A Lightweight Method for Accelerating Discovery of Taint-Style Vulnerabilities in Embedded Systems

Yaowen Zheng[1,2,3], Kai Cheng[1,2,3], Zhi Li[1,2(✉)], Shiran Pan[2,3], Hongsong Zhu[1,2,3], and Limin Sun[1,2,3]

[1] Beijing Key Laboratory of IOT Information Security Technology, Beijing, China
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{zhengyaowen,chengkai,lizhi,panshiran,zhuhongsong,sunlimin}@iie.ac.cn
[3] University of Chinese Academy of Sciences, Beijing, China

Abstract. Nowadays, embedded systems have been widely deployed in numerous applications. Firmwares in embedded systems are typically custom-built to provide a set of very specialized functionalities. They are prone to taint-style vulnerability with a high probability, but traditional whole-program analysis has low efficiency in discovering the vulnerability. In this paper, we propose a two-stage mechanism to accelerate discovery of taint-style vulnerabilities in embedded firmware: first recognizing protocol parsers that are prone to taint-style vulnerabilities from firmware, and then constructing program dependence graph for security-sensitive sinks to analyze their input source. We conduct a real-world experiment to verify the mechanism. The result indicates that the mechanism can help find taint-style vulnerabilities in less time compared with whole-program analysis.

Keywords: Taint-style vulnerability · Embedded security · Protocol parser · Binary analysis · Reverse engineering

1 Introduction

Nowadays, embedded systems have been widely deployed in numerous applications. For example, routers and web cameras are normally used in home and office environment, and programmable logic controllers (PLC) are widely employed in industrial plants. If the security of such embedded systems are compromised, there might be serious consequences. For example, people's private information could be leaked to public or primary production process could be affected. Firmware of embedded systems is typically custom-built to provide a set of very specialized functionalities. Due to the resource-constrained nature of embedded systems, usually, firmware developers are more concerned about implementing the functionality and maximizing system performance; security concerns are often treated as afterthoughts and thus they are often treated inadequately. As a result, embedded system firmware is prone to vulnerabilities. Among various kinds of vulnerabilities, taint-style vulnerability refers to the case where

K.-Y. Lam et al. (Eds.): ICICS 2016, LNCS 9977, pp. 27–36, 2016.
DOI: 10.1007/978-3-319-50011-9_3

data propagates from an attacker-controlled input source to a security-sensitive sink without undergoing proper sanitization which could cause program crash or execute unauthorized operation [20]. Since embedded devices have frequent interaction with outside world through various user-input interfaces, undoubtedly, embedded systems are prone to taint-style vulnerabilities with a higher probability. In fact, taint-style vulnerabilities of embedded systems are reported frequently in various vulnerability reporting sources, including the Common Vulnerabilities and Exposures (CVE) [2], exploit-db [6] in recent years. With the increasing amount of embedded devices, the inability to quickly discover taint-style vulnerabilities will result in more serious security breaches in the future.

Taint-style vulnerability discovery technologies such as fuzz testing [14], symbolic execution [18], tainting analysis [16] could be applied to embedded systems. However, such traditional technologies suffer from low efficiency when analyzing firmware of embedded systems. First, source codes and design documents are often proprietary and thus only binary firmware image might be available, so that static analysis is time-consuming due to lack of semantic information. Then, as peripherals of different embedded devices have profound discrepancy, the unified dynamic simulation analysis is extremely difficult. In addition, firmware comparison technologies aimed at quickly finding homologous vulnerabilities in different devices have been studied. But they still suffer from large temporal and spatial overhead and low accuracy.

According to our study, current taint-style vulnerability discovery has a significant defect. Due to little understanding of code function, most approaches treat all codes equally and waste a lot of time on unimportant codes which have nothing to do with users input. To accelerate discovery of taint-style vulnerability, we investigate function modules that are more prone to this kind of vulnerability. Protocol parsers are function modules that handle protocol interaction and they are first lines of programs dealing with users input. They consume external input and either build an internal data structure for use, or orchestrate the execution of the proper functionality based on the input values. We assume that in firmware of embedded devices, once taint-style vulnerabilities exist, they are generally concentrated in protocol parsers. Therefore, we take two steps to accelerate discovery process. First, we construct a classifier using a set of features to recognize protocol parsers. Then, we derive program dependence graph (PDG) to analyze the input source of security-sensitive sinks. It help quickly extract insecure sinks where a static data flow path from attacker-controlled input source exists. The mechanism is lightweight as no time-consuming technology such as symbolic execution is employed. Finally, average time cost of insecure sinks finding is evaluated to prove efficiency of our work.

