České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Skoumal Ondřej

Studijní program: Otevřená informatika
Obor: Softwarové systémy

Název tématu: Objevování bezpečnostních chyb ve zdrojovém kódu za použití dotazů v grafových databázích

Pokyny pro vypracování:

1) Popsat jaké jsou současné možnosti analýzy zdrojového kódu.
2) Analyzovat nástroj Joern, naučit se jak ho používat a experimentovat s ním.
3) Vytvořit dotazy v Joernu k nalezení již známých bezpečnostních chyb v open-source projektech.
4) Zobecnit dotazy z 3. z cílem najít zatím neznámé bezpečnostní chyby.
5) Navrhnout nástroj který využije zobecněné dotazy k hledání bezpečnostních chyb.
6) Implementovat nástoj navržený v 5.
7) Analyzovat řešení, popsat výhody a omezení, demonstrovat jeho užití na příkladu.

Seznam odborné literatury:

[1]Joern presentation on CCC 2014 - slides: https://user.informatik.uni-goettingen.de/~fyamagu/pdfs/2014-ccc.pdf
[2]Joern presentation on CCC 2014 - video: https://www.youtube.com/watch?v=291hpUE5-3g
[3]Joern web site: http://www.mlsec.org/joern/

Vedoucí: Ing. Tomáš Černý, Ph.D.

Platnost zadání do konce letního semestru 2017/2018

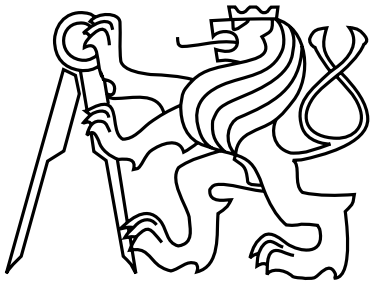
L.S.

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 16.1.2017

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Bachelor's Project

Discovering Security Bugs in Source Code using Graph Database Queries

Ondřej Skoumal

Supervisor: Ing. Tomáš Černý, Ph.D.

Study Programme: Open Informatics, Bachelor's degree

Field of Study: Software Systems

May 26, 2017

Aknowledgements

I would like to thank my technical supervisor Ing. Stanislav Láznička for his friendly attitude, support, guidance and advises, my supervisor Ing. Tomáš Černý, Phd. for guidance and advises, my former technical supervisor Ing. Petr Špaček for guidance and interesting topic to work on, and to my family and all other people who supported me during studies.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on May 5, 2017 .....

Abstract

Finding a security vulnerability in a source code is not an easy task. This thesis aims to explore a new way to find them - to use open-source tool the Joern and modern graph databases. Joern allows to analyze C/C++ codebases and store information about the code in the Graph database. We can then query this database in various ways, e.g. to "find all places where unsanitized input data are used as argument in a function call". This kind of quering of Graph database to find the vulnerabilities can be very efficient and this thesis explores ways how to write the queries to find already known security bugs, how to generalize such queries so they can be used to find yet unknown security vulnerabilities, and finally to utilize simple tool to use these generalized queries for the vulnerability search.

Keywords: Vulnerability, Graph Database, Code analysis

Abstrakt

Najít bezpečnostní chybu ve zdrojovém kódu není jednoduchý úkol. Tato práce si klade za cíl prozkoumat nový způsob jejich nalezení - použitím open-source nástroje Joern a grafových databází. Joern umožňuje analyzovat zdrojové kódy C/C++ a ukládat informace popisující kód do grafové databáze. Do této databáze se pak můžeme dotazovat různými zbůsoby, např. "najděte všechna místa, kde jsou nekontrolovaná vstupní data použita jako argument ve volání funkce". Toto dotazování za účelem hledání chyb může být velice efektivní a tato práce ukazuje způsoby, jak napsat dotazy k nalezení již známých bezpečnostních chyb, jak tyto dotazy zobecnit, abychom je mohli použít k nalezení neznámých bezpečnostních chyb, a nakonec ukazuje použití jednoduchého nástroje, který tyto zobecněné dotazy používá k vyhledávání chyb.

Klíčová slova: Bezpečnostní Chyba, Grafová Databáze, Analýza kódu

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

As it is said in the abstract, finding the security vulnerabilities in the source code is not an easy task, and as said Fabian Yamaguchi[1] during his presentation of Joern on hackativity, "Vulnerability Discovery is the art of navigating inside piles of junk" [10].

That may be the first feeling that approaches your mind when you are tasked to find any vulnerability in the new codebase [10]. Even today in the Information age, when software systems are so crucial and so spread that they can be found everywhere around us, security experts are still those who finds most of the severe vulnerabilities. One of these, the well known *Heartbleed* vulnerability, [7] was found in the cryptographic library OpenSSL in April 2014. The reason why there was the severe security hole in the system was a simple missing sanity check in the code. Unfortunately, the attackers found this security vulnerability earlier than the security experts and used it to read sensitive information [8].

Such example shows that as we as humans rely more and more on the software systems as a whole, the more we need to have these systems *secured*. But because the mentioned security experts are the only ones who mostly audit the source code [8], the information technologies maybe did not advanced so much in this way.

1.1 Motivation

This thesis tries to expand the possibilities of such security experts. In the following chapters, it will try to prove that such severe gap in the security as the *Heartbleed* vulnerability can be relatively easily found by Joern platform [8].

Moreover, maybe it can be as far as that tools based or alike Joern platform [2] can create a little revolution in code reviewing and that the methods of finding security vulnerabilities will hopefully move a little bit forward to the world of the more secure software systems.

Furthermore, Joern platform for discovering the vulnerabilities presented interesting results in several tech talks and papers,[3] and therefore has proven to be interesting.

[1] Dr. Fabian Yamaguchi is the creator of Joern, the program on which this thesis is based on.
[2] The exact nature of Joern platform and its connection to this thesis will be discussed in next chapter.
[3] E.g. [10][5][4].

1.2 Contribution

The chief aim of this thesis is oriented in the practical way to describe methods used to create an example set of the generalized graph database queries, able to uncover yet unknown security vulnerabilities. It describes the creation of the graph database queries able to find known bugs, and these queries are then generalized to be used on any codebase, with the aim of finding yet unknown security bugs.

Also this thesis presents an set of example generalized graph database queries, usable to find yet unknown security bugs.

Thesis also describes the usage and implementation of the tool which is based on the Joern platform. It uses set of these generalized queries to test various codebases for possible bugs.

1.3 Structure

Chapter 2 describes current state of the source code analysis, its usage, methods and also drawbacks.

Chapter 3 describes the Robust code analysis platform Joern. It starts with Joern fundamentals and describes how Joern parses the source code, which data structure it uses for storing the information about the code, how is this data structure saved in the database and finally how can be this database queried to find the bugs.

Chapter 4 offers description of the graph database queries, how they are created to find known vulnerabilities, how then can be generalized and used to find yet unknown vulnerabilities, describes creation of the more complex queries and finally difficulties encountered during experimenting with the Joern tool.

Chapter 5 introduces the code testing tool Code-tester, describes its architecture and design, details of the implementation, how it can be used, options of the tool and finally shows its use in the source code analysis of the VLC media player.

Chapter 6 provides summary of work, discusses the efficiency, advantages and limitations of the provided tool for the source code analysis and contains thoughts about the possible future work.

Chapter 2

The Source code analysis

The source code analysis[1] is usually the part of the Systems development life-cycle, and is performed during the software implementation. The static source code analysis can mean the automated static analysis, manual code auditing or both.

2.1 Reviewing of the source code

To provide complete security for the large software projects of enterprise level may be impossible task. In the strict mathematical sense, the humans simply cannot compute all the possible scenarios and dangerous situations which are defined by the complexity of the code and it's properties, or create tools which would accomplish a task. Although the efforts for such complete analysis are made, as is discussed in the next section, in real world testing, there are used mostly lightweight automatic source code analysis tools and manual code reviewing by the auditors [18].

As said in the introduction, the most of the critical security vulnerabilities are found by manually auditing the source code by the security experts. The auditing itself is not an easy task. Because of this, a new methods are often proposed and welcomed to make this difficult job not so painful. These methods are using detail static code analysis[2], and requires access to the source code, likewise the security experts [8]. This approach is called the static code analysis[3].

The automatic static source code analysis is likely the first mean of the code review. After it, the dynamic code analysis likely follows. Dynamic analysis further tests the program during the execution. One of the popular dynamic analysis tools is Valgrind, the instrumentation framework for building the dynamic analysis tools.

[1] Correct terms are also the Static source code analysis, the Source code analysis or the Static analysis, all these terms are describing the same procedure.

[2] Analysis of the source code is either static or dynamic. Because Joern evaluates code without the need to execute it, this thesis is concerned only with static source code analysis.

[3] The Joern tools also provides this successful but harsh approach of the static source code analysis, and provides whole platform for the code analysis. Joern tool is aimed at getting analysis easier for the auditors.

2.2 The security vulnerabilities

Simply put the security vulnerabilities in the source code are the parts of the code which an attacker can exploit. As well known from media and frequent patches notifications, the security weakness in the code can be destructive.

There are many types of the security vulnerabilities, most of them can be divided into two main categories, the vulnerabilities in the implementation, and vulnerabilities and flaws in the design [18]. As the Joern tool cannot expose design errors and flaws, this thesis is concerned only with vulnerabilities in the implementation.

The ordinary kinds of the implementation vulnerabilities include:

- The input validation, or the insecure arguments, when the user input is not checked or sanitized before used as an argument or processed in any other way. This is a very common vulnerability type and this kind of vulnerability can be found repeatedly even in most secured software systems [8].
- The buffer overflows, the destructive type of security vulnerabilities which can be used to inject malicious code into a program, particularly dangerous in languages such as C and C++.
- Integer overflows, happening when the integer type used is too small to hold the selected value or result of the arithmetic operation.
- Memory disclosures, the memory corruption which can happen when the structures are not properly initialized.
- The NULL pointer dereference, which can lead to the immediate crash.
- The Race conditions, occurring when the system operations have wrong scheduling.

All of these example types of security vulnerabilities which are usually found in the real world systems, except of the race conditions, can be found using the Joern tool.

2.3 The Formal methods

The methods of static code analysis tools are often based on the exact, formal mathematical understanding of the problem. They are trying to use proofs and analytic mathematical logic methods to verify properties of a software. In academic sphere in particular, this is the most common approach to static code analysis [5], with methods such as symbolic execution and model checking [8].

While these methods have solid theoretical foundations, their integration into real world analysis is not finished yet and usage in large scale have many difficulties. As example difficulty, they need to precisely model language semantics including compiler and execution environment details [11][8].

Then it is no surprise than these methods play rather limited role for the real world security community [11].

