, we propose an extendable mechanism to reveal vulnerabilities in source code with low false positives by specifying security requirements and detecting requirements violations of the potential vulnerable sinks. Security requirements of sinks are specified by a 3-element model {Sink, Argument(s), Requirement} which makes it extendable to cover more types of vulnerability. We consider the detection of a vulnerability as detecting the violation of security requirements. The experiments conducted with Juliet Test Suite and a real project show that the proposed mechanism can detect vulnerabilities with low false positive rate and indicate the extendability of the mechanism to cover more types of vulnerabilities.

In future work, we will extend the specification method to cover more complicated vulnerability classes that maybe require more complicated security requirements.

References

[1] M. group, "Common Vulnerabilities and Exposures (CVE)," https://cve.mitre.org/, Accessed Jan. 2016.
[2] Wheeler and David, "Flawfinder," http://www.dwheeler.com/flawfinder/, Accessed Dec. 2015.
[3] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," IEEE Softw., vol.19, no.1, pp.42–51, 2002.
[4] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: exposing missing checks in source code for vulnerability discovery," ACM CCS, pp.499–510, ACM, 2013.
[5] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," IEEE S&P, pp.797–812, 2015.
[6] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," ACM CCS, pp.426–437, ACM, 2015.
[7] H. Li, J. Oh, H. Oh, and H. Lee, "Automated source code instrumentation for verifying potential vulnerabilities," IFIP SEC, vol.471, pp.211–226, IFIP, 2016.
[8] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," IEEE Trans. Autom. Sci. Eng., pp.443–446, IEEE Computer

Society, 2008.
[9] C. Cadar, D. Dunbar, and D.R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," USENIX OSDI, pp.209–224, 2008.
[10] T. Boland and P.E. Black, "Juliet 1.1 c/c++ and java test suite," Computer, vol.45, no.10, pp.88–90, 2012.
[11] S.D. Paola, "Sinks: Dom Xss Test Cases Wiki Project," http://code.google.com/p/domxsswiki/wiki/Sinks, Accessed Jan. 2016.
[12] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," ACM FSE, vol.30, no.5, pp.263–272, 2005.
[13] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in TACAS, vol.4963, pp.337–340, Springer, 2008.
[14] M. Kim, Y. Kim, and G. Rothermel, "A scalable distributed concolic testing approach: An empirical evaluation," IEEE ICST, pp.340–349, IEEE, 2012.