To summarize, our contributions include the following:

- We have proposed a two-stage mechanism to accelerate discovery of taint-style vulnerability in embedded devices.
- We have conducted a real-world experiment on firmwares of two cameras and reduced time cost of insecure sink finding by 81.4 and 44.2 percent.

The rest of the paper is structured as follows. We propose a two-stage mechanism to accelerate discovery of taint-style vulnerabilities in Sect. 2. Then we present experiment details about the classifier for protocol parsers and compare our work with whole-program analysis to illustrate the effectiveness of the proposed method in Sect. 3. Related work is presented in Sect. 4. Finally, conclusions are summarized in Sect. 5.

2 Two-Stage Mechanism

In the two-stage mechanism, we first recognize protocol parsers from firmware to narrow down the analysis scope of security-sensitive sinks, and then analyze input source of sinks based on PDG to help extract insecure cases. Insecure sinks refer to cases where a static data flow path from attacker-controlled input source exists. They need be further checked about data sanitization to finish the discovery of taint-style vulnerability. The mechanism is committed to quickly find insecure sinks in the former stage, which accelerate the discovery of taint-style vulnerabilities.

2.1 Protocol Parsers Recognition

Firmware is a software that provides control, monitoring and data manipulation of embedded systems. It is normally constitute of operating system, file system and user programs. To find out protocol parsers from firmware, we first extract the main service program from firmware. Then, we select a set of discriminative features. Finally, we use support vector machine (SVM) model to construct a classifier for parser recognition.

Firmware Pre-processing. In the pre-processing phase, we obtain the firmware of embedded system by downloading it from the official website. As it is compressed using standard compression algorithm and the file system adopt cramfs, jffs, yaffs these common formats, we use Binwalk [1] and firmware-mod-kit [4] to automatically extract binary programs from the firmware. Then, we manually find out the main service program that contain most of functional services. In general, the main service program is always running on online device and has various protocol keywords in it. It could be simply located with manual analysis.

Features. After finding out the main service program, we select a group of representative features from the respective of assembly form to represent functions. All features are shown in Table 1. In the following, we introduce the motivation why we choose these features.

First, the protocol parser should have features related to service. Since it has complex processing logic, the number of blocks is larger than other function modules. The protocol parser is usually called by various components, so it owns

more parent functions. Similarly, the protocol parser implements its functionality by calling other modules, so it also owns more child functions. In addition, the protocol parser normally executes different functionality depending on received network data, thus the number of child functions appearing in paths of switch branches is also important in recognizing the protocol parser. In Table 1, *blocks*, *inedges*, *outedges*, *switch_call* are used to represent above features.

Then, the protocol parser should have control structure features. As the protocol parser normally deal with external input, it would execute in different path depending on the received network data. Moreover, the process may run in cycles constantly. So a parser need contain switch statements or successive if-else statements, and it should have a parent function that calls it in a while structure. From the perspective of assembly form, the parser function should has a block that owns more than three branches or successive blocks all of which have side branches. Meanwhile, the parser function should be surrounded by a loop structure. In Table 1, *switch* and *loop* are relevant features.

Finally, the protocol parser should have features related to protocol processing. As the protocol parser normally deals with protocol communication, it follows the protocol specification and resolves particular characters and strings. For example, the HTTP protocol parser need compare the front part of header with GET and POST strings to determine the request method. Similarly, newline and return character would appear in the protocol parser as it is used to split protocol fields. Thus the occurrence frequency of const strings and string manipulation functions can reflect the likelihood of being a parser for a function. As shown in Table 1, *cmp_banner*, *strcmp* and *strstr* are features that represent the number of const strings, occurrence frequency of strcmp and strstr function calls. Besides, *cmp_spec* is to represent occurrence frequency of CMP instruction with ASCII code of character as the second argument.