2.4 Free tools

There are dozens or even hundreds of static code analysis tools available today [6]. Even the often underestimated compiler warnings provides sometimes complex static code analysis. As example C compilers[4] can be referred as mature static code analysis tool, given the long time of C language usage [6]. Many developers today use some version or type of the static code analysis[5]. Very popular and easy to use are the analysis tools integrated within Integrated development environment, for example the FindBugs tool for Java, which can be easily integrated into Eclipse IDE.

There exists tools which specializes in one language, like the popular FindBugs for Java, cpp check, Flawfinder for C++, Pylint for Python etc., and tools which supports more languages, like the SourceMeter or Coverity.

The security vulnerabilities and weaknesses which are commonly discovered using automatic static source code analysis includes: [18]

- Undeclared variables
- Unreachable code
- The Syntax type of errors
- The Unused functions, procedures, methods and variables
- Not initialized variables, variables used before initialization
- SQL injections

These tools perform complete analysis of the source code for such defects, providing valuable feedback and reports.

When there is possibly growing usage of free source code analysis tools as these tools are being more developed, more accessible, more known to the developers, there is also an increasing commercial use of static code analysis as a whole [23].

2.5 Commercial usage

In several industries using safety-critical systems is natural demand for static code analysis to improve safety and reliability of software systems. For example aviation software systems and nuclear software systems both use static code analysis [15][23].

Although the commercial tools can address multiple issues of the free static analysis tools, such as the high positive rate, and can be much well crafted, their limitations and drawbacks are almost the same as in the free access tools [18].

[4] E.g. GNU Compiler Collection

[5] According to [23], in 2012 near every third embedded software engineer used static analysis tools.

2.6 Current drawbacks of the source code analysis

The most static analysis tools available today are based on the Formal methods or they are using them as main source code analysis techniques [16], providing with a simple revision of the source code alike the compilers and their warnings[6].

These tools are quite useful for eliminating obvious and in many times possibly just missed code problems, like the logical inconsistencies, obviously wrong usages of certain code elements although correct semantically, the very known and exploitable SQL injection types of vulnerabilities, the bad code practices[7], the source code analysis, although providing very useful feedback for the code, does not just scan the code and return all zero-day vulnerabilities[8] of the analyzed code[9].

The static source code analysis tools currently used serves as the first wave of the software testing, as the automatic reviewing tools, revealing bad code practices, dividing by zero, illegally dereferenced pointers, logical inconsistencies etc., rooting out these bugs and vulnerabilities during the first versions of the source code. The following then can be code review by the auditors. This approach reveals deficiency in such analysis and its considerable limits. Such an analysis is not supposed to find classical CVE-type[10] of vulnerabilities[11].

There is another approach from the current source code analysis which can be ill-minded as well.

Most of up to date analysis tools provides automatic source code analysis[12], often with a lots of options before the scan begins, but they often requires no input during the analysis, because they are aimed at the fully automatic approach. But this approach to the analysis is very difficult, maybe infeasible [8] to accomplish, and may hinder the further advance of the static code analysis as a whole [11]. It may be currently[13] necessary to use deeper human-computer cooperation in the static code analysis[14], to greatly improve the analysis performance [11]. This is the approach which is embraced by the Joern tool.

[6] Especially in C/C++ developing environment they are the primary static analysis tools [6].

[7] Like wrong usage of Getters/Setters in Object-oriented programming, etc.

[8] Zero-day means the software developers had zero time to fix the vulnerability, and thus the attackers used the exploit before the developers can fix it.

[9] The Joern tool, in contrast, is developed exactly with this idea in mind.

[10] The Common Vulnerabilities and Exposures (CVE) system provides a reference-method for publicly known information-security vulnerabilities and exposures operated by the MITRE organization.

[11] The Joern source code analysis tools is designed to find exploitable zero-day security vulnerabilities.

[12] Even the source code analysis itself is defined as automatic [13].

[13] Maybe in the far future, the Artificial intelligence can handle these tasks.

[14] For example, humans and their knowledge of the codebase can efficiently guide the analysis, as shown in next chapter for Joern tool.

Chapter 3

Joern platform

The Joern tool, properly called Robust source code analysis platform Joern, is a platform for the bug hunting, and thus offers the static code analysis. Currently Joern supports C and partially C++ source code analysis, but as Joern is gaining attention, and because it is an open-source software, many people are contributing to it and are currently working on expanding its support for other languages, for instance PHP language support is being worked on. There also exists a variation of Joern tool called BJoern, which provides analysis of a binary code.

3.1 Joern methods of a source code analysis

The Joern tool, although providing static source code analysis, is not using formal methods discussed earlier. In contrast, Joern is using quite inexact methods, the pattern recognition and machine learning[1], and presents new approach - the *pattern-based vulnerability discovery*[2] [10][5][8].

Consequently Joern tries to look on the problems more from the engineering kind of perspective. It uses pattern recognition for building tools to search for bugs, and it tries to assist the code auditors in their work rather than to replace them [4][10].

3.2 Code property graphs

Imagine the simple grep function query with the parameters describing its search pattern. This query will return results based on its authors knowledge of describing what he wants to search for. Even this can be example of a simple static code analysis. If the auditor has a vital knowledge what are the patterns he searches for, even this method of analysis can be successful [4], although probably with very hight false positive rate. On this simple example is based the basic idea of the Joern queries [8].

[1] Joern as the code auditors assistance tool presented in this thesis is a part of a wider work presented in [MFW], the primary source of information about Joern. This work presents much more methods on pattern-based vulnerability discovery, including discovering vulnerabilities using Clustering, which uses supervised Machine learning as additional assistance for vulnerability discovery. In this work is Joern platform as presented in this thesis only a starting platform on which are the other method's merits presented.

[2] This is reason why Joern is named *Robust* code analysis platform, it is using robust techniques.

3.2.1 Representation of a source code

Joern source code pattern recognition is based on the idea that as a simple source code analysis can be done by using the simple grep function, specifying parameters and patterns to seek out the potentially vulnerable code, the graph representation of a code must allow to model such patterns of the code[3] [4].

From the analysis of many security vulnerabilities, core aspects needed to model these important properties of code were derived[4]. When these core aspects are identified, next task is to find an appropriate representation of a source code to be able to model these properties [10].

The problem of parsing the source code and represent it in a format suitable for further analysis has been already identified. The compilers, for instance, must have dealt with this problem before. The construction of Joern gets advantage of this [4]. Graph representations which compilers are usually doing includes abstract syntax trees, control flow graphs and program dependence graphs. Each type of this graph representations can model one different aspect of the code [8].

To better realize various aspects of these code representations, individual graph representations of a simple C function **bar** in Figure 3.1 are shown below in the Figures 3.2, 3.3, 3.4.

```
1  void bar()
2  {
3          int a =  get();
4          int max = 6;
5
6          if (a > max)
7          {
8                  call(a);
9          }
10 }
```

Figure 3.1: Simple C function for graph examples [8].

But when describing real patterns of bugs in the code, one or two of these representations may not be enough, on the one hand because none of these data structures can represent all code aspects we may need, on the other hand because transition can be demanded between syntax, control flow and data flow arbitrarily, because the sample query can describe vulnerability in patterns which require transition between individual graph representations. Therefore, formulation of another representation would be appropriate [9].

However, there may be another possibility, to use all three of these representations together. Joern introduces for this the Code property graph, a graph which is represented by *merging* the abstract syntax tree, the control flow graph and the program dependence graph into one single data structure [9].

The core idea on top of the code property graph is that the vulnerabilities *can be described* as *subgraphs* in this graph [9].

[3] Because it is needed to address these aspects of the code in the query created to find these patterns.
[4] More in e.g. [5].

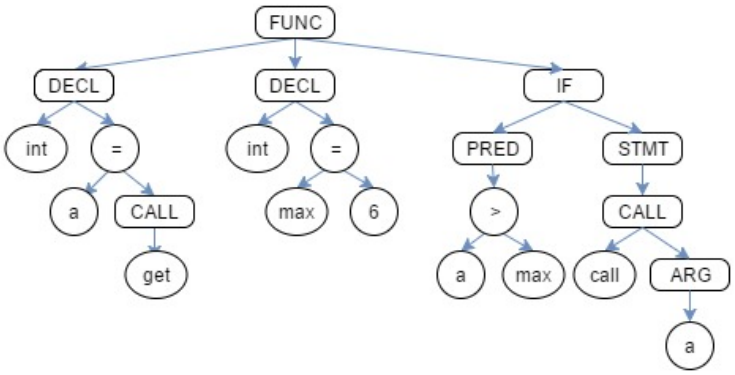


Figure 3.2: Abstract syntax tree for the function bar.

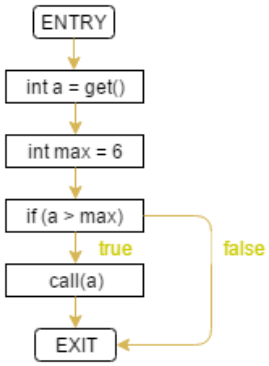


Figure 3.3: Control flow graph for the function bar.

On the Figure 3.5 is shown the code property graph for the example function **bar**(Figure 3.1).

3.2.2 The parser

Joern is using the *fuzzy parser* to parse code into *parse tree*, using *refinement parsing*, which allows to parse even incomplete code. This feature allows user to not worry if the analyzed code is not fully complete, or it is only a part, such as a patch [8].

From this *parse tree* are then created other representations, such as abstract syntax tree, control flow graph, program dependence graph and finally code property graph, all mentioned earlier [5].

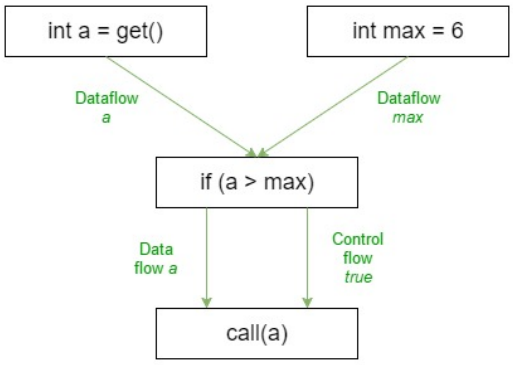


Figure 3.4: Program dependence graph for the function bar.

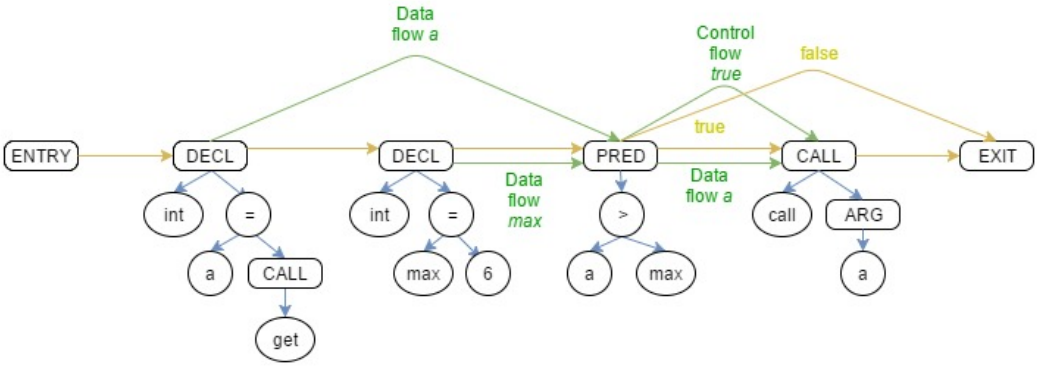


Figure 3.5: Code property graph for the function bar.

