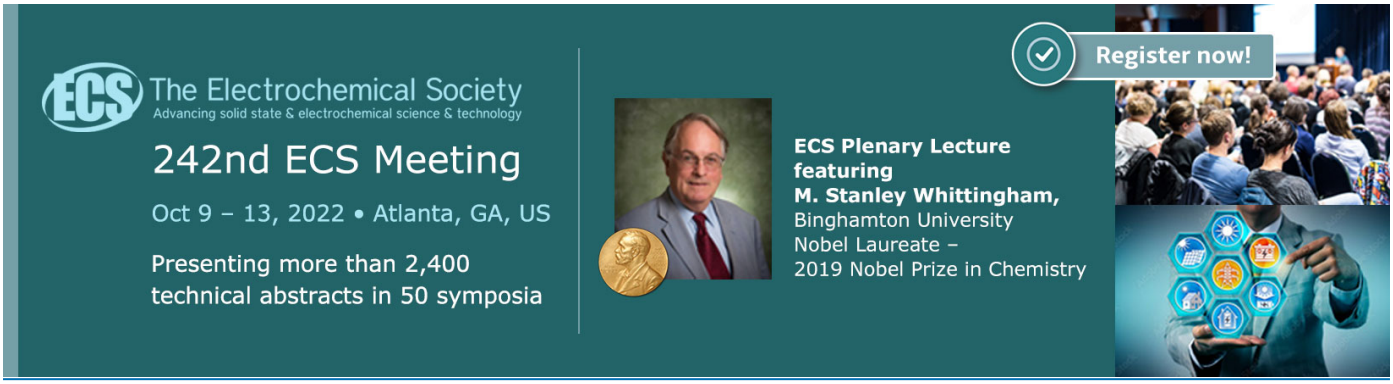
CSChecker : A binary taint-based vulnerability detection method based on static taint analysis



CSChecker： A binary taint-based vulnerability detection method based on static taint analysis

Wei Guo[1b], Qiang Wei[1a*], Qianqiong Wu[1], Zhimin Guo[2c]

[1] Cyberspace Security Academy, University of Information Engineering, Zhengzhou, Henan, 450001, China

[2]Electric Power Research Institute of State Grid Henan Electric Power Company,China

[*a]funnywei@163.com, [b]1037057802@qq.com,[c]zhimin.guo@163.com

Abstract. Embedded devices such as routers not only bring convenience to people's daily life, but also increase the attack surface and security risks of devices. Embedded device applications tend to be closed source and therefore cannot be searched for vulnerabilities through source code audits. Even open source applications can be insecure because they reference third-party libraries. Binary file vulnerability mining is an important means to solve this kind of problem, but it has the problems of path explosion and low efficiency. This article uses the static stain analysis with the method of combining the vulnerability characteristics, in the type of stain into classes and class assignment holes for testing. Based on function call graph, this paper uses atomic combinational optimization to detect the vulnerability of router firmware. The prototype tool -- CSChecker is implemented in D-Link, Tenda, the test was carried out on 267 firmware files of Netgear and other well-known brands, and the experimental results showed that the accuracy of CSChecker in the data set reached 92.51%, indicating that CSChecker could effectively search the injection vulnerabilities and assignment vulnerabilities of binary files.

1. Introduction

As embedded devices become popular in people's daily lives, they bring convenience but also risks such as leakage of sensitive data, especially the security of router devices. Routers, as network hubs, are able to connect most smart devices under the same premises, making them a high-risk target for attacks. Most embedded devices are closed source, which makes them more difficult to attack but also leads to their security relying mainly on a small number of people such as security testers, increasing the likelihood of vulnerabilities in the system and therefore requiring direct detection of the firmware in binary form.

Static taint analysis techniques rely on the possible presence of contaminated data streams in the target object, mainly for weblike applications[1][2] and Android[3][4] applications, as both target objects are characterised by a large number of user interactions, a large number of risk points brought about by complex components and the availability of part or all of the source code. The detection of Web-based applications is mainly for their scripting languages, such as PHP[5] and JavaScript[6]; the detection of Android applications aims at protect whether sensitive information[7] such as user privacy is stolen, for example, B-Droid[8] uses static taint analysis combined with fuzzy testing to

detect privacy leaks by analysing the tested application's behaviour to detect privacy leaks, which can effectively detect the five most popular types of commercial spyware in the market.

