LAPPEENRANTA UNIVERSITY OF TECHNOLOGY
FACULTY OF TECHNOLOGY MANAGEMENT
DEGREE PROGRAMME IN INFORMATION TECHNOLOGY

SOFTWARE SECURITY VULNERABILITIES IN MOBILE PEER-TO-PEER ENVIRONMENT

Master's thesis

Supervisor: D.Sc. Pekka Jäppinen

Examiners: Professor Esa Kerttula
D.Sc. Pekka Jäppinen

Lappeenranta, February 20, 2012

Igor Botyan
Liesharjunkatu 6 D 35
53850 Lappeenranta
Tel. +358 46 599 0559
igor.botyan@lut.fi

ABSTRACT

Lappeenranta University of Technology

Faculty of Technology Management

Degree Programme in Information Technology

Igor Botyan

Software Security Vulnerabilities in Mobile Peer-to-peer Environment

Master's thesis

February 20, 2012

100 pages, 6 images, 3 tables and 6 appendices

Examiners: Professor Esa Kerttula

D.Sc. Pekka Jäppinen

Keywords: mobile environment, peer-to-peer, PeerHood, software security, vulnerabilities

Increase of computational power and emergence of new computer technologies led to popularity of local communications between personal trusted devices. By-turn, it led to emergence of security problems related to user data utilized in such communications. One of the main aspects of the data security assurance is security of software operating on mobile devices.

The aim of this work was to analyze security threats to PeerHood, software intended for performing personal communications between mobile devices regardless of underlying network technologies. To reach this goal, risk-based software security testing was performed. The results of the testing showed that the project has several security vulnerabilities. So PeerHood cannot be considered as a secure software. The analysis made in the work is the first step towards the further implementation of PeerHood security mechanisms, as well as taking into account security in the development process of this project.

Dedicated to the memory of Arkadiy Botyan, my grandfather (1936-2008).

PREFACE

Doing good be thankful for this.
Leo Tolstoy

This project is done as a part of PeerHood project in Communications Software laboratory of Lappeenranta University of Technology. I would like to thank for this great opportunity to be involved in this project.

First and foremost I would like to thank my supervisor, D.Sc. Pekka Jäppinen, for suggested contemporary and interesting research topic, for his guidance and advice. Without his assistance this work would not have been possible. I am grateful to my second supervisor, Esa Kerttula.

I would like to express my sincere gratitude to my parents and brother. They have supported me during the course of this dissertation.

Finally, many thanks to Anna Liss and all those helping hands who directly or indirectly supported, encouraged, and helped me to accomplish this thesis.

Lappeenranta, February 20, 2012

Igor Botyan

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF USED ABBREVIATIONS

API	Application Program Interface
DoS	Denial of Service
GPRS	General Packet Radio Service
OOP	Object-Oriented Programming
OS	Operating System
P2P	Peer-to-Peer
PDA	Personal Digital Assistant
PTD	Personal Trusted Device
QoS	Quality of Service
WLAN	Wireless Local Area Network
XSS	Cross-Site Scripting

1 INTRODUCTION

There is nothing in this world constant but inconstancy.
Jonathan Swift

This section provides the motivation of the research carried out in the work and is comprised of several subsections. The background to the research, as well as the research question, The main objectives, the scope, and the delimitations of the research are considered in the first subsections. The last subsection contains a structure of the thesis.

1.1 Background

Increase of power of computation devices including mobile devices for the last 10-15 years led to emergence of new types of mobile devices. Previously each mobile device was related to one or more categories in terms of it's functionality. For example, it could provide communication functionality, store and process user data, or perform any other type of operations. Whereas it is hard to refer a contemporary mobile device to any category.

The main reason of this fact is that a contemporary mobile device provide the same set of functionalities that different categories of mobile devices could provide in the past. For instance, contemporary smartphone can be used by user to communicate or to store it's personal data. Whereas each user had to use several devices, mobile phone and Personal Digital Assistant (PDA), to perform both of these operations. This fact is the basis of a contemporary trend towards the universalization of contemporary mobile devices.

At the same new communication technologies are widely used. For example, support of Bluetooth [1] technology by mobile devices increased popularity of local interactions between users. Such kind of interactions are charecterized by ease, spontaneity, and short time of communication setup. These factors led to popularity of peer-to-peer network paradigm [2]. Peer-to-peer approach favours shared utilization of mobile environment resources and does not depend on underlying communication technology [3] [4].

The main consequence of the universalization of contemporary mobile devices and technologies is the universalization of software utilized on these mobile devices.

Clear examples are utilization of the same software platform on different mobile devices (Apple iOS [5], Google Android [6]) as well as orientation of software platform manufacturers to support one unified user interface on different devices (Microsoft Metro [7], Ubuntu Unity [8]). Mobile device universalization on the one hand leads to decrease of a number of computer devices used by a user and on the other hand it leads to the fact that a mobile device becomes to be as a user personal representative in virtual environment.

Mobile device utilization as a such representative leads to the emergence of Personal Trusted Device (PTD) idea [9]. The PTD concept is based on the fact that user data is actively utilized within communication process performed by the user. At the same, communication does not depend on it's environment and used technologies. Insufficient personal user data security assurance could lead to various problems [10]:

- unathorized access to personal user data
- malicious software injection
- denial of service
- remote control of a device

In the conditions of the communication platform universalization, personalization is one of the most promising directions in service providing area. So it can be said that the personal user data security problem can be considered as one of the most critical problems in the area. One of the factors affecting data security is security of a software operating on a mobile device and utilizing the data. This it can be said that two factors, broad use of mobile peer-to-peer networks and importance of security assurance of software operating on mobile devices, are the main preconditions for the research conducted in this work.

1.2 Research question and main objectives

Personal user data security depends on a number of factors, both human and technical [11]. Technical factors are the following: security of data transfer technologies supported by user mobile device; security of software that performes communication operations and has access to private user data; and a number of others.

The object of the research is related to security vulnerabilities of software utilized by a user on his PTD to perform communication operations. The subject of the research

is a network environment, based on the concept of peer-to-peer communication between PTD's in a mobile environment [12]. So the question of the research is the following: does PeerHood contain security vulnerabilities or not.

It is necessary to solve the following tasks to answer the question of the research:

1. to study the software security problem
2. to consider existing methods of software security analysis
3. to examine PeerHood project
4. to perform PeerHood analysis to find software security vulnerabilities related to PTD

Software security area is quite extensive. There is a number of aspects for examination of the area. This fact considerably influences on the results of the research conducted in the work. For this reason, it is necessary to define a scope of the research. The scope is described below.

1.3 Scopes and delimitations

Security assurance is performed on each stage of software development lifecycle [13]. Appropriate activities are conducted to provide it. The current development stage of PeerHood project is implementation of it's functionality characteristics, and it is almost completed. So a practical part of the work mainly related to software security testing area.

PeerHood project analysis consists only in search of known security vulnerabilities. No additional activities related to the found vulnerabilities is not conducted in the investigation.