3.3 The storage of the Code property graphs

How to storage parsed code data structures and enable it's effective querying for search for the vulnerabilities are one of the most important and fundamental parts of Joern.

Joern uses modern graph databases instead of the most used relational database management systems. Why it is using this kind of database and how it influences its database interface and data retrieval is described in this section.

3.3.1 The Graph databases and Property graphs

In the beginning, the graph databases were not the first choice for the Joern. The efforts were made to map the code property graph data structure to tables and use RDBMs, or to use a non-SQL document-oriented database. But as these efforts proved difficult, the emerging graph databases were tested. This time the mapping was successful, as one of the core graph database model was matching the data structure of the code property graph [10].

It can be said that the *code property graph* is an instance of the *property graph*, which is an attributed, multi-relational graph [14], one of the two models on which are the most of a

modern graph databases based on. That means that the code property graphs can be stored in a graph database without any alteration [8].

The property graph extends the traditional mathematical concept of the graph by adding key-value pairs to the vertices and edges and labeling the edges, thus allowing diverse edge types and multiple edge connections in the graph. The number of added key-value pairs per vertex or edge is not limited. The simple example of the property graph is shown in Figure 3.6.

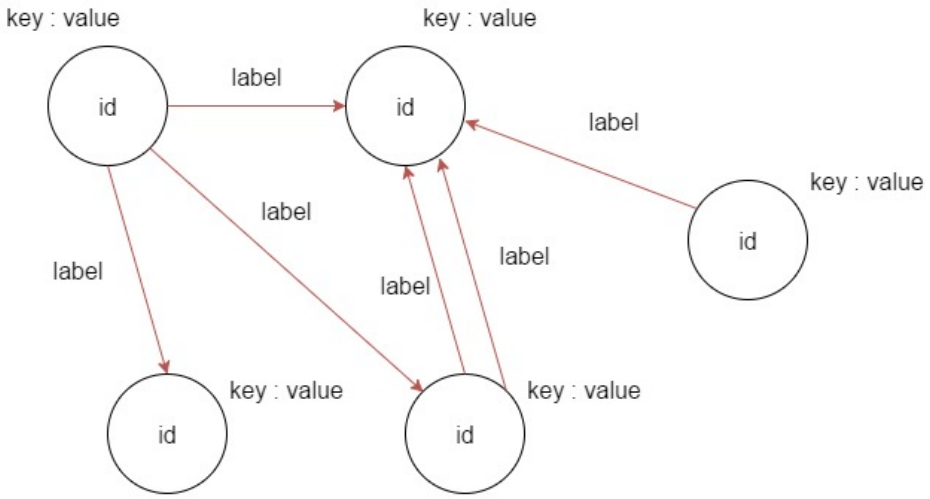


Figure 3.6: Example of a property graph.

Each node has the id, nodes can have the key:value pairs, allowing to store a data, each edge has a label, the graph is directed, can have more edges coming from the same node to another node. [19] As mentioned before, the Code property graph is subclass of the property graph, allowing to store the Joern core data structure - the code property graph in a graph database.

3.3.2 The traversals

For the data retrieval from the database, Joern defines the *traversals*, the subgraphs of the property graph (or the code property graph), which are defined as functions mapping a set of vertices and edges to another set of vertices and edges over a given property graph [8]. The travesals can be chained one atop another to create new traversals. Given this it is possible to compose complex traversals from a simple traversals. Traversals begin with a set of start nodes(or the vertices) or edges, and walks through the graph according to assigned parameters, edge labels and vertex properties. Its result is a set of final reached edges and nodes. If there are no subgraphs that fulfills query requirements, the result is empty [20].

3.4 Query language Gremlin

Gremlin is the graph database query language of choice for Joern. This means that all Joern queries for the graph databases and its whole framework is written in Gremlin-groovy, and it is possible to use Groovy in writing the gremlin queries. The concept of traversals for graph databases were pioneered by Gremlin [8]. In this section is given overview of the Gremlin query language and its usage within the Joern tool.

3.4.1 Usage within Joern

The Gremlin language is one among many introduced for emerging modern graph databases. Because the environment of the graph databases is rather new, many of them came with its own custom languages, so there are no languages that can be used universally. Gremlin language is little of an exception, because it can be employed to many popular graph databases thanks to Blueprints API, interface which is being implemented by databases based on property graph model. These databases include Neo4j[5], OrientDB and Titan database [22].

The important features distinguishing Gremlin among the other graph database query languages and making it the language of choice for the Joern includes:

- The Gremlin offers creating of so called *custom steps*, the user defined traversals, which essentially allows to build domain specific language for Joern. In other words, this feature allows Joern to build the whole *framework* of useful traversals. Furthermore, Gremlin is Groovy-based[6], allowing use of a Groovy code within it's scripts and queries, enabling even more flexibility and customization [5].

- The Gremlin is imperative language[7], allowing user to define exact way how traversal sweeps the graph. This feature allows to use extensive knowledge of graph database engine to enhance query performance to provide much better results than comparable declarative queries [8], but simultaneously flattens Gremlin learning curve, because user needs to know inner workings of Gremlin engine.

- It comes with a large set of the useful predefined traversals called the *gremlin steps*, assembled from the core traversals which Gremlin uses, providing Joern with a lot of functionality [20].

On the Figure 3.8 is shown demonstration of the very basic Gremlin traversals on property graph Figure 3.7. It demonstrates intuitiveness of the Gremlin queries and shows theirs chaining possibilities. On these examples are well seen fundamental Gremlin properties - it describes both *what to search for* and *how search for that.*

[5] Formerly, because Neo4j no more provides official support for Gremlin, although this can be circumvented by using various drivers, e.c. Bolt Neo4j Java driver.

[6] about other gremlin languages

[7] Not completely, because queries can be written in declarative manner as well [14]. But rather interesting fact is, that although Gremlin defines itself as imperative language [20], the Neo Technology, creators of Neo4j popular graph database, describes Gremlin as declarative with imperative features [12].(Neo4j database have also it's own query language, popular Cypher).

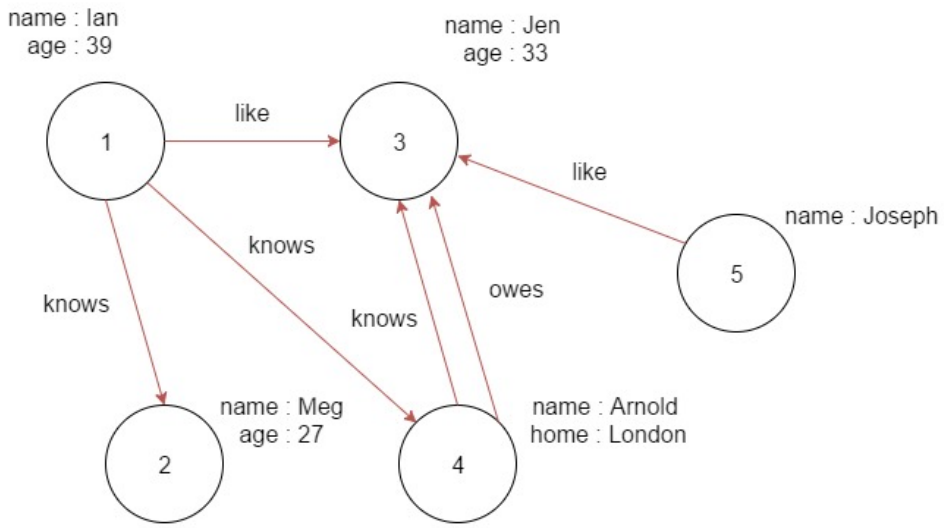


Figure 3.7: Example of the simple Property graph.

```
g.V.filter{ it.age > 27 }
v[1]
v[2]
g.V.filter{ it.age > 27 }.name
Ian
Jen
g.V.filter{ it.age > 27 }.filter{ it.age > 37 }.out.name
Meg
Jen
Arnold
```

Figure 3.8: Demonstration of the Gremlin queries.

3.4.2 Pipeline, vulnerabilities and traversals

As seen on the Figure 3.8, the Gremlin steps can be effectively chained to create the more complex traversals. Each of this whole complex traversals can be then described as a pipeline composed of the smaller pipes, or steps[8]. As mentioned earlier, the security vulnerabilities can be described as the subgraphs in the code property graph. And finally with the Gremlin query language in mind, the vulnerabilities then can be described as the *traversals* in the code property graph [10]. So in the Joern, by creating queries are created the patterns for the security vulnerabilities, and the possible subgraphs in the code property graph returned in this way are possibly the *vulnerabilities*.

[8] Or the Gremlin steps.

3.5 The Architecture

The Joern platform is composed of the many various-sized components, including drivers and plugins required for some components to work. It is an experimental tool and the version of the Joern parser up to date is 0.3.1. The parser is written in Java and should work on systems with the Java virtual machine installed [1]. It comes with the number of dependencies. The utility functionality between the large components such as parser and database is scripted via the Python 2. The Joern parser is tested to work on the Ubuntu, the Arch Linux and the Mac OS X Lion [1].

3.5.1 The components

The Joern platform composes of the Robust source code parser Joern, the Graph database, the Python-joern interface and the Joern-tools, the shell utility tools. The Python-joern and the Joern-tools are two possible ways how to mine the Code property graph for the vulnerabilities.

On the Figure 3.9 is shown the architecture scheme of the Joern platform.

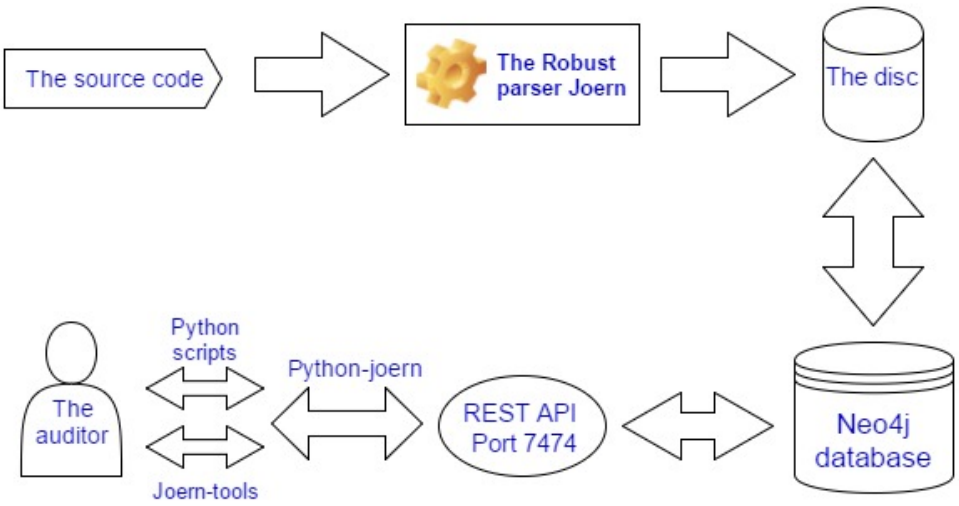


Figure 3.9: The architecture scheme of the Joern platform.