Static detection of objects is generally low-level languages[9], and large-scale data stream detection increases the complexity of the model, leading to an increase in detection time and a decrease in accuracy. In addition, static detection technology also has some problems such as path explosion and data stream insensitivity. In this paper, we use an atomic assembly-based approach to avoid the path explosion problem and improve the detection efficiency and accuracy by detecting the disassembled C language.

The contributions of this paper are shown below

1. This paper uses the method of function atomic pooling to limit the path increase within a function, effectively avoiding the path explosion problem in the process of taint analysis.
2. This paper uses a combination of taint analysis and vulnerability features to detect tainted vulnerabilities, and optimizes the detection paths based on function call graphs by atomic combination to directly obtain reachable paths.
3. This paper designs and implements a prototype tool for binary vulnerability detection, CSChecker, for identifying injection-type and assignment-type vulnerabilities in router firmware and binary dynamic link library files, with an accuracy rate of 92.51% for vulnerabilities.

The paper is organised as follows. Section II describes the preliminaries of the paper; Section III details the specific ideas of the method used in this paper and the algorithms implemented in each section; Section IV describes the performance tests of CSChecker on a dataset and provides a comparative analysis of its experimental results; and finally discusses the limitations of the method used in CSChecker and the direction of effort for subsequent work.

As a network hub, the router usually connects all the smart components in the same scenario, making its security even more noteworthy. Similar to the two applications mentioned above, router systems also have web interfaces that interact with the user, such as login and configuration pages. In addition, the UPNP protocol, which was introduced for the sake of convenience, has been uncovered in recent years. This paper researches the percentage of vulnerability types of popular router brands such as Tenda, Netgear and D-Link from 2019 to the present on the CVE website, and the findings are shown in Table 1.

Table 1. Vulnerability type statistics

Router Name	Assignment ratio	Injected ratio
D-Link	11.54%	31.2%
Tenda	51.28%	17.95%
Netgear	13.99%	28.78%
TP-Link	15.38%	19.23%
Cicso	3.26%	43.83%

The average percentage of assignment-based vulnerabilities and injection-based vulnerabilities reached 45.68%, and they fit well with the static taint analysis-based approach proposed in this paper, so they were used as the main targets of the method.

2. Method

The method used in this paper is divided into three steps: Building the atomic pool, path network diagram optimisation, and vulnerability establishment determination.

2.1 Building the atomic pool

Using the function as the subject, the call relationship of the function, the contaminated variables within the function and the contamination of the parameters of the called function are obtained. This step is designed to perform complexity degradation on functions, treating them as atomic nodes with attributes and simplifying the complexity of constructing contamination paths.

The collected binaries are disassembled using IDA Pro. Within the function, a contamination pool is first set up, with initial contamination sources being function arguments and external inputs such as nvram_bufget, get_var, websGetVar. The internal code path of the function is traversed, variables that are directly or indirectly contaminated are collected, a list of called functions is obtained and the contamination of these function arguments is recorded. As shown in algorithm 1.

Algorithm 1. Atomic pool construction algorithm

```
Input:   FunctionArgs，InputVal                    //List of function parameters，List of external inputs
         NSList，TFH                               //List of called functions，handle of the target function object
OutPut：Result_list = []
def TaintAnalysis():
   ValPools = FunctionArgs + InputVal              //Storing contaminated variables
   while TFH.isEnd()：                             //Ends the loop if it reaches the end of the function
      CodeArgs = JudgeAssignment(TFH)              //AType refers to the assignment method and is assigned the value 1
                                                   //CodeArgs assignment variables and assigned variables
      if (AType == 1) and (CodeArgs[1] in ValPools) and (CodeArgs[0] not in ValPools):
                                                   //If it is an assignment operation and contamination occurs
         ValPools.append(CodeArgs[0])              //Storing the assigned variable
         TFH.start()                               //Restart testing
         continue
      else：
         TFH.next()                                //If there is no assignment operation, run down
         continue
   for each_section in NSList:                     //Detects if the parameters of the next function are contaminated
      midArgs = GetArgs(TFH.each_section)          //Get the parameters of the called function
      midflag = JudgeTaint(midArgs,ValPools)       //Detects if the parameters of the called function are contaminated
      if not midflag：
         Result_list.append(each_section)
   return
```

The extracted atomic attributes are then stored using a mongoDB database.