The practical part of the work is performed in GNU/Linux [15] environment. It influences on a number of applied tools used to perform software security analysis. Only those tools are utilized during the analysis that are available on this platform.

PeerHood is implemented in C++ programming languange [16] and interacts with Operating System (OS) through Qt framework [17]. Therefore, those vulnerabilities that related to these technologies are focused during studying of the software security concept.

1.4 Structure

The first two sections comprise a theoretical part of the work. Section 2 deals with software security problem and contains basic information about the object domain. Software security analysis methods needed to conduct a practical part of the work are considered in Section 3. Next two sections are related to the practical part of the work. PeerHood network environment examined in this work is briefly described in Section 4. Concept, goals, key requirements and architecture of the project are considered in this section. Section 5 deals with PeerHood security analysis. Last section contains general conclusions of the work, a list of unsolved tasks and recommendations for further work.

2 SOFTWARE SECURITY PROBLEM

There is more to life than simply increasing its speed.
Mohandas Karamchand Gandhi

The section contains general information about software security problem. The basics are considered, including a definition, approaches to assurance and affecting factors. Software security defects, including vulnerabilities, are described in detail. Classifications of known software security vulnerabilities are given. Software security problems specific to mobile and peer-to-peer environments are considered.

2.1 Software security concept

Nowadays the general role of information is solely increased in the process of continual complication of science and technology [18]. According to the Cambridge Dictionary of Philosophy [19], information is "an objective entity and can be generated, carried, stored, and others". Modern business environment is increasingly interconnected, thus information transmission operation forms the basis of information systems [20]. Therefore information security assurance problem in such systems becomes more and more urgent.

The term information security means "information and information system protection from unauthorized access, use, disclosure, alteration, or destruction" [21]. It aims to protection of the following information properties:

Integrity Guarding against improper information modification or destruction, and includes ensuring information nonrepudiation and authenticity.

Confidentiality Preserving authorized restrictions on access and disclosure, including means for protecting personal privacy and proprietary information.

Availability Ensuring timely and reliable access to and use of information.

Protection of these properties is known as *CIA Triad*, that is represented on Fig. 1. Besides the properties the following additional properties are also noted: trustworthiness, accountability, non-repudiation, and reliability [20].

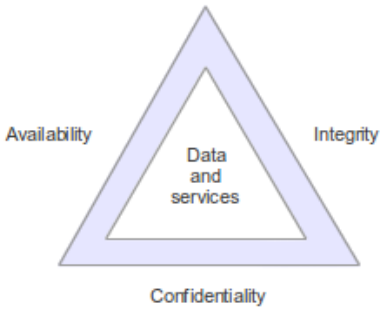

Figure 1. Information security goals [25]

If one CIA Triad element is not taken into account, this fact may cause the following complications [22]. A loss of confidentiality could lead to an unauthorized disclosure of information. A loss of availability could lead to a disruption of access to or use of information or an information system. A loss of integrity could lead to an unauthorized modification or destruction of information.

To avoid the problems listed above, various approaches to information security assurance are applied. Domarev marks out the following methods: legislative, administrative, procedural, and program-technical [23]. The methods are complementary, so they may be used both in the aggregate and separately. In this work only program-technical approach to information security assurance is considered. It is closely related to the notion of computer security.

According to the RFC 4949 [24], computer security is "the protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)". By contrast with information security, computer security has also two additional goals [25]:

Authenticity The property of being genuine and being able to be verified and trusted; confidence in the validity of a transmission, a message, or message originator [24].

Accountability The security goal generates the requirement for actions of an entity to be traced uniquely to that entity [24].

Software security forms the basis of information security. McGraw considers that

software is principal and the most critical aspect of computer security [26]. The idea of software security consists is to provide proper software execution under a malicious person attack [27]. A number of factors that affect this capability is considered in the next subsection.

2.2 Software security factors

There are three main factors that sufficiently influence to software security. These include complexity, extensebility, and connectivity [28]. In the area of software security the factors are known as *Trinity of Trouble* [26]. They are the following: complexity, extensibility, and connectivity.

When developing a software, complexity brings to a big number of problems including security problems. It is explained by the fact that the complexity has the character of non-randomness and non-linearity, depending on the developing software size [29]. McConnell thinks that the complexity is the main programming imperative [30]. With the developing software size increasing, A chance to make a mistake is increased considerably. Including an error that influences on security of the software. This dependence is represented on Fig. 2. The figure represents the fact that number of errors is considerably increased as the appropriate software size grows. The more bigger a project, the more functionality it has. Therefore the more complex it becomes, thus working on this project one has more chances to make an error.

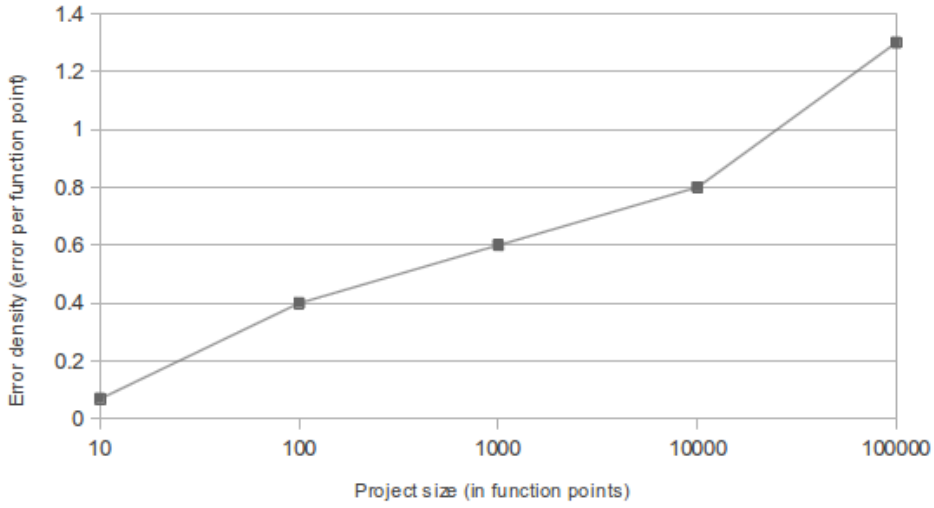

Figure 2. Dependance of errors on software size [31]

Usually the size of a modern software project grows owing to increase of it's additional components (or extensions). It effects positively on the fact that the software is deployed as rapidly as possible in order to gain market share [28]. But, in turn, there is a risk of vulnerability occurrence in one of the software plug-ins. Thereby the software security assurance process becomes more sophisticated.

Connectivity also influences on software security considerably. A vulnerability occurrence in a such software compromises security of all installed and connected to a network copies of the software [26] [28]. It promotes to a growth of attacks and ease of their carrying out in automatic manner [20]. Thus attacks related to capability of attacked software to connectivity may have a character of massiveness.

There are also secondary factors that influences on software security. These include used during development principles and practices, tools, acquired third-party components, execution environment, and many others [11]. Ignoring any security factors may cause serious consequences, so software can be exposed.