In the following, individual components are described in detail:

- The Robust code parser Joern is providing the source code parsing and saves the output to the disc, creating the folder under the name ".joernIdex/", where is stored information for a Graph database. The parser automatically parses all relevant content from the selected source folder. It is the primary component of the Joern tool and in theory it is the only part necessary. The database output of the parser can be processed by any tool or database capable of it. The parser is written in the Java 7 and requires JVM version 1.7 to run [1]. When the parser is run via the command

line, it automatically creates designated folder and automatically puts currently parsed source code to the folder. If the Joern database folder is already created, it adds the currently parsed data to the contents of the folder.

- The Graph database server which is Joern platform currently using is the Neo4j, one of the most popular and most developed graph databases available today [3], written in Java. The database server reads the folder ".joernIdex/", and provides the web interface for an access to the database via web browser or the REST API on the port 7474.

- The Python-joern interface is the utility library written in the Python 2.7, connecting to the Neo4j REST API port 7474, and exposing several functions available for querying the database via the Python applications. The Python-joern also contains the Joern framework of very useful graph traversals written in Gremlin and Groovy. The Python-joern interface is using the Gremlin-plugin, the plugin for the Neo4j server providing Gremlin query language scripting support for Neo4j database[9]. The Python-joern interface is also using the Py2neo[10], the Python library for working with the Neo4j database server in Python applications.

- The shell utilities Joern-tools are featuring several useful shell programs providing various utility functionality over Neo4j database. The Joern-tools use the Python-joern for access the Neo4j server. But theirs advantage is that they can be used directly from the command line. Even the database queries can be sent to the database via the shell utilities.

- The code auditor is conducting the analysis by writing the database queries and modifies the queries depending on the feedback received[11].

3.5.2 The Performance

The Joern tool do not need the server hardware to run properly, as it is possible to achieve the satisfactory performance even on the mid-end notebook [5]. For the optimal performance, several Joern component's settings adjustments are necessary:

- The Robust parser needs around 2GB for the Java virtual machine heap memory, to be able to sufficiently parse low, medium and large codebases. For the very large codebases, even more heap memory is recommended. If there is insufficient memory, the large codebases must be parsed in parts [1].

- The Neo4j database's minimum memory sufficient for acceptable performance of the server is 1GB of the Java heap, and another 2GB for the system RAM, and 5GB of the disc free space. For the large codebases, much more memory and disc space is needed for the acceptable querying times [2].

[9] Natively, the Neo4j provides only the Cypher query language support

[10] The Py2neo is developed and tested exclusively under Linux [17].

[11] This is the fundamental idea of the Joern tool. The code auditors using their's *knowledge* of the codebase to *guide* the source code analysis.

- The custom changes in the system settings could be needed depending on the operating system. For instance, the Neo4j database needs to increase the limit of maximum number of the open files on the various Linux-base systems to improve the performance [2].

3.5.3 The Joern platform versions and the future development

The Joern platform in its current state has been introduced for the first time at the Chaos communication congress[12] in Hamburg in 2014. Since then, many changes have followed and also the development of the entire new version of Joern has begun, with the major changes to its architecture.

The development version is currently testable and functional, but still under development. The major features it contains is a new graph database server, instead of the Neo4j server, the development version uses standalone Octopus server, the incorporated Titan database. With the new server comes also the octopus shell, which provides scripting support on the database in the real time. The Titan database also provides wide support for the Gremlin query language. The newer version also contains more updated version of the Joern parser, and experimental support of the other languages. To this date, the development version of Joern has not been released yet.

The main disadvantages of the development version are:

- The Joern framework of the Gremlin traversals in not compatible with the standalone Octopus server, because the Octopus server uses newer version of Gremlin - the version 3, which contains considerable changes in both syntax and function [22].
- Due to the change of the Gremlin version, the Python interface Python-joern is not functional, because various plugins and libraries used were considerably changed to support newer version of the Gremlin. In the development version, Python-joern is deprecated.
- Joern-tools support is limited due to the same reasons.
- The development version is prone to even major changes at any moment.

Because of these reasons, the Joern version used and described in this thesis is the current master[13] version.

As was mentioned in the previous sections, the Joern platform consists of many different components, the database server, the database plugin, the Python libraries etc. Because it is an experimental tool and is supported only with a few major contributors, the newer versions of the used software and components *does not have to* be working properly without an user modifications[14].

[12] The Chaos communication congress is an association of hackers organizing talks and events about hacktivism and data security, apart from other activities. It is the largest hacker association in Europe.

[13] Master branch on git [21].

[14] As an example, the Neo4j database server up from version 2.1 is not working properly with the Joern platform. Neo4j's up to date version is version 3.2.

Chapter 4

Experimenting with Joern and creating queries

In the first part of the experimenting, various ways of creating the queries to find the already known bugs were found. In the second part, these queries were made generalized to be usable to find bugs on any project, and were more improved.

4.1 The testing setup

The Joern tool used for the experiments was adjusted according to the recommended settings[1] described in the previous chapter.

4.1.1 The Hardware and Software

The computer used for the testing was the two years old MSI notebook, with Intel core i7, 8 GB RAM, 1 TB secondary disc, 128 GB SSD primary disc, dedicated graphic and 64-bit Windows 10 operating system. All Joern tool experiments were conducted in the 32-bit Linux Mint 17.1 Rebecca, in the Oracle VM VirtualBox on this notebook. It was given 4 GB RAM, 20 GB as primary disc space and no other restrictions from the host OS. The primary disc was dynamically allocated from the hosted OS's SSD.

4.1.2 Executing the queries

As mentioned in the Joern description, to begin the analysis, it is required to parse the chosen source code and to point the graph database to the created Joern database folder. After this, the Neo4j database server can be started and the database can be queried through Python-joern library by writing the python scripts or by using the console executing the Joern-tool's pre-written python scripts.

The used hardware, software, and the Joern tool adjustments greatly effects the analysis efficiency. If it is used the recommended hardware, software and the Joern platform is

[1] Optimizing the performance.

optimally configured, the query times can be so short[2], that they can allow the testing in a real time, which greatly simplifies query development.

4.2 Creating Joern queries

From the previous chapter, it is known that in the Joern platform, the Gremlin query[3] describes the Code property subgraph of a Code property graph in a graph database. This subgraph *is* the vulnerability searched for. By creating the Gremlin queries, the definitions for bugs are created. And in context of the Joern platform, to create a Gremlin query can mean to describe some type of a bug in the Gremlin language.

4.2.1 Structure of the query

In the Joern, the first part of the query always defines which vertexes are the start nodes of a graph walk[4]. From these start nodes, the walk through the graph continues if the conditions defined in the query are met, and finally can return the traversal result.

In the Figure 4.1 is the real-world example of the Joern query. The custom step getCallsTo, which is defined in the Python-joern library, is used for the starting node selection. The getCallsTo step requires one argument, the name of the sink function[5]. Generally, for the start node selection in the Joern are used the Apache Lucene queries [2]. The Neo4j database uses Apache Lucene node index as legacy indexing, and its queries can be written as the arguments of the custom steps used for start node selection.

In the query on the Listing 4.1, the getCallsTo step is retrieving the vertexes containing the call to calloc in the statements. The following step ithArguments returns an argument on the selected position of the call node[6].

```
getCallsTo("calloc").ithArguments("0")
.sideEffect { expression = it.code.toList()[0] }
.statements()
.in("REACHES")
.filter{ it.code.contains("int64_t") }
.filter{ it.code.contains(expression) }
.filter{ !it.code.contains("calloc") }
.dedup()
.locations()
})
```

Listing 4.1: Example of a Joern query.

After the start node selection, the steps describing the walk in the graph follows. How to create these steps is described in following sections. The average length of the queries, if each step is on the separate line, is alike the query in the Listing 4.1.

[2] Even over the large codebases.

[3] Or the Joern query, because Joern tool is using the Gremlin query language, Joern query is the same as Gremlin query.

[4] Or a traversal.

[5] The function, which is used as the start node.

[6] The counting starts from zero.

At the end of the query are to be found the steps modifying the results, to either adjusting the format or further filtering of the result set.

Every query created in the Joern tool have the basic structure described in this section.

4.2.2 Using the analysis of a code property graph

During the query creation or during the detail analysis of the query inner workings, it is very helpful to be able to display the details of the code property graph stored in the database.

Fortunately, the Joern-tools library offers such a tools. Using these shell utilities, it is possible to plot the abstract syntax trees, the control flows graphs and the control flows graphs with the data flow edges added. The functions whose graphs are desired can be easily found by using the start node selection steps.

Figure 4.1 shows the example of the generated abstract syntax tree by Joern-tools.

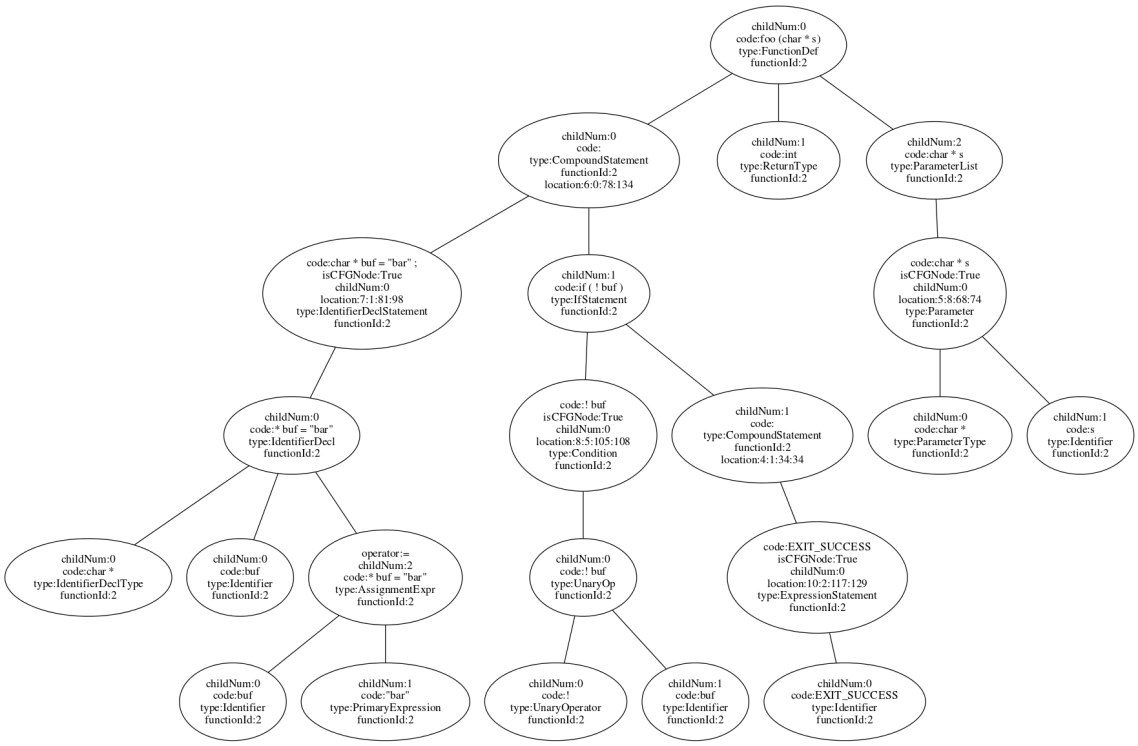


Figure 4.1: Generated abstract syntax tree example.