Table 1. List of features

Features	Explanation
switch	Number of switch branches
loop	Equal 1 when surrounding loop structure exists otherwize 0
cmp_spec	Character comparison times
cmp_banner	String comparision times
strcmp	Frequency of strcmp occurrence
strstr	Frequency of strstr occurrence
switch_call	Number of functions invoked in switch branches
blocks	Number of blocks in functions
inedges	Number of parent functions
outedges	Number of child functions

SVM Learning. After extracting features, we use SVM learning to construct the classifier which can recognize parsers. First, we combine all features to construct a feature vector and normalize them to represent each function. Though observation, the feature value is either an integer value that is greater than zero or a boolean variable. As the distribution of feature values is not uniform and has a long tail, the Min-Max scaling transformation is not suitable. We take logarithmic transformation to normalize all feature values.

$$\overline{x_f} = \log_{10} x_f \tag{1}$$

where x_f represents the feature f's value for a given function. $\overline{x_f}$ represents the feature f's value after normalization. Then, we select Standard Support Vector Machine (C-SVM) with linear kernel to train our classifier based on normalized samples. The result of classifier training is presented in Sect. 3.

2.2 Analysis of Security-Sensitive Sinks

After recognizing protocol parsers from firmware, we first design a selection mechanism for security-sensitive sinks. Then, we construct PDG to identify related input source and extract insecure sinks.

Selection of Security-Sensitive Sinks. The security-sensitive sink refers to the common library function which could be affected by malicious external input if the verification of input data is not strict. Since both buffer overflow and command injection belong to taint-style vulnerability, we list their related vulnerable functions as sinks in Table 2.

To accelerate the discovery of the vulnerability, we only analyse sinks in the parser function and its child functions. If a child function is deeper from the parser function, the probability of suffering from vulnerability for the sink is lower as external data usually does not spread too deep. Thus, we limit the scope of our analysis to the parser function itself and its child functions within three layers.

Table 2. Security-sensitive sinks

Category	Functions
Buffer overflow vulnerability	strcpy sprintf strcat memcpy gets fgets getws sscanf strncpy memmove
Command injection vulnerability	system exec execv execl

Program Dependence Graph. PDG is firstly constructed to guarantee precise backtracking to source for sinks. It is an intermediate program representation that makes explicit both the data and control dependence for each operation in a program [13]. Three kinds of directed edges are contained in PDG, and we describe

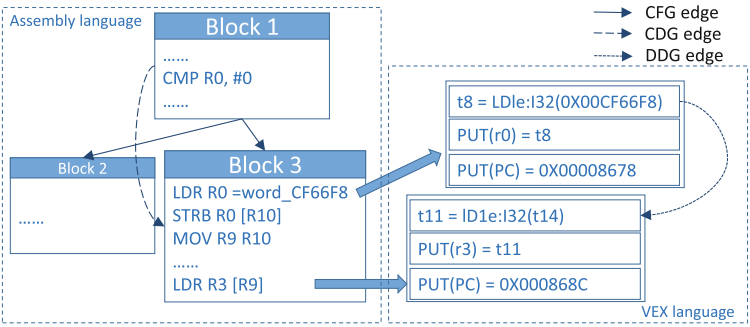

Fig. 1. PDG diagram

them in detail using a diagram of binary sample shown in Fig. 1. First, control flow edge [10] represents the transition between program blocks. Second, control dependance edge [12] refers the relationship that execution result of one specific statement affects execution of the other statement. In Fig. 1, one of control dependence analysis is that execution result of comparison instruction in Block 1 determines execution of all instructions in Block 3. In addition, data dependence edge [13] represents the production and consumption of data in program. Instructions in assembly form are transformed to VEX intermediate language and data dependence is analyzed by construction and inferring of def-use chain [21]. In Fig. 1, one of data dependence analysis is that R0 in first instruction of Block 3 and R3 in last instruction of Block 3 points to the same address. Source of any instruction's argument could be deduced iteratively in this way.

Since most external inputs are located in parser functions, we then construct PDG from the entry point of identified parser functions to all child functions within three layers. Based on PDG, we can identify the input source of sinks. If the type of input source is string constant or from read-only data segments like *.rodata*, we take it as insecure sinks. Finally, it need be deeply analyzed to confirm the existence of vulnerability and the analysis does not belong to the work of this paper.

3 Evaluation

In this section, we first describe the implementation details, then present training result of the classifier. Next, we evaluate the performance of parsers recognition on real-world embedded devices. Finally, the performance of our mechanism is compared with whole-program analysis.