2.3 Software security threats

According to the RFC 4949 [24], threat is a potential for violation of security, which exists when there is a circumstance, capability, action, or event that could breach security and cause harm. That is, a threat is a possible danger that might exploit a vulnerability

Security threat may become apparent on any stage of software development lifecycle. This fact is used for classification of software security threats based on their origin. According to the classification, threats are divided by the following categories [11]:

Unintentional It can be ignoring secure programming practices, failure to take into account security on deployment stage of development lifecycle, invalid requirement to a project from the security point of view.

Intentional but not malicious It can be absence of input data check.

Malicious It can be intentional developing of a backdoor in an application, use of a default password, possibility of a penetration attack.

Software can not be considered as a safe one if it has a security vulnerability. Even

if security threat is found, the fact leads to possibility of an attack carrying out by a malicious person.

2.4 Attacks on software

Attack on software is an an intentional act by which an entity attempts to evade security services and violate the security policy of a system [24]. A person conducting such kind of attacks is called malicious person, or threat agent [25]. Purposes of the attack carrying out can be the following: gathering or destruction of information system resources (assets) or information itself and denial of service [32]. An attack is targeted to an asset. An asset is is a system resource that is required to be protected by an information systems' security policy or required by a system's mission [24]. At this moment there are various types of the attacks, and also classifications.

Attacks are classified by their nature and have the following types: interception, modification, falsification, and interference [33]. Interception is an unathorized party gaining access to an asset. Modification is an unauthorized party gaining control of an asset and tampering with it. Falsification is an unauthorized party inserts counterfeited objects into the system. Interference is occured when an asset is destroyed or made unusable.

Attacks are also classified by their principle of operation and have the following types [13]. Reconnaissance-attacks help a malicious person to gather more important information about an attacked system and it's environment. Enabling-attacks are referred to actions enabled a malicious person to carry out another types of attacks. Disclosure-attacks are aimed at obtain confidential data. Subversion-attacks are related to change of attacked system workflow. Sabotage-attacks have one of the following goals: denial of service or denial of access for normal users.

Attacks are also classified by development lifecycle stages: when an appropriate error led to possibility of an attack performance has been made [34]. If it has been made on design stage then the following types of attacks may be performed: man-in-the-middle, race conditions, replay-attacks, sniffer-attacks, and others [25]. If an error has been made during code writing, then attacks related to buffer overflow or input data check, as well as use of a backdoor, may be performed [35]. Finally, an attack may be performed owing to unsafe software deployment. The most common

type of such attacks is DoS-attack (Denial of Service) [25].

Attacks on software are performed owing to exploits. Exploit is a method which enables a malicious person to successfully perform an attack [36]. It can be a special formed shellcode, an application or just a set of instructions [32], [37], [38]. Exploits are always aimed at use by a malicious person of some security defect, which attacked software contains.

2.5 Software security vulnerabilities

Developers often make errors during software development. The errors may appear on any software development lifecycle stage: in requirements, at design time, during writing the code, testing, or deployment. In common words, an error is an human's action that leads to incorrect results (for example, software that contains some defects) [39]. The consequence of any software error existence is the fact that defects appear in this software. The defects are incorrect instruction, process, or data definition in a computer program [39]. A defect in it's turn can bring to a fault. Fault is an impossibility of a software or it's individual component to perform needed operations according to the specification [39].

Error related to security may become to apparent because of a number of reasons. It can be both lack of developers' motivation or knowledge and lack of utilized software security assurance technolody [13]. Graff marks out the following three factors influenced on lack of software security assurance by a developer: technical, psychological, and ordinary [34]. The technological factors are related to complexity of modern software in the first place. The psychological factors are related to incorrect security risk estimation and difference of developer's and attacker's attitude to a software. Finally, ordinary factors are the following: a source of third-party source code, existence of strict term of software development lifecycle, lack of security requirements.

But not each error made by a developer affects security of software developing by him. Vulnerability is a such error in software specification, development, or configuration, that can lead to security policy violation and further exploiting of the software [24] [28] [38] [40]. According to the Dowd's research results, the majority of vulnerabilities is related to improper input data handling, a trust level between the system's components, Application Program Interface (API), interaction with

software environment and other applications [41].

The majority of known at this moment vulnerabilities is discovered in software written in middle-level languages [42] [43] [44]. Middle-level languages are characterized by the fact that they contain both elements of high-level languages (portability, abstraction from low-level computer processor operations) and elements of low-level languages (direct memory access, support of direct processor instructions execution). On one hand, it can be explained by the fact that the languages provide less number of restrictions by design compared to more high-level languages [45]. For instance, middle-level languages are characterized by manual memory management. On other hand, such languages as C [46] and C++ [16] are the most popular programming languages at this moment [47].

But even utilizing more high-level languages, one can not be sured that a developed software is secure because runtime environments of high-level languages are written themself in a middle-level language (PHP [48], Python [49], Java [50], and others). So if a runtime environment has a vulnerability, then an appropriate high security risk becomes to apparent in all software systems based on this runtime environment.

Most often vulnerabilities specific to C/C++ languages are consequences of errors related to inadequate understanding by a developer of that how data is allocated in memory. According to the CERT [51], there is a set of most dangerous and frequent vulnerabilities. Information about them is utilized in the practical part of this work, so their description is given below.

2.5.1 Integer errors

Almost all known programming languages are exposed to integer errors. The problem consists in the fact that integer data type has a limited number of it's possible values, so it has it's maximum and minimum values. The reason of the fact is architectural because each number is represented in memory as fixed block of bits [52].

There are several types of errors related to integer data types: integer overflow, sign error, and cast error [45] [52] [53] [54] [55]. Overflow of an integer variable appears when it is assigned to a value that is greater than the maximum value of the integer type of the variable. A similar error can appear if an integer variable is assigned to a value stored in another integer variable, but the value range of the second variable is wider than the appropriate value range of the first variable. Also, if an integer type is unsigned, then it's minimum value is equal to 0, but its' maximum value

is doubled. In that connection, the overflow can appear during computations that involve operands of both signed and unsigned integer types.

In general case the may lead only to denial of service or logical errors. It means that they can'be exploited themself but they can lead to appearance of other types of vulnerabilities, such as buffer overflow (see Section 2.5.2) [56].

Appendix 6.1 contains an example of a program that checks whether an entered number is ordinary or not. This program contains a number of vulnerabilities that can be utilized by a malicious person. One of them is related to the integer error. A number entered by a user is stored in `number` variable that has `unsigned char` type, and it's maximum value equals to `UCHAR_MAX` (255). If the user enters a number greater than the maximum, then the `number` variable is overflowed. In this program it affects only to it's correctness. But in other cases this type of errors may lead to change of software normal state, denial of service, or confidential information leakage.

To find an integer error the following methods can be used: to verify source code, to utilize static analysis tools, to utilize facilities provided by compilers [53] [56].

2.5.2 Buffer overflow

Buffer overflow appears when a computer program writes data beyond the bounds of an allocated memory block. Buffer overflow is considered as most dangerous and frequent vulnerability related to the middle level languages [56]. In such languages built-in facilities aimed at array bound checking are not provided, thus operations related to arrays are possible unsafe.