The queries walks the code property graph and the plotted graphs can be ultimately used to exactly track this walk or to model it. This technique is particularly useful when *debugging* the erroneous queries.

4.2.3 The custom steps

In the previous chapter it was mentioned that in the Gremlin query language, the Gremlin steps can be chained to create the more complex traversals, and also that user can define its own Gremlin steps, called the custom steps. By creating the custom steps, it is possible to craft a whole framework of the useful traversals. The Joern tool comes with many common traversals necessary for searching for bugs, creating the whole domain specific language. In this language, it is possible to describe the vulnerabilities far easier than by using only the Gremlin already defined steps[7]. It is advantageous to use this already crafted Joern traversals in the new Joern queries whenever possible.

It is easy to add own custom steps to the existing Joern custom steps framework or create a new. It is practical to reuse especially complex traversals this way.

On the Listing 4.2 is shown an example of the Gremlin custom step definition, the "locations" step, which is defined in the Python-joern steps library. The custom step on the Listing 4.2 is the upgraded version of the step defined in Python-joern. This step is used in the most Joern queries to improve the information given by the result.

```
Gremlin.defineStep('locations', [Vertex,Pipe], {
    _()
    .statements()
    .sideEffect{ code = it.code; }
    .sideEffect{ id = it.id; }
    .functions()
    .sideEffect{ name = it.name; }
    .functionToFiles()
    .sideEffect{ filename = it.filepath; }
    .transform{ "Node id: " + id + "  End source: " + code
                    + "  Function: " + name + "
                   Filepath: " + filename }
})

Gremlin.defineStep('functions', [Vertex,Pipe],{
    _().functionId.idsToNodes()
});

Gremlin.defineStep("functionToFiles", [Vertex,Pipe], {
    _().in(FILE_TO_FUNCTION_EDGE)
})
```

Listing 4.2: Definition of the custom step "locations".

4.2.4 Creating the queries for the already known security vulnerabilities

When crafting a new queries, the vulnerabilities which are the queries describing must be analyzed first. The code snippets where the vulnerabilities are to be found needs to be carefully analyzed, so the patterns of those vulnerabilities can be identified. When crafting the new queries, these patterns are then *translated* into the Gremlin query.

[7]But Gremlin itself already defines a lot of useful traversals, called the Gremlin steps.

As an example, in the VLC Media Player version 2.1.5, the following security vulnerabilities were identified:

- The attacker controlled stack allocation in RTP streaming code, CVE-2014-9630
- The attacker controlled stack allocation in SAP service discovery code, CVE-2015-1202
- The attacker controlled stack allocation in FTP access module, CVE-2015-1203

After these vulnerabilities are more closely examined, it is revealed that they have a lot in common, as they all have the same erroneous code pattern. It should be enough to try to craft the query for just one of these vulnerabilities.

In the Figure 4.2 is shown the code snippet for the vulnerability in the RTP streaming code.

```
1  int rtp_packetize_xiph_config( sout_stream_id_t *id,
2  const char *fmtp, int64_t i_pts )
3  {
4      if (fmtp == NULL)
5          return VLC_EGENERIC;
6
7      /* extract base64 configuration from fmtp */
8      char *start = strstr(fmtp, "configuration=");
9      assert(start != NULL);
10     start += sizeof("configuration=") - 1;
11     char *end = strchr(start, ';');
12     assert(end != NULL);
13     size_t len = end - start;
14     char b64[len + 1];
15     memcpy(b64, start, len);
16     b64[len] = '\0';
17     ...
18 }
```

Figure 4.2: The attacker controlled stack allocation vulnerability in VLC MP 2.1.5.

In the function rtp_packetize_xiph_config at line 13, the *len* variable represents the length of the string in the user file, and thus can be attacker-controlled. The problem is that at the line 14 the same variable directly controls the size of the buffer, which is allocated at the stack. If the *len* exceeds the size of the stack, the buffer pointer will point outside of the stack, and subsequently in the *memcpy* call on next line the memory will be corrupted and program crashes.

After the understanding what caused the vulnerability, the query can be constructed. Firstly, the sink function is to be chosen. In this case, it can be the *memcpy*:

```
getArguments('memcpy', '2')
```

Afterwards, it is important to save the argument, because the same argument as in *memcpy* call is used in the buffer allocation. Furthermore, because we can be sweeping large Code property graphs, it is suitable to filter arguments to only those resembling the names

which are often used when naming the length of string, the all strings containing "len" as substring should be enough[8].

```
getArguments('memcpy', '2')
.filter{ it.code.contains("len") }
```

As the next step, the query will walk the graph to try to reach the declaration of the buffer, containing the argument we found in *memcpy* call.

```
.statements()
.in("REACHES")
.filter{ it.code.contains("char") }
.match { it.type == "IdentifierDeclStatement" }
.filter{ it.code.contains(argument) }
```

From all these declarations, we filter those which can have any sign of pointers and which for certain contains brackets, because this strongly indicates the array declaration.

```
.filter{ !it.code.contains("*") }
.filter{ it.code.contains("[") }
```

The walk now contains many important patterns the vulnerability of this type have in the code. Before the testing, removing the duplicates and adding additional information to the output steps can be added.

```
getArguments('memcpy', '2')
.sideEffect{ argument = it.code; }
.filter{ it.code.contains("len") }
.statements().in("REACHES")
.filter{ it.code.contains("char") }
.match { it.type == "
                    IdentifierDeclStatement"
                     }
.filter{ it.code.contains(argument) }
.filter{ !it.code.contains("*") }
.filter{ it.code.contains("[") }
.dedup()
.locations()
```

Listing 4.3: The query able to find all three attacker controlled stack allocation vulnerabilities in the VLC MP 2.1.5.

The Listing 4.3 shows the complete query. The query is able to find all the three attacker controlled stack allocation vulnerabilities in the VLC MP code, and yields *only one* false positive.

The process of the query creation to search for the already known security vulnerabilities is always the same. The optimal choosing of the range of desired search can be adjusted during the testing.

4.3 Generalizing the Joern queries

During the vulnerability hunting, it is very effective and tempting to use any specific knowledge about the codebase to augment the query. It could be the special uncommon

[8] This filter step can be of course skipped, resulting in the more general search, but the other bug patterns shall remain.

functions from used framework which are hard to handle, the already known exotic type of vulnerabilities or the specified functions known to be dangerous etc.

But if any of these patterns are explicitly used in the query, the query will become less reusable, or not reusable at all. It would be needless to use this query for the vulnerability search in any other codebases.

However, as the queries are describing the vulnerabilities, the method how to generalize the queries which are using the code specific patterns is to reasonably *rewrite* the graph walk of these queries. Consequently, the key task is to *reformulate* the vulnerability description.

This also implies that the extent of how much the query can be generalized depends largely on the properties of the vulnerability, which is the query describing. For example, think of this problem: The specifically named dangerous function often have the user-controlled second argument unsanitized before using it. To describe this kind of vulnerability in the query, it is necessary to specify the name of this function as the sink. Without it, it is possible to create workarounds such as scan all dangerous functions, or to try to create regular expression to find the names which such function could have, but these have serious difficulties.

But regardless of the possible difficulties, in general, the most types of the vulnerabilities can be described by carefully choosing only those patterns in the code which can be ordinarily found in any codebase.

On the Listing 4.4 [4] is shown the query presented in the CCC talk by Dr. Fabian Yamaguchi, created to find the heap-based buffer overflow in VLC media player's automatic updater[9]. This query is not generalized, because it uses as the sink the stream_Size function used specifically in the VLC media player.

```
getCallsTo("stream_Size").statements()
.filter{ it.code.contains("int64_t") }
.out("REACHES")
.filter{ it.code.contains("malloc") }
.code()
```

Listing 4.4: The non-general query used to find the heap-based buffer overflow in VLC media player.

To generalize this query, it is necessary to analyze the vulnerability it describes. On the Figure 4.3 is the code snippet of the vulnerable code.

On the line 17, in the constant *i_read* is stored the size of the stream *p_stream*. In the next line, the buffer *psz_update_data* is allocated to store the data. In the malloc call, the *i_read* + 1 is used as an argument, but in the 32-bit environment, after the addition the result is truncated accordingly to fit the 32-bit size_t. Because the *i_read* is user-controlled, the truncation can occur at user's will.

After the vulnerability analysis, the query can be rewritten, excluding the code-specific patterns. On the Listing 4.5 is shown the generalized query.

[9] CVE-2014-9625.

```
1  static bool GetUpdateFile( update_t *p_update )
2  {
3      stream_t *p_stream = NULL;
4      char *psz_version_line = NULL;
5      char *psz_update_data = NULL;
6
7      p_stream = stream_UrlNew( p_update->p_libvlc,
8                                UPDATE_VLC_STATUS_URL );
9      if( !p_stream )
10     {
11         msg_Err( p_update->p_libvlc,
12                  "Failed to open %s for reading",
13                  UPDATE_VLC_STATUS_URL );
14         goto error;
15     }
16     const int64_t i_read = stream_Size( p_stream );
17     psz_update_data = malloc( i_read + 1 );
18     /* terminating '\0' */
19     if( !psz_update_data )
20         goto error;
21     if( stream_Read( p_stream, psz_update_data,
22                      i_read ) != i_read )
23     ...
24 }
```

Figure 4.3: The heap-based buffer overflow in VLC media player's automatic updater.

```
getCallsTo("malloc")
.ithArguments("0")
.sideEffect { expression = it.code.toList()[0] }
.statements()
.in("REACHES")
.filter{ it.code.contains("int64_t") }
.filter{ it.code.contains(expression) }
.dedup()
.locations()
```

Listing 4.5: Generalized query from Listing 4.4.