3.1 Implementation

We implement the two-stage mechanism on a server. It is Ubuntu 14.04.2 LTS(GNU/Linux 3.13.0-48-generic x86 64) with Intel Xeon(R) E5-2687W v3 CPU and 125.8 GiB of memory. In the first stage, the main service program

is loaded in The Interactive Disassembler (IDA) [9] which can disassemble the program from binary to assembly form and recover its control flow graph. Since features are unrelated to instruction set architecture and the disassembler code is sufficient for analysis, we write python scripts using programming API supported by IDA to extract feature values. Then we use open source tool scikit-learn [7] to train classifier based on SVM learning. In the second stage, we modify the code of tool Angr [21] to support PDG and analyze the input source of security-sensitive sinks which help quickly find out insecure cases. Finally, We conduct the experiment on firmwares of Hikvision and TVT brand camera. Hikvision [5] and TVT [8] are famous camera manufacturers and their cameras are wildly deployed in numerous applications. We select DS-2CD6223F model for Hikvision camera and TD-9436T model for TVT camera. The main programs from two cameras firmwares are centaurus (13.2 MB) and ipcamera (14.4 MB). The performance of the classifier and two-stage mechanism are evaluated on them.

3.2 Cross Validation for Classifier

In the learning phase of classifier, We label 63 parsers from several firmwares as positive samples, and choose 300 negative samples at random. We divide samples into training set and validation set, respectively are S1 and S2. The experiment starts to take 10 features as candidates and construct an initial classifier on S1. Then the classifier is tested on S2, and the feature with lowest weight would be discarded. The rest of features are kept training, and the result indicates that when candidate features are reduced to specified 5 features, the recognition accuracy rate is the best. The best features combination are *switch*, *com_spec* *com_banner*, *strcmp* and *strstr*. The corresponding weights are 0.93, 1.76, 0.61, 1.88 and 1.71.

Unimportant features are *loop*, *inedge*, *switch_call*, *blocks* and *outedge* sorted by unimportance. The *loop* feature cannot be easily extracted since the broken chain of function calls from IDA perspective. So it appears to be unimportant in recognition of parser functions. Conversely, other features are indeed unimportant for parser recognition. Through our experiment, we find features like *cmp_spec*, *strcmp* and *strstr* may be more significant.

3.3 Performance of Classifier

To evaluate performance of the classifier, we manually label parser functions and normal functions in centaurus and ipcamera. The classifier is applied to them and the parser recognition accuracy is shown in Table 3. The mechanism tolerates imprecision of parsers recognition to a certain extent as it is an intermediate step to accelerate the speed of finding code locations where probability of the taint-style vulnerability existence is high.

3.4 Performance of Two-Stage Mechanism

In whole-program analysis, functional modules are treated equally without discrimination so that the entire security-sensitive sinks are analysed in spite that

Table 3. Recognition accuracy of classifier

Program	Identified parsers	Correct num	Accuracy
centaurus	93	79	84.9%
ipcamera	97	83	85.6%
Total	190	162	85.3%

many of these are free of vulnerability. In our two-stage mechanism, only sinks inside parser functions are analysed which greatly reduce the time cost. To prove the efficiency, we get the number of function analyzed, sinks analyzed and insecure sinks within them for specified sink types in different methods and calculate time cost of each insecure sink discovery.

Table 4. Analysis of two camera firmwares

	ipcamera		centaurus	
	Global analysis	Our method	Global analysis	Our method
Analysis time	2950.8s	252.6s	3007.1s	321.4s
Function num	16053	1174	7232	853
Total (insecure/all)	623/2094	288/655	287/4528	54/427
strcpy (insecure/all)	116/180	22/34	36/92	8/13
memcpy (insecure/all)	352/1328	221/451	141/2347	29/287
strncpy (insecure/all)	20/252	6/73	11/1638	4/39
sscanf (insecure/all)	4/29	4/22	8/67	5/52
sprintf (insecure/all)	120/288	34/73	63/318	7/30
Time per insecure sink	4.74 s	0.88 s	10.48 s	5.85 s

The evaluation result on ipcamera and centaurus program is shown in Table 4. It shows that to ipcamera and centaurus, average time cost of each insecure sink discovery by our method are 0.88 s and 5.85 s. And the corresponding time cost by whole-program analysis are 4.74 s and 10.48 s. By comparison, our mechanism reduce the time cost of insecure sink discovery by 81.4 and 44.2 percent. Therefore, our method has an advantage over accelerating discovery of the taint-style vulnerability in embedded systems.

4 Related Work

4.1 Static Analysis

Yamaguichi et al. [19,20] propose to design code property graph representation and path traversal patterns for various vulnerability types. Unfortunately, code

property graph is depended on source code, hence it cannot be applied directly on embedded binary programs. Angr [21] extracts control flow, control dependence, data dependence graph and discovers vulnerabilities by using backward slicing and symbolic execution technologies. However, accurate analysis still suffers from low efficiency. PIE [11] finds parsers by extracting features and using machine learning. However, only structure features are selected which affects accuracy of protocol parsers recognition and vulnerability analysis work is rare.

