FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master′s Thesis in Informatics

Automated Detection, Localization and Removal of Information Exposure Errors

Kommanapalli Vasantha

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Automated Detection, Localization and Removal of Information Exposure Errors

Automatisches Erkennen, Lokalisieren und Entfernen von ungewollter Informationsfreigabe

Author:	Kommanapalli Vasantha
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	M. Sc. Paul Muntean
Date:	November 15, 2015

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 6. August 2015 Kommanapalli Vasantha

Acknowledgments

If someone contributed to the thesis... might be good to thank them here.

Abstract

Automatically detecting, localizing and fixing information errors is very much needed in current generation as people are not ready to spend much on their time in debugging and fixing the errors. In general debugging requires a lot of time and effort. Even if the bug's root cause is known, finding the bug and fixing it is a tedious task. The information that is exposed may be very valuable information like passwords or any information that is used for launching many deadly attacks. In order to fix the information exposure bug we should refactor the code in such a way that the attacks or information exposure can be restricted.

The main objective of this master thesis is to develop a quick fix generation tool for information exposure bugs. Based on the available information exposure checker which detects errors in the open source Juliet test cases: CWE-526, CWE-534 and CWE-535 a new quick fix tool for the removal of these bugs should be developed. The bug location and the bug fix location can be different (code lines) in a buggy program. Thus, there is a need for developing a bug quick fix localization algorithm based on software bug fix localization techniques. It should help to determine the code location where the quick fix should be inserted. Based on the bug location the developed algorithm should indicate where the quick fix should be inserted in the program. We can consider a bug fix to be a valid fix if it is able to remove the confidential parameter inside a function call to a system trust boundary. After the quick fix location was determined and the format of the quick fix was chosen then the quick fix will be inserted in the program with the help of the Eclipse CDT/LTK API.

The effectiveness of the implemented code refactoring is checked by re-running the information flow checker on the above mentioned open source Juliet test cases. If the checker detects no bug then the code patches are considered to be valid. The generated patches are syntactically correct, can be semi-automatically inserted into code and do not need additional human refinement. The generated patches should be correct and sound.

Contents

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION
This chapter presents an overview of the thesis and it purpose. Furthermore, it will discuss the sense of life in a very general approach.

CHAPTER 2: TECHNICAL AND SCIENTIFIC FUNDAMENTALS
No thesis without theory.

CHAPTER 3: RELATED WORK
write about related work.

Part II: Design and Development

CHAPTER 4: OVERVIEW
This chapter presents the requirements for the process.

Part III: Evaluation and Discussion

CHAPTER 5: EVALUATION
This chapter presents the requirements for the process.

CHAPTER 6: DISCUSSION
This chapter presents the requirements for the process.

CHAPTER 7: CONCLUSION
This chapter presents the requirements for the process.

Part I.

Introduction and Theory

1. Introduction

"The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards." — ***Gene Spafford***

According to the ultimate security vulnerabiltiy datasource CVE details [3], there are 2082 vulnerabilities related to information exposure. Information exposure bugs could be introduced in different stages of software development. During design, architecture or coding phase and can lead to exposure of critical information or can also lead to strange program behavior. Recently, Forbes 2015 [1] published in one of the articles that among the topmost data breaches occurred in the previous years Neiman Marcus hack is famous. In January 2013, many debit and credit card information of almost 350,000 customers have been hacked. It is believed that the breach happened because of a malicious software that was installed onto the Neiman Marcus system. This software collected all the payment card information from customers who purchased. This proves that the software has helped the attackers to leak all the sensitive information through which they could get access to the system without leaving any trace of hacking. Sensitive information can be leaked in many ways [2]. It can be through the environmental variables which contain the sensitive information about any remote server, through a log file that was used while debugging the application where access to that file was not restricted or through a command shell error message which indicates that the web application code has some unhandled exception. In the last case, the attacker can take advantage of that error causing condition in order to gain access to the system without authorisation.