Most often the attacker's purpose of buffer overflow is to execute special code [28]. For this he firstly prepares special input data that leads to buffer overflow, then gains control of the system, and executes needed operations.

In code of the program represented in Appendix 6.1 threre is an error that may lead to buffer overflow. If a user enters a number greater than 9999 (it means that the number has 5 or more digits), then it leads to overflow of the `number_buf_size` buffer. Presence of such error gives a malicious person to gain control over the program and execute a consequence of operations. It is most dangerous in the case when a program-victim has privileged access rights. Software gain control tecnhiques

related to buffer overflow are explained in detail by Erickson and others [36] [57].

To discover this type of vulnerability, the following methods can be used: source code inspection, utilization of compiler facilities, performing stress testing, utilization of automatic code analysis tools [53] [54] [56]. Stress testing is a kind of software testing that is utilized to evaluate reliability and fault tolerance of a system under conditions of normal operating limits exceeding.

2.5.3 Format string vulnerabilities

Cause of format string vulnerabilities appearance is input data use without any checking. While formatted output functions, for instance `sprintf()` of C programming language, that write data to an array of characters are used, it is supposed that the buffer has intenionally big size [55]. It may lead to it's overflow.

In C/C++ languages these vulnerabilities can be used by a malicious person to write some data to arbitrary blocks of memory or to carry out information leakage [56]. An application can be exploited to bring it into crash, to get stack contents, to get memory contents, or to rewrite the value of a memory block [45] [57] [58].

In code of the program represented in Appendix 6.1 there is a vulnerability relatedt to a format string. `printf()` format string function is used to write output data. The first argument of this function is omitted, but it defines a format of output string. It permits an attacker to define it's own format, and an appropriate input data in this case can change the normal state of the program and even execute any consequence of operations [36].

To detect such kind of vulnerabilities the following methods are applied [56] [57]:

- inspection of all blocks of code that use format output functions
- utilization of automatic static analysis tools (RATS [59], Flawfinder [60], PScan [61], and others)
- utilization of additional libraries that increase safety (FormatGuard [62], libformat [63], libsafe [64], and others)

2.5.4 Pointer vulnerabilities

Pointer is a variable that stores a function address, an array's element or another type of data structure [45]. It can be quite difficult to use pointers because they are

error prone by their nature. The most frequent error is double free of a memory block pointed by a pointer [55]. Also there are other types of pointer vulnerabilities, such as pointers to data or functions.

Pointer subterfuge is used as a general term for exploits that change value of a pointer [45]. Pointers to data also can be changed to execute arbitrary code. If a pointer to data is used as a target for the following assignment, then an attacker may control it's address to change values of other blocks of memory [45].

In code of the program represented in Appendix 6.1, an input data entered by a user are stored in a array. Access to them is performed via `number_buf_size` pointer. If it has an appropriate value, then an attacker may perform arbitrary operations. Exploiting is directly performed using this pointer to data.

The main measure to prevent this kind of vulnerabilities is thorough revision of all blocks of code that use pointers. Additional measure is to use special compiler facilities or it's extensions that are aimed at search of this kind of errors. Mudflap is an example of such kind of technical tool [65].

2.5.5 Incorrect error and exception handling

The error can lead to crash or denial of service of software, as well as transition to an unsafe state and other appearance of other types of vulnerabilities [56] [66]. In general, it may appear in the following cases [30]:

- an application prints too much information in an error message
- error ignoring
- incorrect error handling
- handling of not all possible exceptions
- exception handling on an inappropriate abstraction level

In the listing below a fragment of code written in C++ language represents API for providing information about range of goods in an internet-store. The program contains an error related to wrong choice of an abstraction level of the exception that the code may raise. In a signature of `get_tshirts()` method, information about how the method may raise a `sql_exception` exception is given. This exception is used to sginal about database errors. From the security point of view such kind of

signature is incorrect. At first, a malicious person may know details about the inner program mechanism, but it can not be allowed. At second, using the error message of the exception he may retrieve important information that makes him easier to intrude to the system.

Listing 1. Example of an API element that declares an exception with incorrect abstraction level

```
#include <list>
#include "sql_exception.h"

class fan_store {
public:
  std::list<tshirt> get_tshirts() throw sql_exception;
  // ...
}
```

Such kind of vulnerabilities depend strongly on a context, so they can be found and eliminated only by thorough code inspection. To eleminate them the following techniques can be used [56]:

- proper handling of all possible exceptions with appropriate abstraction levels
- checking of returned values if they can signal about errors
- utilization of logging facilities instead of brief information about an axception inclusion into a message about error

2.5.6 Information leakage

Information leakage can help a malicious person to perform an attack to the system or data [56]. System configuration and other type of important information can be included to messages about errors delivered to a user [54]. Confidential data can be stored unencrypted both in source code and in an application's resources that are readable for a user [56].

In the listing below a fragment of code of an internet-store is represented. The `get_tshirts()` method is not safe because it contains a vulnerabilitity that can lead to information leakage. If the operation related to retrieving of details about goods from a database fails, then the method raises an exception with `fan_store_exception` type. But beside text message about the error, the exception stores information about causes of the operation failure. It is not safe because a

malicious person has access to information about system configuration and details about inner details of the system.

Listing 2. Example of code that contains important information leakage

```
#include <list>
#include "sql_exception.h"

class fan_store_exception : sql_exception {
  // ...
}

class fan_store_impl : fan_store {

public:
  std::list<tshirt> get_tshirts()
    throw fan_store_exception;

private:
  std::list<tshirt> get_tshirts_from_db()
    throw sql_exception;
}

std::list<tshirt> fan_store_impl::get_tshirts()
  throw fan_store_exception {

  return get_tshirts_from_db();
}
```

The following measures can be used to prevent such kind of vulnerabilities:

- utilization of logging facilities for storing detail information about errors; messages about errors do not contain this information
- storing confidential information in encrypted form
- utilization of safe channels to perform interprocess communication
- data transmission in encrypted form if the channels are not established

2.5.7 Input data checking

From an attacker point of view, his interaction with attacked software is occuring via input data handling. For every application there are several input data sources.

Such kind of sources can be standard input, file system, software environment, user interface, and others [66] [69].

The majority of attacks is carried out using specially formed input data for an attacked application [45]. As a result of an error related to the input data handling, the application brings to an unsafe state and can perform operations needed for a malicious person. In addition, similar activities performed by a malicious person could lead to "penetration" that is characterized by logical access gain to important critical system data [24].

Code of the program represented in Appendix 6.1 contains buffer overflow and format string vulnerabilities. Checking of input data entered by user can decrease probability of these vulnerabilities exploitation and even prevent them.