4.2 Dynamic Analysis

Dynamic approaches like fuzz testing have been proposed to discover vulnerability more precisely. Codenomicon Defensics [3], a mature network protocol fuzzing product, supports many species of industrial protocols. However, code coverage for testing is usually low and vulnerabilities in uncommon paths are not found. As a remedy, some dynamic analysis tool like Driller [17] is proposed to combine fuzz testing with symbolic execution to guide analysis into specified code areas. Unfortunately, it requires simulation of programs which is difficult to implement in embedded systems. In the aspect of dynamic simulation, Avatar framework [22] is proposed to dynamically analyze embedded systems by orchestrating the execution of an emulator together with the real hardware. Decaf [15] is proposed to support virtual machine based, multi-target, whole-system dynamic binary analysis. However, both frameworks have not been used to directly analyze vulnerabilities.

5 Conclusion

In this paper, we propose a lightweight method to accelerate discovery of taint-style vulnerabilities in embedded systems. Instead of analyzing entire security-sensitive sinks, we focus on protocol parsers which are more prone to taint-style vulnerabilities. We firstly use machine learning technologies to construct a classifier that can recognize parser functions accurately. Then, we derive PDG to identify input source of sinks that can help extract insecure cases quickly. We demonstrate effectiveness of our work by comparing it with whole-program analysis. Our work can effectively help analysts spend less time on insignificant codes and find taint-style vulnerabilities in time. In the future, we need improve protocol parsers recognition model and extend our work to cover more vulnerability types.

Acknowledgments. This work was supported in part by the National Key Research and Development Program (Grant No. 2016YFB0800202), the National Defense Basic Research Program of China (Grant No. JCKY2016602B001), the “Strategic Priority Research Program” of the Chinese Academy of Sciences (Grant No. XDA06040100), and the National Defense Science and Technology Innovation Fund, CAS (Grant No. CXJJ-16M118).

References

1. Binwalk — firmware analysis tool. http://binwalk.org/
2. Cve - common vulnerabilities and exposures (cve). http://www.wooyun.org/
3. Defensics - fuzzing - fuzz testing - black box testing - negative testing — codenomicon. http://www.codenomicon.com/products/defensics/
4. Firmware-mod-kit - google code. https://code.google.com/archive/p/firmware-mod-kit/
5. Hikvision usa. http://www.hikvision.com/
6. Offensive security exploit database archive. https://www.exploit-db.com/
7. scikit-learn: machine learning in python. http://scikit-learn.org/
8. Shenzhen tvt digital technology co., ltd. http://www.tvt.net.cn/
9. Welcome to hex-rays!. https://www.hex-rays.com/index.shtml
10. Allen, F.E.: Control flow analysis. In: ACM Sigplan Notices, vol. 5, pp. 1–19. ACM (1970)
11. Cojocar, L., Zaddach, J., Verdult, R., Bos, H., Francillon, A., Balzarotti, D.: Pie: parser identification in embedded systems. In: Proceedings of the 31st Annual Computer Security Applications Conference, pp. 251–260. ACM (2015)
12. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. (TOPLAS) **13**(4), 451–490 (1991)
13. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) **9**(3), 319–349 (1987)
14. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. In: NDSS, vol. 8, pp. 151–166 (2008)
15. Henderson, A., Prakash, A., Yan, L.K., Hu, X., Wang, X., Zhou, R., Yin, H.: Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 248–258. ACM (2014)
16. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software (2005)
17. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of the Network and Distributed System Security Symposium (2016)
18. Wang, T., Wei, T., Lin, Z., Zou, W.: Intscope: automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In: NDSS. Citeseer (2009)
19. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, pp. 590–604. IEEE (2014)
20. Yamaguchi, F., Maier, A., Gascon, H., Rieck, K.: Automatic inference of search patterns for taint-style vulnerabilities. In: 2015 IEEE Symposium on Security and Privacy, pp. 797–812. IEEE (2015)
21. Shoshitaishvili, Y., Ruoyu Wang, C., Salls, C., Stephens, N., Polino, M., Dutcher, A.: (state of) the art of war: offensive techniques in binary analysis (2016)
22. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D.: Avatar: a framework to support dynamic security analysis of embedded systems' firmwares. In: NDSS (2014)