The specific sink was simply replaced by standard *malloc* call, and from the *malloc* statement, the graph is simply walked to find the necessary vulnerability pattern, the statement containing the declaration of int64_t type named equally as the *malloc* argument.

The original query Listing 4.4 found the seven results, from which one was positive. The generalized query Listing 4.5 found six results, including the positive one. However, it is important that the generalized query yielded *exactly identical* results, excluding one false positive. This demonstrates the ability to form different traversals yielding nearly the same results.

This example demonstrates how can be a non-general query generalized, providing that the vulnerability it describes can be expressed differently in a graph walk. If the vulnerabili-

ties are expressed in a newly crafted queries with the regards to the ordinary code patterns[10], the newly created queries will be also *already generalized*.

After these findings, it was easy to ensure that almost all queries which were experimented with and which were created during work on this thesis were already defined as *generalized* from the start, and so usable for searching for bugs at any codebase. Furthermore, all the queries used as examples in this thesis are generalized, except the Fabian Yamaguchi's query at Listing 4.4.

4.4 The taint-style queries

The taint-style queries are the queries expressing the taint-style vulnerabilities, mentioned in the chapter 1. This type of vulnerabilities can be described as the insufficient sanitizing of a user-controlled data, which is passed from the input to a sink function. Even today, this kind of vulnerabilities is a persistent security problem [8].

The Joern is well suited for the taint-style vulnerability search. The existing framework of useful traversals contains the *unsanitized* step, a very complex traversal whose definition easily exceeds 100 lines of code. This traversal is principal in the revelation of the taint-style vulnerabilities.

4.4.1 Properties of the taint-style traversals

The Listing 4.6 shows a typical taint-style query.

```
import java.util.regex.Pattern;
getArguments('memcpy', '2')
.filter{ !it.argToCall().toList()[0].code.matches('.*(sizeof|min).*') }
.sideEffect{ argument = it.code; }
.sideEffect{ sId = it.statements().toList()[0].id; }
.unsanitized(
{ it, s -> it._().or(
        _().isCheck('.*' + Pattern.quote(argument) + '.*'),
        _().codeContains('.*lloc.*' + Pattern.quote(argument) + '.*'),
        _().codeContains('.*min.*'))})
.filter{ it.id != sId }
.filter{ !it.code.contains("lloc") }
.dedup()
.locations()
```

Listing 4.6: The typical taint-style query.

The taint-style query works likewise the *grep* search. It sweeps over all occurrences of a chosen sink and retrieves its selected argument, then tests the sanitization[11] of selected arguments for all these occurrences.

[10] If it is possible.

[11] If it is possible to reach from a user-defined input to a sensitive sink without any type of check or control.

During the experiments, the taint-style queries demonstrated a lot of potential and had usually the highest scores of bug founding. This can be certainly contributed to their enormously broad search. Because of this, they were tested a lot. Most interesting properties found were:

- Versatility, the great variety of their checks and filters gives a lot of space for easy possible adjusting to the source code. If they returns too big result set, it is possible to adjust their settings and filters accordingly during the analysis by analyzing the false positives and immediately rerun the improved query, already filtering these false positives

- As they sweeps over all occurrences of a chosen sink, they have an enormously broad search, resulting in the biggest size of a result set of all other queries, and potentially the greatest number of the false positives

- Due to their great complexity and a broad stroke, the taint-style queries also have usually the longest query times[12]

4.4.2 The more complex taint-style traversals

It is possible to create even more complex traversals, by combining more *sanitized* steps in one query. Various complex graph walks can be created this way, elegantly and precisely expressing vulnerabilities to reduce the number of the false positives.

The Listing 4.7 shows the query designed to find the null-pointer dereference type of vulnerabilities, using as a sink the *memcpy* function. This query was extensively tested and is using complex adjustments to find either *every possible instance* of a bug, either to reduce number of the false positives *as much as possible*. The more complex vulnerability description expressed in a query can lead to a better results, but also to the increased complexity and long query times.

[12]The longest query time experienced during the experiments was 17 minutes. It was the taint-style query in Listing 4.7.

```
import java.util.regex.Pattern;
getCallsTo("memcpy") /* first unsanitized step */
.ithArguments("2")
.filter{ !it.argToCall().toList()[0].code.matches('.*(sizeof|min).*') }
.sideEffect{ argument = it.code; }
.sideEffect{ sId = it.statements().toList()[0].id; }
.unsanitized(
{ it, s -> it._().or(
        _().isCheck('.*' + Pattern.quote(argument) + '.*'),
        _().codeContains('.*lloc.*' + Pattern.quote(argument) + '.*'),
        _().codeContains('.*min.*'))})
.filter{ it.id != sId }
.filter{ it.code.contains("alloc") }

.statements()    /* graph walk to return back */
.out("REACHES")
.dedup()
.match { it.type == "CallExpression" && it.code.startsWith("memcpy") }

.getCallsTo("memcpy")  /* second unsanitized step */
.ithArguments("0")
.sideEffect{ arg = it.code; }
.sideEffect{ sIdd = it.statements().toList()[0].id; }
.unsanitized(
{ it, s -> it._().or(
        _().isCheck('.*' + Pattern.quote(arg) + '.*'),
        _().codeContains('.*!.*' + Pattern.quote(arg)),
        _().codeContains(Pattern.quote(arg)))})
.filter{ it.id != sIdd }
.filter{ it.code.contains("alloc") }
.filter{ !it.code.contains("xmalloc") && !it.code.contains("xrealloc") }
.dedup()
.locations()
```

Listing 4.7: The more complex taint-style traversal.

In the first part of the query, the third argument of the sink function *memcpy* is filtered for size of or min words, indicating any sign of a check. Then, the argument is stored as the variable, and the statement id is also stored. Next, the *unsanitized* step checks the control flow from the sink to the source[13], yielding the source. The source is then checked if it is not the starting sink, or it contains the word *alloc*, which would mean the allocation is occurring. On this position, the first part of the search finishes.

The next part is a short graph walk - from the yielded source, the graph is walked to the statements, then try to reach for the outgoing edges, remove the duplicates, and try to match the same sink function[14].

From here, the *unsanitized* step is again used, this time to check the control of the first argument. In the end, the resulted allocating statements are filtered for undesirable *xmalloc*

[13] Accordingly the defined checks.

[14] The check is not made strictly for the same id, because the other checks in the query are already strict enough, so more general approach during return walk can uncover more bugs.

and *xrealloc*, which both checks for the null pointer.

On the Figure 4.4 is shown code snippet of the one of the results of this query, the one of the uncovered forgotten checks in *malloc* allocation in the VLC media player in real_sdpplin.c, which can possibly result in a crash. In the code can be directly seen the vulnerable walk pattern, which the query searched for.

```
1  ...
2  if ( desc->mlti_data_size ) {
3       desc->mlti_data = malloc(desc->mlti_data_size);
4       memcpy(desc->mlti_data, decoded, desc->mlti_data_size);
5       handled=1;
6       *data=nl(*data);
7       lprintf("mlti_data_size: %i\n", desc->mlti_data_size);
8  }
9  ...
```

Figure 4.4: The null pointer dereference in VLC MP 2.1.5.

4.5 Testing of the queries

At the beginning, the testing of the queries can be easily done by copying the vulnerable snippets of the code and thus creating a small database to test the queries on. After initial testing, the queries should be tested on a small or a medium codebases, due to the inability of the effective debugging. The erroneous queries can run for a very long time over a sizeable codebase, so it is not possible to debug such queries effectively.

After the testing of the queries over a small codebases, the working queries can be then tested on a medium and a large-sized codebases for the further improvements, such as reducing the number of the false positives.

4.6 Evaluation of the query results

Although the ratio of one positive sample over ten false positives is viewed as good ratio CCC [5], during the experiments, it has shown to be effective to pursuit even better results, because it can shorten often lengthy process of the result classifying. The many queries created during work on the thesis were improved to the point of a very few false positives. For example, when executed over the VLC media player 2.1.5, the query in Listing 4.6 yields four results, from these the three are positives, and the query in the Listing 4.7 yields ten results, from these are nine the positives.

4.7 Problems and difficulties

The biggest problem over the course of the experimenting was actually the installation of the Joern tool itself and learning of its use.

4.7.1 Documentation

As described in chapter three in the section 3.5.3, it is currently around three years since Joern's first presentation. Since then, the major part of the tool did not changed much, though the work on the next major release took place. The problem was that the small but essential changes in the Joern query framework and the other Joern platform components has not been documented or documentation on their part has been outdated, resulting in the misleading information. The impact of this was the very much delayed understanding of the platform and process of creation of the queries. The official documentation itself, although very helpful, is outdated and contains many critical errors.

4.7.2 Installation

The next major challenge was the installation process, because many essential parts of the Joern platform were critically dependent on the exact versions of various plugins, libraries and components. Because these tools evolved in their versions and Joern platform did not incorporated these changes, sometimes the very outdated versions were required. But even with the correct versions, several severe errors were still encountered during the installation. These errors required often only minor changes in the configuration of the installation tools due to the outdated settings, but still delayed the installation process.

To add, although Joern's Robust parser was tested on Windows, Mac and Linux [1], the other components, for example the Python-joern, requires the Python development libraries supported only for the Linux, in fact allowing the installation of the Joern platform only on the systems with Linux[15], preferably the Ubuntu-based[16], and so not guaranteeing to work on any other systems or Linux versions.

[15] Although, the Joern's Robust parser can be used separately.
[16] The Joern platform with all its components was tested on the Ubuntu Linux.

Chapter 5

The code testing tool

For the coherent testing of the created queries, the simple tool was designed, the *Code-tester*. It allows the simple and effective testing of the code, using the all created queries.

5.1 Architecture and design

As described in the Joern analysis, for the communication with a graph database, the Joern platform uses the Python-joern interface written in the Python 2.7. This interface offers several methods for the interaction with the Neo4j database, and because of this, the language chosen for the tool was also the Python. The chosen Python version was also the same as used for the Python-joern interface, the 2.7, for the compatibility reasons. Moreover, the Python is also very popular and widely used language.

5.1.1 Application design

The tool is a simple console application based on the object-oriented Model–View–Controller architecture. The MVC pattern was chosen to allow both easy future development and anytime simple modification. The application is not designed as the product used by any user, but rather as the open tool used by a code auditor to help to organize the set of the already created queries and to test them for bug mining over various codebases.

5.1.2 The functionality

The tool serves as the last line in the architecture scheme of the Joern platform (Figure 3.9), where it occupies the position of the Python script, but is providing with the much more helpful functionality.