The main measure to prevent such kind of vulnerability is input data checking on all application levels, realizing "security in depth" principle [70]. The principle of input data checking on all application levels is also known as secure programming technique [30]. Input data safety assurance consists in the following activities [52]:

1. identification of all application input data sources on all levels
2. selection of appropriate strategy to recognize incorrect data on each level of the application
3. attempt to decrease possibility of incorrect data input through API or other sources

However secure programming technique has it'w own number of advantages and disadvantages [30]. The main disadvantages are the following: increase of software complexity and decrease of it's execution efficiency after putting into operation of input data checking blocks. And the code can contain errors at that. The obvious advantage consists in radip error dection and elimination.

2.5.8 Race conditions

Some operations require atomicity, continuity of their execution [71]. During their execution they can not and should not be interrupted by other threads. Errors happen when condition of atomicity in an application is violated [41]. It happens in the case of parallel execution of such kind of operations. Race condition is a design error of multitasking software, when success of operation execution depends on the

order it's blocks of code are executed [72]. In this case condition of atomicity is violated.

Race condition does not depend on programming language. More often it happens in a moment when two different executing threads or processes have possibility to alter some specified shared resource, and because of this fact they intervene themselves [56] [66]. A shared memory block, a database, a file, or event a variable can be used as such kind of shared resources.

Exploitation of applications that contain race condition vulnerability occurs by the means of shared resource alteration [25]. And the attack is carried out in specified time window at that. In this moment these resources can be altered by developer's error. Consequences of such attacks are the following: destruction of a shared resource, crash of attacked application, data fabrication and it's further handling by a victim at the level of it's privilegies.

In the listing below a fragment of code written in C language uses some shared resource. This fragment contains a race condition error. Originally, after each checking of the resource availability (`is_resource_available()` method) it is possible to handle the resource data (`process_resource_data()` method). From the security point of view, it is not correct because there is a gap between execution of these methods. At this moment thread manager may interrupt execution of this block of code. And after it's resuming the following situation may occur: resource data is not yet ready to be processed but this operation will be executed.

Listing 3. Example of race condition

```
void consume() {
  for(;;) {
    if(is_resource_available()) {
      process_resource_data();
      break;
    }

    wait_for_resource_availability();
  }
}
```

To provide correct execution of an application in a multitasking environment, it is necessary to fulfill one of the following conditions depending on the context [41] [72]:

- a state of a process should not depend on a state of a shared resource

- each shared resource should be read-only
- access to each shared resource should be synchronized

Additional measure to find and prevent race conditions related to file system is utilization of static and dynamic tools [45]. The most well-known such static analysis tools are the following: ITS4 [73], PVS Studio [74], and RATS [59]. And the most well-known such dynamic analysis tools are Helgrind [75], and Inspector XE [76].

2.6 Classification of software vulnerabilities

According to the CERT data for the period from 1995 to QIII-2008 [51], a number of discovered software security vulnerabilities constantly increases (see Fig. 3). The fact demonstates that a risk of new attacks performance rises permamently. This leads to growth of software security assurance in general.

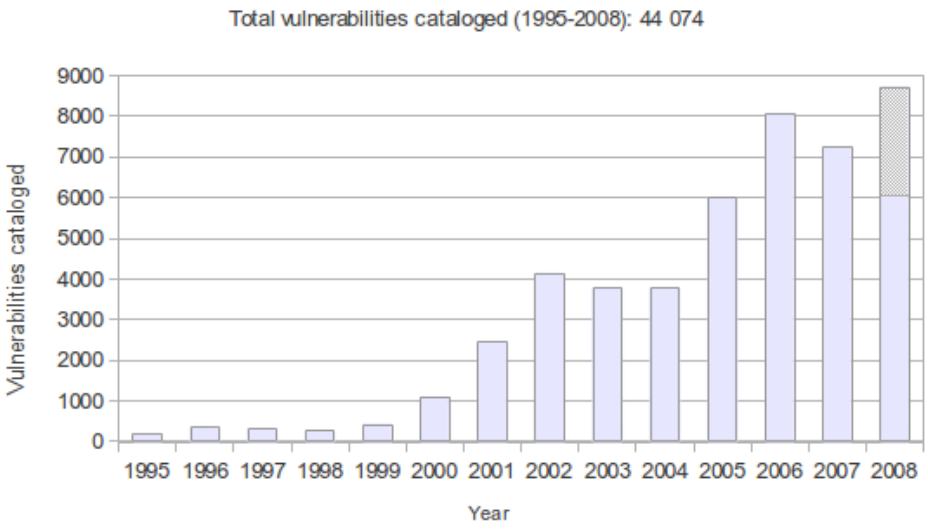

Figure 3. Total number of vulnerabilities reported to CERT [51]

The consequence of the continued new vulnerability discovery growth for the last 20 years is appearance of a great number of classifications. A purpose of each such classification is to help a software developer and a securirty specialist to study common programming errors which affect software security [26]. The first classifications became to apparent as long ago as 1970s, when the software security problem only appeared. At this moment the most widely used classifications are

the following: OWASP [77], Seven Pernicious Kingdoms [78], Fortify Taxonomy of Software Security Errors [79], and others.

2.6.1 General classification

Classifying software security vulnerabilities, a specialist investigates a way, at what moment, and in which parts of the system each vulnerability appears [39] [41] [80]. Thus the vulnerabilities can be classified in general by their origin, moment of appearance, and location. Also, the same vulnerability can be classified by several criterions.

Vulnerabilities are classified by origin on the basis of a way how they appear in the system, and the classification is closely related to software development lifecycle stages. By this criterion vulnerabilities may appear in the system during making requirements, the specification development, code writing, or deployment [39] [41].

Vulnerabilities also can be classified by their location in the system [80]. They may appear in OS components, applications, or third-party modules.

2.6.2 Seven Pernicious Kingdoms

One of the most well-known and widely used vulnerability classifications, called Seven Pernicious Kingdoms, has been developed by Gary McGraw [13] [26] [78]. According to the specification, all vulnerabilities are considered as commong programming errors from attacker's point of view, not developer's. It's the main advantage states on the fact that all vulnerabilities of this category can be easily discovered using automatic tools. The main disadvantage is it's incompleteness because it contains only known vulnerabilities. In this classification all vulnerabilities are divided by the seven categories, called kingdoms by the author:

Input data check and representation. Buffer overflow, shellcode injection, Cross-Site Scripting (XSS), format string vulnerabilities, integer overflow, invalid pointer value, SQL injection, relative file path use, and others.

Improper API use. Violation of an API contract, incorrect exception handling stategy, lack of returning values check, and so on.

Security mechanisms. Full or particular lack of security mechanisms: access control, authorization, authentification, and others.

Time and state. Lack or incorrect use of synchronization primitives, race conditions, deadlock, and others.

Errors. Lack of error handling strategy or it's incorrectness, double free of a memory block, memory leakage, NULL pointer dereferencing, and others.

Encapsulation. Important information disclosure, return of a private class field by it's public method, output of information about software environment, and others.

Software environment. Incorrect set of file access rights, keeping of unencrypted passwords or other confidential information.