2.2 Path network diagram optimisation

This step involves constructing a complete propagation path graph and optimising it. First, the start and end points are set; then all paths from the start point to the end point are constructed; finally, the paths are optimised using the atomic properties in the atomic pool, and unusable paths are removed.

The start point is the function that directly calls the input function. The end point is the function that calls the function for the end point that matches the vulnerability profile. The path network graph is constructed using the breadth-first algorithm to traverse all paths from the beginning to the end. The traversal algorithm is shown in Algorithm 2.

Algorithm 2. Breadth-first path network graph traversal algorithm

```
Input:   RouteList                                  //Record a list of paths
         FunctionRelation                           //Function call relationships, dictionaries
            VulFunctionList                         //List of dangerous functions, endpoints
Output:  Result_list = []                           //Record all paths to the end point
def create_all_route(RouteList):
   mid_list = []
   for eRoute in RouteList:
     nSection = FunctionRelation[eRoute[-1]]        //The set of nodes next to a particular path
        mid_val = VulFunctionList ∩ nSection        //The endpoint function present in the next node set
        If mid_val:
          SaveTheRoute(Result_list,eRoute,mid_val)  //If there is a path to the end, record it
         mid_list+=AddFunction(eRoute,nSection,mid_val)  //Add the next node to the end of the path
   create_all_route(mid_list)
   return
```

The path network diagram constructed according to algorithm 2 is shown in Fig. 1.

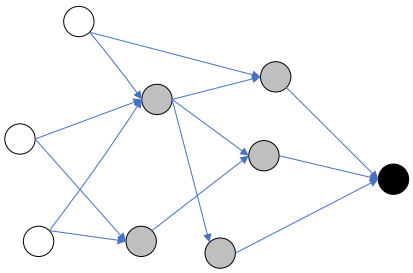


Fig. 1 Path network diagram

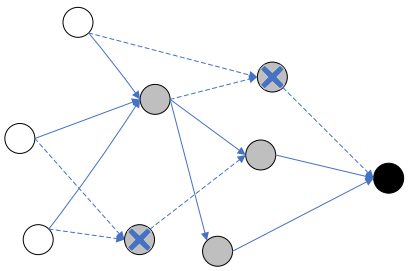


Fig. 2 Taint analysis diagram

All the nodes represented by circles in Fig. 1 are functions, with each node connected by a directed arrow, the starting point of which is the function calling function and the end point of which is the called function; the blank circle on the left is the function that can accept external input, the black solid circle on the right is the hazard function and the grey circle in the middle is the tainted propagation function.

The path network diagram contains many unusable paths, for example where the parameters of the called function are not contaminated or not contaminated at all, so that the input data cannot pass through this node and therefore such nodes are deleted. The basis for node deletion is the contamination of the called function's parameters in the atomic attributes. The optimisation process is shown in Fig. 2 and Fig. 3.

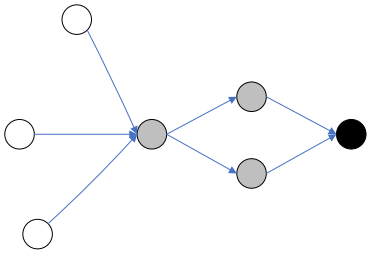


Fig. 3 Path network diagram after optimization

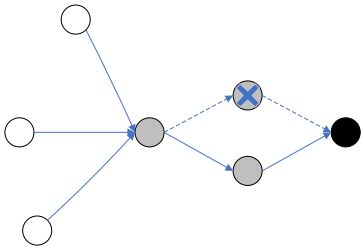


Fig. 4 Vulnerability point establishment determination diagram

2.3 Vulnerability establishment determination

So far an optimised tainted transport path has been obtained and the external external input has reached the dangerous function call function correctly. This step therefore determines whether the parameters of the dangerous function are controllable and whether the parameter characteristics are those of the injection and overflow vulnerabilities, and if so, the call point is considered to be a potential vulnerability.