The goal of this thesis is to develop an algorithm for fault localization(localizer) and also to repair the information exposure buggy programs. Repairing is done using the precise information like failure detection, bug diagnosis, buggy variables which are nothing but the program variables that directly influence the appearance of a bug in the program. For example the buggy variable "line" reported by the information exposure checker [ref]. In order to repair the buggy program, failure detection and bug diagnosis data has been used to generate the quick fixes for information exposure bugs with the help of a refactoring wizard. We have developed a localization algorithm in order to localize the bug. We need a localizer since the cause for the bug will not be the place where the bug was detected but earlier in the program code where the information flows into the buggy variable. Therefore a novel algorithm is developed in order to detect possible insertion locations for the generated code patches. Here the code patches can be inserted at two different locations: (i) place where the bug was found —"in-place-fix", (ii) place where the information flows in the buggy variable —"not-in-place-fix". Approach for generating the program repair depends on the code patch patterns, SMT solving and searching possible quick fix locations searching in the program execution paths that will not affect the program behavior by inserting the patch at "not-in-place" location. Generated patches do not change the program behavior for the input that doesnot trigger the bug, therefore the generated code patch is sound. It doesnot need further human refinement(final) , no alien code(human readable), syntactically correct and compilable. The defect class that have been addressed here is information exposure through log files , error reports and environmental variables exposure through which sensitive information like the password or the path to remote server could be exposed and the hacker could exploit the system. The fix defect class consists of the removal of the confidential data at the point where the information flows from the trust boundary based on semi-defined patch patterns.The aim of the quick-fix code patch is information exposure error mitigation (e.g., to prevent that an attacker exploits the error in order to gain system access or display of sensitive information). Program repair depends on two dimensions: (i) an oracle which decides upon what is incorrect in order to detect the bug, (ii) another oracle to decide what should be kept correct in order to attain software correctness [ref]. Patches are generated automatically and inserted semi-automatically offline with the possibility to insert them also online. 90 C programs of Juliet test suite CWE-526, CWE-534, CWE-535 [2] have been used to evaluate the developed approach. CWE-526 contains information exposure through environmental variables related bugs and the potential mitigation would be to protect the information stored in the environmental from being exposed to the user. CWE-534 contains information exposure through debug log files related bugs and the potential mitigation would be to remove the debug files before deploying the application into produciton. CWE-535 contains information exposure through shell error message related bugs . In all the three CWE's the common consequence of the bug is loss of confidentiality.

1.1. Information Exposure Bug

Information Exposure Bug is an error in the software code which intentionally or unintentionally discloses sensitive information to an user who is not explicitly authorized to have access rights to that information. This information could be sensitive within the developed product's own functionality like a private message, which provides information about the product itself or which exposes the environment that is not available for the attacker and that could be very useful for the attacker like the installation path of that product which can be accessed remotely. Many of the information exposures is due to many of the program errors like the PHP script error that may reveal the path of the program or could be some timing discrepancies in encrypting the data. There are in general many kinds of problems that involve information exposures and the severity of those problems can range vastly based on the type of the sensitive information that has been revealed by the errors. Information exposures are also named as information leak or information disclosure.

1.2. Motivation with Example

In general detecting a information exposure bug depends on accurately finding the source code location where some sensitive information leaves the defined trust- boundary. This is the point where an attacker can exploit this IE vulnerability where sensitive information is leaked out. As already mentioned this information could be any confidential data. So in order to make sure that no sensitive information leaks out , a tool is developed in order to restrict the flow of information outside of the system's trust boundary at the location where the bug was found. In this section a real-world bug fix is presented as an example in order to depict that generating a code patch is not that easy. One needs an insight into the program's functionality and also the kind of bug they are dealing with. Normally, a bug can be fixed in many ways with functionally correct patches. Some of the generated patches may change the program behavior. So care must be take that the program behavior is not changed after the patch insertion.

In this section we present two real-world bug fixes as an example to highlight the fact that bug patch generation is not a trivial task. It needs deep insights into the functionality of the program and merits further study. There are typically an endless number of programs who adhere to a formal specification. As such, a bug can be fixed with infinite number of functionally correct patches. The automatically generated patches will change the behavior of the program or not. We present two distinctive patches depicted in listing 1 on lines 5â6 and 11â13 with â+â and by using an italic font. Note, that these two fixes do not change program behavior for program input which does not trigger the bug. Listing 1 contains on line 6 code comments we present other possible quick fixes usable to remove the buffer overflow bug located at line 12 which most likely will change program behavior.