2.6.3 Vulnerability databases

All existing vulnerability classifications are used in structure of vulnerability databases. At this moment there are several such kind of databases. The most well-known are ICAT [42], CWE [43], BugTraq [44], and a number of others. Such diversity leads to their redundancy. On one hand, information about vulnerability may be duplicated in each database. On another hand, it is necessary to utilize as more as possible databases to get comprehensive information. Moreover, each of the databases listed above has it's own vulnerability classification.

It can be said that the most important disadvantage of such kind of databases existence and utilization is a possibility to make statistical calculations: which vulnerabilities are appeared most often, which of them are the most urgent, an so on. Due this fact it is possible to get information about which of them are the most dangerous.

For instance, according to the CWE, all vulnerabilities are classified by three categories: unsafe interaction between the system components, risky resource management, and weak protection mechanism [81]. Vulnerabilities of the first category are related to unsafety of used data transfer methods both between components of the application and between the application and it's software environment. Vulnerabilities of the second category are related to unsafe allocation, utilization and deallocation of system resources. Vulnerabilities of the third category are related to an incorrect selection of protection techniques or even lack of them.

2.7 Software security in mobile and peer-to-peer environments

Research of the work is related to search of vulnerabilities in a mobile Peer-to-Peer (P2P) environment. Thus it is necessary to consider not only common vulnerabilities but even those that are specific for mobile P2P environments.

2.7.1 Software security in a mobile environment

Mobile environment is characterized by the fact that data and information system access is realized by means of portable wireless devices [82]. A notebook, PDA, a pager, a smartphone, a common mobile phone, and many others can be used as such portable mobile devices. Mobile environment is characterized by heterogeneity of computational devices, as well as instability of Quality of Service (QoS) performed by underlying communication infrastructure [83]. Also the following essential properties of the environment can be noted: user and network element mobility, wireless nature of communication devices [84].

From the security point of view, a mobile environment by its nature is even more vulnerable than a general network environment. The consequence of this fact is software security complication. Li Wenjia gives the following security problems that can be occured in a mobile environment [85]:

Lack of security boundaries. Mobile environment becomes more and more exposed to such attacks as passive interception, active interference in data transfer process, confidential information leakage, Denial of Service (DoS).

Threat from compromised nodes. Gaining control of one or more nodes in an environment makes it possible to perform more large-scale attacks other nodes.

Lack of central control facility. It leads to complication of attack detection and prevention because of node autonomy.

Limited power supply. It promotes to DoS-attacks performance and explosion of operations executing between several nodes together.

At this time it is popular to use mobile code and content in a mobile environment. It also somewhat affects the security. Goertzel notes the following security problems related to mobile code and content: trust problems between sender and receiver,

data integrity problems during delivery, software environment security [11]. Denial of service, data modification, confidential information leakage, data interception are risks that may be rated as the most critical [86]. To solve these problems, the following methods can be used: digital signature of mobile code, sandboxes, usage of static and dynamic analysis tools, and a number of others [11] [86] [87] [88]. Digital signature of code is an additional information attached to the code. It allows to indentify it's author and integrity. Using of a sandbox is an application execution in a special isolated virtual environment to decrease potential consequences of execution a malicious code.

Asokan gives a general security threat classification in mobile environment on the basis of notion of CIA Triad (see Section 2.1) [84]. According to the classification, security threats are divided by categories depending on an CIA Triad element associated to every threat. A threat to availability consists in a possibility to perform a DoS-attack. A threat to confidentiality consists in possibilty to perform traffic analysis or data interception attacks. A threat to integrity consists in a possibility to perform a man-in-the-middle attack or session hijacking.

At the present, it is popular to set up spontaneous and often short-term mobile environments. In this case each node shares data for other nodes. This communication way led to application of P2P architecture in mobile environments.

2.7.2 Software security in peer-to-peer environment

Network environment is P2P, if it is based on distributed architecture, and all nodes share some own resources (computational power, printers, services, etc.) [2]. The distinctions of each P2P environment are the following: lack of centralized servers, nodes are communicated with each other directly. In this case, each node may play a role of both service provider and consumer.

P2P network architecture is most often used in the cases of file exchange, distributed processing, overlay multicast, and others [2]. Depending on a specified P2P environment goal, a malicious person can aim at different purposes [89] [90]. Low level of attack traceability in such networks promotes to easy virus propagation and other types of malicious software. Setting up a botnet from nodes of an P2P network, a malicious person can perform both traffic analysis and large-scale DoS-attacks. Thus a node, a service, or transferred data can be victims of an attack.

To understand how an attacker may perform attacks in a P2P environment, it is

necessary to consider that can act as a possible attack reason. In the security area they are closely related with the notion of a threat. Security threats are classified depending on which CIA Triad (see Section 2.1) are related to them [89] [90] [91]. Threats related to availability consist in denial of a service in a network or decrease of the service workload. Threats related to integrity consist in corruption of data provided by an attacked service. It should be noted that in this case a malicious person may pose himself as a source of the transferring data. Finally, threats related to confidentiality conist in the fact that all services operating in a network are available to all nodes.

Because of the distinctive characteristics of P2P environment, it is easier to perform some kinds of attacks. These attacks are man-in-the-middle and self-replication [92]. Because of lack of a central server in P2P environments, it is quite difficult to detect a real sender or a receiver of any message. Each node of the network may potentially pose itself as other, and it permits a malicious person to use it for his purposes. Providing it's own resources each node performing some operations poses as a broker between the sender and receiver. In this case it is more easy to carry out man-in-the-middle attack for a malicious person.

To protect software against the attacks mentioned above, there is a number of the following methods. One of them is based on early detection of an attacker and has two variations depending on a mode the protection is taking place: proactive or reactive [89]. Proactive protection mode is a periodic other nodes activity check for suspiciousness. Reactive protection mode is an activity check for suspiciousness of one node by another one during their communication. A fundamentally different protection method is based on idea of reputation system that involves all nodes of a P2P network [89] [92]. In this case a node performing a malicious activitity has low reputation, and other nodes are notified about it.

2.7.3 Summary

Security assurance in mobile P2P is quite hard. It can be explained by the nature of the environment, it gives a malicious person a possibility to carry out attacks, including large-scale. Also the fact considerably complicates ways to detect the attacks by other nodes in the environment.

Considering security threats in a mobile P2P environment as threats to CIA Triad element violation, it appears that threats to availability are related to denial of

service operating in the environment, threats to confidentiality are related to a high probability of data interception and lack of confidential information, threats to integrity are related to transfering data fabrication, session hijacking, or posing as any other use by a malicious person.

Mobile P2P environment architecture permits an attacker to perform a big number of variuos attacks, and the most popular of them are man-in-the-middle and self-replication. Capability of repulsing of the attacks on operating in the environment services is fully related to a level of their safety. Additional protection methods may include be following: digital signature, reputation system, early attacker detection, and mobile code execution in sandboxes.

3 SOFTWARE SECURITY ANALYSIS

I prefer the sign: "No entry", to the one that says: "No exit".
Stanisław Jerzy Lec

The section contains general information about software security analysis techniques. The concept of software security assurance is considered. Description of secure software is given. The concept of software security analysis is described. Software security analysis techniques are considered.