Algorithm 3 Vulnerability Establishment Determination Algorithm

```
Input:  SrcArgs , DstArgs                          //Source variable, destination variable
OutPut:  True/False                                //Determine if the result is valid
def VulJudge():
   JudgeResult = JudgeTaint(SrcArgs)               //Determining whether source variable is contaminated
   VulType ==>Assignment Vulnerability
        JudgeResult1 = JudgeTaint(DstArgs)         //Determine if the destination variable points to the stack space
        if JudgeResult and JudgeResult1:
           CheckResult = CheckArgs(SrcArgs,DstArgs) //Checking the length of variables
                Set up==>return True
                Don't set up==>return False
 VulType ==>Injection Vulnerability
        JudgeResult2 = JudgeType(SrcArgs)
        JudgeResult3 = JudgeLength(SrcArgs)        //Checking variable types and length limits
        if JudgeResul2 and JudgeResult3:
             Set up==>return True
        Don't set up==>return False
   Return False
```

As shown in Fig. 4, for fragile points that are judged to be untenable, the paths for which the call relationship exists are removed, the group of call relationships that generally exist with the end of the path, and the paths that are judged to be valid are then saved to the database.

3. Experiment

This section describes the various aspects of the CSchecker performance evaluation experiments, including the construction of the dataset, the experiments and the analysis of the experimental results.

The hardware environment for this experiment is a laptop with Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz 1.61 GHz, 16GB memory; the software environment is a Windows 10 system with IDA Pro 7.5 and python 3.7 installed. To facilitate data flow analysis, the FIDL framework was installed for IDA Pro. A MongoDB V4.2 database was selected for the storage of vulnerability information and data.

3.1 Dataset construction

This experiment constructs a vulnerability dataset with a data volume of 267, which consists of popular router firmware as well as dynamic link libraries, and stores the data entries in the form of (CVE number, firmware name, file name, vulnerability address) into a database.

The first step is to obtain the vulnerability information. The experiments were conducted to obtain the vulnerability information of all data set objects in the CVE website, filtering out injection and assignment vulnerabilities based on the CWE number and keywords in the CVE description information, such as 'overflow', 'command injection', etc.

The second step is to manually obtain the address of the vulnerability point. As the vulnerability information in the CVE website varies greatly, and there are many non-existent vulnerability file download links, so the experiments use a manual way to download the existence of vulnerability files, and then use IDA pro to determine whether there is a vulnerability point described in the vulnerability description information, and if it exists, store its point information, and finally stored in the database.

3.2 Performance tests

The target files are placed in the same directory and files with the same filename are named differently. Then run the program and batch the target files using IDA Python. Each target file will output individual analysis results and compare these results with the vulnerability points collected in the first step to determine if they are hits or not, thus calculating CSChecker's accuracy and redundancy rates. The relevant results are calculated as shown below.

$$HitRate = (HitPoints/TotlePoints) \times 100\% \quad (1)$$

The above equation is the formula for the accuracy rate, where HitRate indicates the accuracy rate, HitPoints indicates the number of hit vulnerability points, and TotlePoints indicates the number of all vulnerability points in the dataset.

$$ReRate = \frac{\sum_{i\in HitPoints}^{i=1} 1-(HitRate)}{HitPoints} \times 100\% \quad (2)$$

The above formula is how the redundancy rate is calculated, where ReRate, which is RedundancyRate, represents the redundancy rate, HitRate represents the accuracy rate and HitPoints represents the number of hit vulnerability points.

3.3 Analysis of experimental results

After experimenting with the above steps, the output of the CSChecker tool is as follows.