The primary function of the code testing tool is to do the source code analysis - it allows user to choose from the different code tests, and executes these tests over the source code. "Under the hood", it is executing the set of the already defined generalized queries over the parsed source code which is located in the graph database. It secures that the chosen code tests are executed properly and the analysis will finish even if individual queries fails or do not finish in the given time limit.

5.1.3 The user interface

The application offers both execution with the arguments and execution without the arguments, which starts the console menu with the several choices.

5.1.4 Database communication

For the communication with the database, the testing tool uses the Python-joern library, which is directly using the REST API on the port 7474, which is exposed by the Neo4j database. The database interface uses these methods to create a connection and use this connection for executing the Joern queries over the Neo4j database.

5.2 Implementation

The complete source code of the application is stored on the github[git] and also on the enclosed disc.

5.2.1 The data objects

In the application, the Joern queries are represented by the query object in the Model package. On the Listing 5.1 is shown the *Query* class definition.

```
class Query(object):
    """Defines query object type"""

    def __init__(self, number, name, code, desc, usage):
        self.number = number
        self.name = name
        self.code = code
        self.desc = desc
        self.usage = usage

    def __str__(self):
        return ("Number: {}\n"
            "Name: {}\n"
            "Code: {}\n"
            "Description: {}\n"
            "Usage: {}").format(self.number, self.name, self.code,
            self.desc, self.usage)
```

Listing 5.1: The Query.py class defining the query object.

The all generalized queries are divided into three categories by the type of the vulnerabilities they describe. Their individual instances are statically created in their respective model repositories, allowing direct access and modification.

5.2.2 Database connection

The *DBInterface* class in the Model package manages the database connection, using the JoernSteps class methods to get connection, connecting to the database and running the Gremlin queries. The *JoernSteps* class is imported from the Python-joern library.

5.2.3 Control of the query execution

To achieve the control over the query execution over the Neo4j database, the individual queries are executed consecutively in the separate processes using the Python *multiprocessing* package. The more lightweight threads cannot be used instead of the processes, because the queries are executed outside of the Python interpreter, so the control over the application control flow after the query starts the execution cannot be handled from the Python. Using the processes, the query execution can be stopped after certain time, and also its eventual crash will not crash the application.

The *runAllTests* method in the Listing 5.2 shows the consecutively executed processes in the tool.

```
def runAllTests(self):
    if not self.__dbInterface.connectToDB():
        return False

    print "[+] Fetching query list."
    queryList = self.__queryRepository.getQueryList()

    for q in queryList:
        # Define subprocess
        p = multiprocessing.Process(
            target = self.__runQueryInProcess, args=(q, ))

        print "\n[+] Running query {}, {}."
            .format(q.number, q.name)
        p.start()

        # Waiting on process to finish set ammount of time
        p.join(self.__query_time_limit)
        # Terminate process if it didn't finished in time
        if p.is_alive():
            self.__testView.delSymbol()
            print "[+] Query didn't finished in time limit, skipping."
            p.terminate()
    return True
```

Listing 5.2: The runAllTests method managing creation of the Processes.

The individual queries of the chosen test are executed consecutively, writing its results on the console and starting another, until the all queries of the chosen test are not executed.

5.2.4 Argument parsing

For the argument parsing and the command-line options, the application is using the Python *argparse* module, providing it with standard console application behavior.

5.3 Usage

The tool can be started by running the *CodeTester.py* file, using the command:

```
python CodeTester.py
```

with or without arguments. In both mods, the same options are available and the result of the chosen tests has the same format.

5.3.1 Running Code-tester with the arguments

Using the command-line options and arguments, the desired task will be immediately carried out and after the task, the application exists.

The command-line options and their arguments are:

- `-o {1,2,3,4,5,6}, --option {1,2,3,4,5,6}`

  which controls which of the respective options will be carried out, the option are: 1) Run all Buffer Overflow tests 2) Run all Memory Disclosure tests 3) Run all Null Pointer Dereference tests 4) Run all tests 5) Settings 6) Help

- `-l LIMIT, --limit LIMIT`

  which sets the maximum query time in minutes allowed "code"

- `-h, --help`

  which shows help message with usage description

The command-line options are defined as optional, but if is not specified the *-o* parameter, the application will print the interactive console menu (5.3). If the *option* argument is specified, but without the *limit* argument, as the maximum query time limit will be used default value in the *CodeTester.py* file.

5.3.2 Running Code-tester without the arguments

Running the application without the arguments (Listing 5.3) prints the interactive console menu with the numbered options, which can be chosen by typing desired number and pressing enter.

```
  ////////////////////////////
 ///     Code-tester 1.0     ///
////////////////////////////

Options:
  1) Run all Buffer Overflow tests
  2) Run all Memory Disclosure tests
  3) Run all Null Pointer Dereference tests
  4) Run all tests
  5) Settings
  6) Help
  7) Quit
```

Listing 5.3: The interactive console menu of the code testing tool, written into the console.

The chosen option will be executed and after the task the application will wait for the next commands. As the maximum query time limit is used the default value stored in the *CodeTester.py* file.

5.3.3 Example of the use

As the example, the Code-tester can be used to analyze the source code of VLC media player 2.1.5. After the usage of the Joern platform to parse the source code and starting the database server, Code-tester can be started from its folder, simply by using the command line options and arguments.

The command

```
python CodeTester.py -o 1 -l 20
```

starts the buffer-overflow tests with time limit of 20 minutes over the VLC MP, which will yield over time the complete results:

```
==========================================
[+] Running Buffer-Overflow tests.
[+] Creating connection.
[+] Connecting to the database.
[+] Fetching query list.

[+] Running query 1, Buffer-overflow
[+] Quering finished.
[+] Elapsed time: 42.2251839638 seconds.
[+] Number of positive samples: 4
[+] Possible vulnerabilities:
Node id: 630547  End source: memcpy ( ( ( char * ) & id ->
clutID ) + 2 , p_vide + 70 , i_vide - 70 )  Function: OpenVideo
Filepath: /home/ondra/joern-0.3.1/vlc-2.1.4/modules/codec/
quicktime.c
Node id: 872623  End source: memcpy ( p_box -> data . p_name ->
psz_text , p_peek , p_box -> i_size - 8 )  Function:
MP4_ReadBox_name  Filepath: /home/ondra/joern-0.3.1/vlc-2.1.4/
modules/demux/mp4/libmp4.c
Node id: 460248  End source: memcpy ( sink -> name , i -> name ,
```

```
namelen + 1 )  Function: sink_add_cb  Filepath: /home/ondra/
joern-0.3.1/vlc-2.1.4/modules/audio_output/pulse.c
Node id: 2151103  End source: memcpy ( psz_buf , psz_command ,
psz_temp - psz_command )  Function: ExecuteCommand  Filepath:
/home/ondra/joern-0.3.1/vlc-2.1.4/src/input/vlmshell.c

[+] Running query 2, buffer calloc
[+] Quering finished.
[+] Elapsed time: 6.6542570591 seconds.
[+] Number of positive samples: 1
[+] Possible vulnerabilities:
Node id: 337126  End source: memcpy ( * pp_sectors ,
p_vcddev -> p_sectors , ( i_tracks + 1 ) *
sizeof ( * * pp_sectors ) )  Function: ioctl_GetTracksMap
Filepath: /home/ondra/joern-0.3.1/vlc-2.1.4/modules/access/
vcd/cdrom.c

[+] Running query 3, modified buffer
[+] Quering finished.
[+] Elapsed time: 8.65275502205 seconds.
[+] Number of positive samples: 13
[+] Possible vulnerabilities:
Node id: 1562312  End source: memcpy ( psz_uri_scheme ,
psz_subtitles , i_scheme_len )  Function: Item :: buildInputSlaveOption
Filepath: /home/ondra/joern-0.3.1/vlc-2.1.4/modules/
services_discovery/upnp.cpp
Node id: 1396529  End source: memcpy ( p_data , psz_data , i_data )
Function: vlclua_todata  Filepath: /home/ondra/joern-0.3.1/
vlc-2.1.4/modules/lua/libs/httpd.c
Node id: 535696  End source: memcpy ( p_enc -> fmt_out . p_extra ,
p_block -> p_buffer , len )  Function: Encode  Filepath:
/home/ondra/joern-0.3.1/vlc-2.1.4/modules/codec/dirac.c
Node id: 599057  End source: memcpy ( p_sys -> name ,
name_ptr , name_len )  Function: OpenDecoder  Filepath:
/home/ondra/joern-0.3.1/vlc-2.1.4/modules/codec/omxil/
android_mediacodec.c
...
```

The complete result spans the 10 query results for the buffer-overflow kind of test.

5.4 Possible future work

There are numerous ways in which the code testing tool can be improved, though it is already providing the basic functionality for use as an analysis tool.

It could be useful to incorporate other effective tools and components of the Joern platform into the Code-tester, including the traversal framework and Joern-tools command line library, which would create universal Python tool for managing queries and code property graphs, and all functionality would be in the one place.

Chapter 6

The results of analyses and conclusion

After the completion of Code-tester, the generalized queries created so far were inserted into the tool and project was tested on the several open-source projects. During the testing, the queries were further improved and advantages and limitations of the tool were analyzed.

6.1 Results

The results presented are the results found to date.

6.1.1 VLC media player 2.1.5

The well known and popular media player. VLC MP was already in the center of attention in the original Joern presentation at the CCC 2014. It is used in Joern tutorial as example project.

Except for the all bugs that were already found by Fabian Yamaguchi[1], in the VLC media player 2.1.5 were found two additional unknown minor issues - the null pointer dereferences. Snippet of the one of them, found in matroska_segment_parse.cpp, is shown in the Figure 6.1. The other one was already shown in chapter 4, in the Figure 4.4.

```
1  ...
2  if( tk->i_extra_data > 0 )
3  {
4      tk->p_extra_data = (uint8_t*)malloc( tk->i_extra_data );
5      memcpy( tk->p_extra_data, cpriv.GetBuffer(),
6          tk->i_extra_data );
7  }
8  ...
```

Figure 6.1: The possible null pointer dereference found in VLC MP 2.1.5.

Additionally, the high number of the bad practices in the code were also found.

[1] Excluding the bugs found using the Machine learning

6.1.2 Nemea

Nemea is the open-source system for the network analysis of traffic and anomaly detection developed under CESNET. During the analysis, a few minor null pointer deferences were found. In the Figure 6.2 is one of them, found in the blacklist_downloader.c.

```
1  ...
2  if (start == end) {
3      // Only filename was given
4      return_path = malloc(strlen("./") + 1);
5      memcpy(return_path, "./", 3); // copy with terminating byte
6      } else {
7      // Copy path
8      return_path = malloc(end - start + 2);
9      memcpy(return_path, start, end - start + 1);
10     return_path[end - start + 1] = 0;
11 }
12 ...
```

Figure 6.2: The possible null pointer dereference found in Nemea.

These rather minor bugs were acknowledged by the Nemea developers. Additionally, the number of the bad practices in the code were also found and reported.