3.1 Secure software concept

The security problem is one of the most critical for a variety of software. User trusts such software to perform operations most important from security point of view, for example, processing of his personal data [11]. Examples of software that should be secure are following: servers available from network, web applications, applets, daemons, as well as applications that have appropriate access rights to perform a privileged operation.

To be considered as a secure software it should be designed, implemented, configured, and maintained in such a way as to operate properly under attack by a malicious person and to reduce effects of software defects that are independent from the software [13]. Secure software has the following properties: reliability, trustworthiness, and reducibility [11]. Reliable software operates properly under any circumstances, including attacks. Software that has trustworthiness property does not contain operations that could be used to injury the software by a malicious person. Software that has reduciblity property is tolerant to attacks and can recover it's proper operating after their performing.

As much developer takes into consideration these factors as more secure software is developed by him. It allows him to engage security assurance activities related to the software. According [13], software security assurance consists of "a planned and systematic set of activities that are intended for meeting by the software security requirements, standards and procedures". McGraw gives three pillars of software security: risk management, developer's knowledge about software security, and compliance with practices specific to security [26]. Software security assurance is

based on these pillars.

As it was noted in Section 1.2, the practical part of the thesis is comprised of search of software vulnerabilities. It is explained by the fact that the project being investigated is on a testing stage of it's development lifecycle. Search of software vulnerabilites is performed by means of it's security analysis.

3.2 Software security analysis

Software security analysis is a variety of software testing. But unlike other varieties it is not based on requirements to software because it is not functional [11]. The main reason of this fact is that success of tests and fulfilment of all requirements to a tested software can not say that the software does not contain security vulnerabilities at all. It is related to complexity of software security assessment.

Software security testing is intended for vulnerability searching, not traditional errors. They are differed qualitatively. Traditional errors are detected by developer accidentally and generally they do not affect other users. On the contrary, unlike a normal user, a malicious person has some knowledge of security and experience in this area, that is interacts with targeted software differently and looks for security defectes intentionally. Consequences of his futher attack often affect other users utilizing the attacking software. For this reason, software security testing is different to traditional testing methods. Michael enumerates the following differences [93]:

- properly operating code is not always secure
- most of security requirements can not be checked by performing of traditional software testing methods
- software security testing is intended for checking the fact that tested software should not do, whereas traditional software testing methods are intended for checking correctness of tested software functionality

From a formal point of view, software security testing consists in verification of security-related characteristics. They are following [13]:

1. behaviour of software is predictable and secure
2. software does not contain known security vulnerabilities
3. software is tolerant to defects of it's environment and operates correctly in a case of exception

4. software meets all non-functional security requirements
5. software does not violate security factors

There are two approaches to perform software security analysis: testing of software security mechanisms and risk-based security testing (simulation of an attacker's behaviour) [94]. The first approach does not differ to traditional testing methods but it is aimed mainly to security mechanisms. The second approach is intended for decreasing of the level of risks to detect security vulnerability in software.

While performing risk-based software security testing, CIA Triad principles (see Section 2.1) can be utilized. The security analysis is performed in terms of security risks and a set of security requirements, if they exist. Sucessful peforming of security analysis, meanwhile, is quite difficult because it requires taking into account many details. Moreover, a person performing the analysis should be high-skilled.

Risk-based software security analysis consists in source code revision and utilization of various software security testing methods using appropriate tools [13]. Source code revision is performed at an attacker's side and it is aimed at the most important components and interfaces beetween components, including interfaces for plug-ins [11]. Software security methods are applied to testing techniques based on "white box", "gray box", and "black box" principles.

The methods mentioned above have such names because they depend on artifacts available at the security analysis performing. While performing black box testing, only binary code is applied. While performing gray box testing, both source and binary code are applied. Finally, While performing white box testing, only source code is applied.

3.3 White box software security testing techniques

As it was noted above, availablity of source code is required to perform white box testing. Applying of methods that utilize source code are most useful and popular at development stage. These methods involve static and dynamic source code analysis, and property-based testing [11].

White box testing has a number of advantages and disadvantages [95]. The main advantage is a high degree of source code coverage owing it's availability. All possible application executing paths are analyzed to find potential security vulnerabilities.

Whereas the main disadvantage is complexity of performing this kind of analysis itself. It is not rare when an analyzer gives a lot of false results, especially if testing software is sizeable.

3.3.1 Source code static analysis

The secure programming technique [30] is utilized to defense against invalid input data on all implemented software levels. It promotes faster detection of errors and rise of software security integrally. But application of the analysis is not enough [52]. It is directed to check whether calling party complies with the contract of appropriate called party. Transfering data safety is not assured.

In the listing below, a function of message logging is presented. Specially formed value of `msg` argument can be valid but it is leading to the vulnerability of `fprintf()` format function in the body of this function. Such kind of vulnerabilities can be found by means of performing static analysis.

Listing 4. Unsafe program example

```
void log(char* msg) {
  if(msg == NULL) {
    printf("Attempt to print null message");
    return;
  }

  fprintf(log_file, "%s", msg);
}
```

Static analysis carrying out leads to search of various problems such as errors committed by accident, common programming errors, including security vulnerabilities [52]. Source code static analysis helps to reduce a total number of such kind of errors as early as at development stage.

Static analysis is often performed during the process of source code revision. In this case the revision is performed sequentially. It involves the following steps [52]:

- goals of revision are established
- static analysis tools are utilized
- then accoring to the reports of their work source code revision is performed itself

- found errors are eliminated

Static analysis carrying out has a number of disadvantages [11] [52]. It can be used only as an additional tool in the process of error detection. It can be explained by the fact that error absence in the report does not mean that they do not exist in a tested application. For example, errors invisible in the code explicitly but become apparent during the application execution are referred to such kind of errors. Also, a static analysis tool report often contains a lot of false positives. Threrefore the analysis should be performed by a security high-skilled expert.

The well-known static analysis tools are the following: Splint [96], Cppcheck [97], RATS [59], ITS4 [73], PVS Studio [74], and others. RATS and PVS Studio can be used to search concurrency problems, including race conditions. ITS4 and Splint are intended to search of known vulnerabilities specific to C and C++ programming languages, for example, buffer overflow, format function vulnerabilities, and others (see Section 2.5.3).

3.3.2 Property-based testing

Purpose of any set of software tests is to examine reliability and functionality of tested software. In other words, verification of software semantic characteristics is carried out. This kind of testing is a variety of formal analysis techniques, threrefore it is utilized only after implementation of declared tested software features [11].

To perform property-based testing the following steps are carried out [98]. Specification of tested software is analyzed. On the basis of the specification, a list of the tested software properties is formed. These properties, for instance, correctness of authentication mechanism, should be verified. Then verification of each property is formalized, and an appropriate test is prepared. The process of verification is performed using source code.

The main disadvantage of property-based testing is the fact that it requires a lot of time to perform all activities listed above [11]. Especially, if the tested software is quite sizeable. The main reason of the disadvantage is a requirement to prepare strictly formalized tests.