Table 2 Table of results of CSChecker experiments

Router Name	Total Vulnerabilities	Accuracy Rate	Redundancy Rate
D-Link	43	95.35%	4.43%
Tenda	26	92.31%	7.10%
Netgear	140	92.86%	6.63%
TP-Link	29	93.1%	6.24%
Cicso	29	86.21%	11.89%

As can be seen from the above table, the vulnerability accuracy of CSChecker in the five common router brands in the dataset ranges from 86.21% to 95.35%, with average accuracy of 92.51%; the hit redundancy rate does not exceed 11%. The experimental results show that CSChecker can effectively detect injection-type vulnerabilities and overflow-type vulnerabilities in router firmware.

```
1  v40 = a2;
2  strcpy(v80, v79);
3  memcpy(a1, a2, a3);
4  strncpy(v12, v32, v10);
5  v3 = nvram_bufget(0, "portal_manage");
6  ...
```

Figure 5. Example diagram of direct/indirect assignment

```
1  valueString((int)v10, v7, v8);
2  v7 = get_cgdata_by_uid(v20, v28);
3  ...
```

Figure 6. Example diagram of user-defined function pollution

Although the accuracy rate of CSChecker is high, there are still undetected vulnerability points for two main reasons. Firstly, there are still some shortcomings in the judgement of tainted variables within functions; the detection of direct and indirect assignments shown in Figure 5 is good, but there is still room for improvement in the detection of user-defined function taint shown in Figure 6. In addition to this, there is also the problem that some of the firmware functions are inclined which prevents the FIDL framework from correctly identifying functions within functions, leading to biased analysis of taint propagation.

4. Conclusion

This paper uses a combination of taint analysis and vulnerability characterisation to detect injection and assignment class vulnerabilities, and optimises the detection path using atomic attributes, as a way to detect vulnerabilities in router firmware. It was tested on 267 publicly available vulnerability points of popular brands such as D-Link and Tenda, with an accuracy of 92.51%. The experiments proved that the method is effective in detecting injection and assignment class vulnerabilities in binaries such as router firmware, and avoids problems such as path explosion.

Although the method is effective in detecting known injection and assignment vulnerabilities in router firmware, there are still areas for improvement, such as not being able to determine the full size of the memory referred to by the target variable for assignment vulnerabilities, leading to false positives. In addition, there is still room for improvement in the identification of the contamination capabilities of user-defined partial user-defined functions. These are all issues that need to be addressed in the next step of the work.

Acknowledgments

This work was financially supported by the Foundation strengthening program projects (2021JCJQZD06311).

References

[1] Maskur, A.F., and Yudistira D.W.A. (2019) Static Code Analysis Tools with the Taint Analysis Method for Detecting Web Application Vulnerability. In 2019 International Conference on Data and Software Engineering (ICoDSE), pp. 1-6.
[2] Saoji, T., Thomas H.A, and Cormac F. (2017) Using precise taint tracking for auto-sanitization. In Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, pp. 15-24.
[3] Zhang, J.B., Wang Y.Y., Qiu L.N., and Julia R. (2021) Analyzing android taint analysis tools: FlowDroid, Amandroid, and DroidSafe. In IEEE Transactions on Software Engineering.
[4] Xiong, B., Xiang G.L., Du T.L., He J.S., and Ji S.L. (2017) Static taint analysis method for intent injection vulnerability in android applications. In International Symposium on Cyberspace Safety and Security, pp. 16-31 .
[5] Yan, X.X., Wang Q.X., and Ma H.T. (2017) Path sensitive static analysis of taint-style vulnerabilities in PHP code. In 2017 IEEE 17th International Conference on Communication Technology (ICCT), pp. 1382-1386.
[6] Almashfi, N., and Lu L.J. (2019) Static Taint Analysis for JavaScript Programs. In International Conference on Tools and Methods for Program Analysis, pp. 155-167.
[7] Ferrara, P., Luca O., and Fausto S. (2018) Tailoring taint analysis to GDPR. In Annual Privacy Forum, pp. 63-76.
[8] ALmotairy, R., and Yassine D. "B-droid: A Static Taint Analysis Framework for Android Applications."
[9] Feng, C., and Zhang X. (2017) A static taint detection method for stack overflow vulnerabilities in binaries. In 2017 4th International Conference on Information Science and Control Engineering (ICISCE), pp. 110-114.