6.1.3 Kodi

Kodi is an open-source media player. It was chosen for the analysis as it is also the open-source media player like the VLC, although written primarily in C++. It was possible that it will contain similar vulnerabilities alike found in the VLC MP. But, no major vulnerability in the Kodi was found yet by Code-tester, only the minor bugs were found to date. In the file asn1.c, the null pointer dereference was found. It is shown in the Figure 6.3.

```
1  ...
2      if (cert[(*offset)++] != ASN1_BIT_STRING)
3          goto end_sig;
4
5      x509_ctx->sig_len = get_asn1_length(cert, offset)-1;
6      (*offset)++;          /* ignore bit string padding bits */
7      x509_ctx->signature = (uint8_t *)malloc(x509_ctx->sig_len);
8      memcpy(x509_ctx->signature, &cert[*offset],
9              x509_ctx->sig_len);
10     *offset += x509_ctx->sig_len;
11     ret = X509_OK;
12 ...
```

Figure 6.3: The possible null pointer dereference found in Kodi media player.

6.1.4 Apache HTTP Server

The Apache HTTP Server is a powerful HTTP/1.1 web server. In this open-source project, minor issues were found as well, like the bad code practice, or few null pointer dereferences, as the errors found in mpm_winnt.c, in the Figure 6.4.

```
1  ...
2  args = malloc((ap_server_conf->process->argc + 1) *
3                sizeof (char*));
4  memcpy(args + 1, ap_server_conf->process->argv + 1,
5         (ap_server_conf->process->argc - 1) * sizeof (char*));
6  args[0] = malloc(strlen(cmd) + 1);
7  strcpy(args[0], cmd);
8  args[ap_server_conf->process->argc] = NULL;
9  ...
```

Figure 6.4: The possible null pointer dereference found in Apache HTTP Server.

6.1.5 System Security Services Daemon and Network Security Services

Unlike the other open-source projects, both System Security Services Daemon and Network Security Services are oriented on the security, therefore should contain much less vulnerable code and no bad practice at all. This theory proved to be right, because as was seen from the analysis conducted so far, only the potentially dangerous code constructs were yet found, not even minor issues found in the previous projects.

6.1.6 Outcome of the analyses

But the source code analysis of some of these projects is not completed yet. The tests using the queries with the most broad search - the mentioned taint-style queries, can find many possible vulnerabilities. These kind of queries found in various projects the large number of possible bugs. In the SSSD were found more than 50 potentially vulnerable code issues, in the NSS even more than 170 possible vulnerable code issues. In some projects were found even greater numbers, taking into consideration the size of the individual projects.

Due to these numbers, only part of these findings were examined to date, others were left for further investigation later. The preferably examined results were the ones presented by the queries which have the less broad search. As all queries used in the tool are describing the real bug patterns found in the open-source projects before, it is possible that the unchecked query results may yet contain vulnerabilities. The reason why this kind of queries are not cut out of analysis entirely or modified is that their number of returned results varies greatly between different projects.

In the examined results of those queries checked so far, there are often very dangerously looking parts of the code, yet no obvious vulnerability was found between them yet. The advancement is slow because the investigation of the vulnerable code findings often requires deep analysis of the code.

6.2 Conclusion

The work started by describing the static source code analysis, showing its methods, current use and also its drawbacks. It was revealed that some of this drawbacks, the underestimation of the possibilities of human-computer cooperation in vulnerability discovery and too exact approach of the formal methods, are resolved by the Joern tool. The Joern tool was analyzed, it was described how it works, what is its core data structure - the code property graph and how is this data structure stored in the graph database. The query language Gremlin was introduced and it was described how the Gremlin queries are used to mine the code property graph for bugs.

After the analysis, it was described how the various Joern queries, which are able to find already known bugs, can be created. Then it was demonstrated how these queries can be generalized to be used on other projects. The description of design, implementation and usage of the tool using these generalized queries was presented. This tool was eventually tested for the source code analysis on the several open-source projects. The tool presents three types of tests, the buffer overflow test using ten queries, the memory disclosure test using six queries and the null pointer dereference test using two queries.

As the conclusion from the analyses and the results discovered so far, it was demonstrated that the tool is definitely working, and by quick running of the chosen test or tests, the most vulnerable parts of the code, containing patterns defined as vulnerable in the queries, are found. Many minor previously unknown bugs were discovered so far, several of them in nearly every project tested, so the queries seem to be generalized and the principles described in the analysis seem to work properly. The reason why queries which found the severe security bugs in the VLC media player or Linux kernel did not immediately found similar bugs in the other projects, is probably that simply there are *none the same* bugs. The patterns in the code are certainly recognized correctly by the description of bugs in the queries, so this is the only explanation.

From these findings, it seems that the way how to augment the performance of the code analysis tool is to simply add more traversals defining the various bugs and vulnerabilities. There is simply no general prescription how the bug looks like, so more patterns and descriptions of the vulnerable code are needed. As the creation of the new queries and incorporating them in the tool is very simple, it is also quite easy to refine the analysis of the tool. This easy way of refinement of the analysis is certainly also the great advantage of the tool.

The limit of the provided tool is simply the limited number of the described bugs and vulnerabilities by the individual queries. The same bugs simply do not have to appear in the similar "format" in the various projects submitted to the analysis.

As the results seem to be very promising, the future work on the tool and adding new bug definitions could lead to an unparalleled source code analysis.

Bibliography

[1] Joern documentation.
`http://joern.readthedocs.io/en/latest/index.html`, from 20. 5. 2017.

[2] The Neo4j Manual v2.1.5.
`http://neo4j.com/docs/2.1.5/index.html`, from 20. 5. 2017.

[3] *TOP 31 GRAPH DATABASES* [online]. 2014. [cit. 20. 4. 2017]. Available from: <http://www.predictiveanalyticstoday.com/top-graph-databases/>.

[4] CCC. *Mining for Bugs with Graph Database Queries* [online]. 2014. [cit. 5. 10. 2016]. Available from: <https://user.informatik.uni-goettingen.de/~fyamagu/pdfs/2014-ccc.pdf>.

[5] CCC. *Mining for Bugs with Graph Database Queries* [online]. 2014. [cit. 5. 10. 2016]. Available from: <https://www.youtube.com/watch?v=291hpUE5-3g>.

[6] CppCon. *CppCon 2015: Jason Turner The Current State of (free) Static Analysis* [online]. 2015. [cit. 7. 3. 2017]. Available from: <https://www.youtube.com/watch?v=sn1Vg8A_MPU>.

[7] CVE-2014-0160. *The Heartbleed Bug* [online]. 2014. [cit. 5. 4. 2017]. Available from: <http://heartbleed.com>.

[8] Fabian Yamaguchi. *Pattern-Based Vulnerability Discovery* [online]. 2015. [cit. 1. 2. 2017]. Available from: <https://ediss.uni-goettingen.de/bitstream/handle/11858/00-1735-0000-0023-9682-0/mainFastWeb.pdf>.

[9] Fabian Yamaguchi, Nico Golde. *Hunting Vulnerabilities with Graph Databases* [online]. 2014. [cit. 7. 12. 2016]. Available from: <http://mlsec.org/joern/docs/2014-inbot.pdf>.

[10] hackativity. *Fabian Yamaguchi – Mining for Bugs with Graph Database Queries* [online]. 2015. [cit. 5. 3. 2017]. Available from: <https://www.youtube.com/watch?v=lGjc3kl1zXo>.

[11] HEELAN, S. Vulnerability Detection Systems: Think Cyborg, Not Robot. *IEEE Security & Privacy*. 2011, 9, s. 74–77.

[12] Joy Chao. *Imperative vs. Declarative Query Languages: What's the Difference?* [online]. 2016. [cit. 17. 4. 2017]. Available from: <https://neo4j.com/blog/imperative-vs-declarative-query-languages/>.

[13] Margaret Rouse. *source code analysis* [online]. 2010. [cit. 17. 4. 2017]. Available from: <http://searchsoftwarequality.techtarget.com/definition/source-code-analysis>.

[14] Marko A. Rodriguez. *Property Graph Algorithms* [online]. 2011. [cit. 7. 3. 2017]. Available from: <https://markorodriguez.com/2011/02/08/property-graph-algorithms/>.

[15] NATHANIEL AYEWAH, J. D. M. W. P. D. H. J. P. Using Static Analysis to Find Bugs. *IEEE Software*. 2008, 25, s. 22–29.

[16] OWASP contributors. *Static Code Analysis* [online]. 2017. [cit. 10. 3. 2017]. Available from: <https://www.owasp.org/index.php/Static_Code_Analysis>.

[17] SMALL, N. The Py2neo v3 Handbook. `http://py2neo.org/v3/`, from 20. 5. 2017.

[18] Steven Lavenhar. *Code Analysis* [online]. 2008. [cit. 17. 4. 2017]. Available from: <https://www.us-cert.gov/bsi/articles/best-practices/code-analysis/code-analysis>.

[19] Thomas Frisendal. *What is a Graph anyway?* [online]. 2016. [cit. 7. 3. 2017]. Available from: <http://graphdatamodeling.com/Graph%20Data%20Modeling/GraphDataModeling/page/PropertyGraphs.html>.

[20] web:gremlin. Gremlin wiki. `https://github.com/tinkerpop/gremlin/wiki`, from 20. 5. 2017.

[21] web:joern. Octopus-platform joern. `https://github.com/octopus-platform/joern`, from 20. 5. 2017.

[22] web:tinkerpop. Apache TinkerPop. `http://tinkerpop.apache.org/`, from 20. 5. 2017.

[23] Wikipedia contributors. *Static program analysis* [online]. 2017. [cit. 10. 3. 2017]. Available from: <https://en.wikipedia.org/wiki/Static_program_analysis>.

Appendix A

Nomenclature

API	Application Programming Interface
CCC	Communication Chaos Congress
DSL	Domain Specific Language
IDE	Integrated Development Environment
MP	Media Player
NSS	Network Security Services
SSSD	System Security Services Daemon
VM	Virtual Machine

Appendix B

Content of the included CD

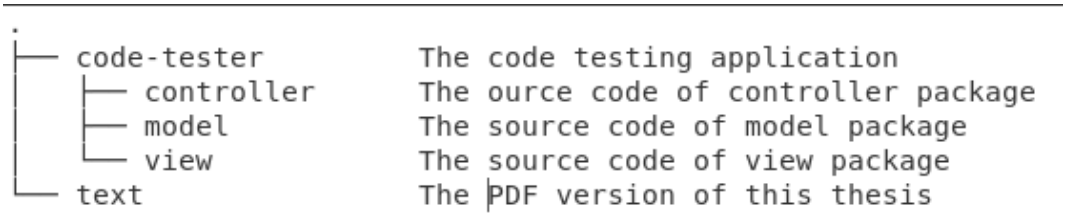


Figure B.1: Included CD