Listing 1.1: CWE-534 test programs source

```
1
2  if(STATIC_CONST_TRUE)
3      {
4          {
5              char password[100] = "";
6              size_t passwordLen = 0;
7              HANDLE pHandle;
8              char * username = "User";
9              char * domain = "Domain";
10             FILE * pFile = fopen("debug.txt", "a+");
11             if (fgets(password, 100, stdin) == NULL)
12             {
13                 printLine("fgets() failed");
14                 /* Restore NUL terminator if fgets fails */
15                 password[0] = '\0';
16             }
17             /* Remove the carriage return from the string that is inserted
                  by fgets() */
18             passwordLen = strlen(password);
19             if (passwordLen > 0)
20             {
21                 password[passwordLen-1] = '\0';
22             }
23             /* Use the password in LogonUser() to establish that it is
                  "sensitive" */
24             if (LogonUserA(
25                         username,
26                         domain,
27                         password,
28                         LOGON32_LOGON_NETWORK,
29                         LOGON32_PROVIDER_DEFAULT,
                  &pHandle) != 0)
         {
            printLine("User logged in successfully.");
            CloseHandle(pHandle);
         }
         else
         {
            printLine("Unable to login.");
         }
         /* FLAW: Write sensitive data to the log */
    - fprintf(pFile, "User attempted access with password: %s\n",
        password);
    /*Source code patch(Quick fix)*/
     + fprintf(pFile,"User attempted access with password:");
         if (pFile)
         {
            fclose(pFile);
         }
      }
   }
```

Listing 1.2: CWE-526 test programs source

```

void CWE526_bad(){
     if (staticFive == 5){
        /*FLAW:environment variable exposed*/
        - printLine(getenv("PATH"));
         /*Source code patch(Quick fix)*/
          + printLine(getenv(" "));
         }
              }
```

Listing 1.3: CWE-526 test programs sink

```
void printLine (const char *line){
          if(line != NULL){
               printf("%s\n", line);
                       }
                    }
```

1.3. Security Hazards

securitx hazards of info expo bugs

1.4. Basic Terminologies

Terminologies

1.5. Contribution

Contribution

2. Technical and Scientific Fundamentals

2.1. Information Exposure Bug with examples

about info expo bug.

2.2. Different Attacks

Write about motivating example.

2.3. Analysis Techniques

like data flow and control flow

2.4. Program Representation

Terminologies

2.5. Code Transformation

Contribution

2.6. Analysis Methods

like static, dynamic or combined

3. Related Work

Part II.

Design and Development

4. Localization Algorithm

5. Approach

5.1. About the Tool

Contribution

5.2. Bug Detection with SMT

Contribution

5.3. Semi Automated Patch insertion Wizard

about info expo bug.

6. Implementation

Part III.

Evaluation and Discussion

7. Evaluation

7.1. Test Setup

Setup configurations

7.2. Methodology

Methodology followed

7.3. Correctness validation

all the validations

7.4. Efficiency and Overhead

about the tool Efficiency.

7.5. Program Behaviour

about the program Behaviour

7.6. Usefulness of the generated Patches

Usefulness of the generated patches.

8. Discussion

8.1. Limitations

limitations of the tool and approach.

8.2. Applications

Where can this be applied.

8.3. Future Work

about the Future work.

9. Summary and Conclusion

Appendix

A. Detailed Descriptions

Here come the details that are not supposed to be in the regular text.

Bibliography

[1] *http://www.forbes.com/sites/moneybuilder/2015/01/13/the-big-data-breaches-of-2014/*.

[2] CWE. *https://cwe.mitre.org/*.

[3] *http://www.cvedetails.com/cwe-definitions/1/orderbyvulnerabilities.html?order=2&trc=668&sha=0427874cc45*
jan 2015.