3.4 Black box software security testing techniques

Only binary code is needed to perform black box software security testing. In the case of this type of testing, software is considered as a whole one. It responds to input data by performing predefined operations and returning appropriate output data. The principle of the testing is based on compliance between input and output data from security point of view, subject to changing of software environment [11].

Black box security testing has a number of advantages [95]. The main advantage is availability because the testing can be performed irrespective of presence of tested software source code. Another advantage is repeatability of the testing because it considers tested software as a whole one regardless of it's inner mechanisms. Finally, the testing is characterized by simplicity because it does not require knowledge about implementation details that can be complex sufficiently.

On the other hand, black box software security testing has also a number of disadvantages [95]. The main disadvantage is complication of the testing quality assessment. Another disadvantage is complexity of complex attacks modeling, targeted to a specific vulnerability.

3.4.1 Fuzz-testing

Fuzz-testing consists in random invalid data transmission to a tested application through it's entry points by the use of software environment or third-party components, and the results are verified afterwards [11]. Generated input data is usually based on alteration of valid input data. Data tranmission is performed by executing a special application called fuzzer. Usually every fuzzer is related to a specific way of data transmission, for example, by the use of HTTP protocol.

There are several variations of methods that are used to perform fuzz-testing [95]. While writing test scenarios with predefined data, boundary values can be used in data structures, packets, or messages that are input data for tested software. On the other hand, randomly generated data can be used, it is effective enough but resource-intensive. There is a subkind of this type of testing, brute force, that performs examination of all options. Finally, in some cases a whole framework can be used. It analyzes output data of tested software and determines next values of input data depending on the output data.

Fuzz-testing has a number of limitations related to specific vulnerability type that

can be found using the testing [95]. These vulnerabilities are defects related to an access control mechanism, logical and architectural errors, backdoor search, memory corruption, as well as complex vulnerabilities that are exposed by performing multistage complex attacks. The main disadvantage is requirement of appropriate fuzzer implementing or search of existing one. Moreover, in the second case, setup of the fuzzer is needed. Therefore it can be said that it is neccesary to have deep knowledge about tested software [11].

3.4.2 Binary code fault injection

The method is most widely used in combination with penetration testing to realize better how a tested application operates under attack [13]. The testing consists in asset alteration of tested software in a way that simulates aftermath of attack on the software. The goal of the fault injection is to examine state, behavior, and characteristics of tested software security under such circumstances [11]. Fault injection gives the following benefits [11]:

- possibility of environment anomalies simulation without understanding of the fact how they can be exposed afterwards
- possibility of selection which anomalies are simaluting in a specific test without setup of entire testing environment
- ease of automation

3.4.3 Reverse engineering

The method comprises executing code analysis and can be used to find security vulnerabilities. In this case, a tester is playing the role of an intruder and performs tested software analysis by the means of various special tools. It allows to assess tested software security from outside and find security vulnerabilities visible from this point of view [13]. There is a lot of various tools used to perform this operation [99]. Eeach of them is aimed to perform certain actions (debugging, for instance). Therefore a set of tools is selected depending on tested software.

Disassemblers are most widely used tools that are utilized to perform reverse engineering. They decode binary code to text to make it human-readable. The most well-known disassemblers are IDA Pro [100] and ILDasm [101]. IDA Pro allows to carry out of x86-instruction disassembling, and ILDasm allows to carry out .NET

[102] bytecode disassembling.

Unlike disassemblers, debuggers can be used to perform step-by-step execution of tested software. They give a chance of retrieveing of information about current application state in every moment. Also they can be used to alter workflow of tested software. The most well-known debuggers are OllyDbg [103] and WinDbg [104].

Decompilers are tools that have ability to decode compiled binary code back to source code. At this moment such kind of operation can be performed only particularly because a lot of information contained in source code is not used during compile time. Moreover, the information is not remained in compiled code. The most well-known debugging tool is .NET Reflector [105].

While performing reverse engineering, system monitoring tools are often used. They allow to get a lot of information about interaction between a tested software and it's software environment. For example, they can let to know which system functions are called or which files are accessed by a tested software. The most well-known sets of such kind of tools are the following. Tools provided by SysInternals [107] are the most popular on Windows platform [106]. A different sets of tools, such as ps, lsof, and others, can be used on other platforms [108].

Reverse engineering, like any other security testing method, has a number of disadvantages [52]. The main disadvantage is complexity of executing code decoding and analysis. Additional complications consists in the fact that it is necessary for a specialist to know the basics of working with various tools. Also, as it was noted above, composition of used tools can differ completely depending on testing software.

3.4.4 Black box debugging

Black box debugging consists of the following activitites [11]. Initially, tested software is divided by parts. Then each part is tested individually. The goal of the black box testing is to find vulnerable parts of a tested software. Especially the method is useful when third-party closed source components are used by a tested software.

Black box testing is aimed to gather debug information about interaction of a tested software with it's third-party components and software environment. It is achieved by monitoring of external behaviour of tested software or it's part. By using this technique, a set of vulnerabilities, such as error skipping, can be found and eliminated

[109].

Performing black box debugging implies utilization of appropriate tools. Gathering of debug information about tested software can be performed, for example, by file system monitoring. The most well-known tools that support such functionality are IDA Pro [100], OllyDbg [103], and tools provided by SysInternals [107].

3.4.5 Vulnerability scanning

Automatic vulnerability scanning is one of the most useful ways of software security testing and is performed by the means of appropriate tools. The principle of work of such kind of tools is that they interact with testing environment by passing of template data, called signature [11]. Each vulnerability scanner contains a lot of such signatures, and each of them is related to specified type of security vulnerability.

Utilization of vulnerability scanners is the most effective during security analysis of third-party components before the moment of their real usage by tested software [11]. Also, it is the most useful when it is performed before stage of penetration testing to decrease a number of testing scenarios [11].

Utilization of such kind of tools has a number of disadvantages [11]. Usually a vulnerability scanner is able to find only 30 % of all types of vulnerabilities. Since efficiency of a vulnerability scanner depends on a number of signatures in it's signature database, it is necessary to update the database periodically.

Vulnerability scanners are divided by several categories depending on a way they interact with tested software. These categories are port scanners, web application vulnerability scanners, CGI scanners and others. The most well-known tools are Nmap [110], Nessus [111], ITS4 [73], and others.

3.4.6 Penetration testing

Penetration testing is one of the most widely used software security methods. It performes on last software development stages and consist in analysis of tested software behaviour under attack [112].

Penetration testing has a number of characteristics [11]. While performing of the testing, tested software is executin in normal mode, without establishment of appropriate testing environment. Penetration testing as opposed to other testing methods

is also directed to search of architectural and design errors. Finally, penetration testing scenarios take into consideration the worst cases that can damage tested software greatly.

The testing process is usually consists of the following steps [113]. The first step is assessment of security risks, such as denial of service, information disclosure, data corruption, and others. The second step is testing planning: which components should be analyzed, how the testing should be performed, and so on. The next steps are tests running and working out of report about found security vulnerabilities.

By-turn, penetration testing has a number of disadvantages [11]. The main disadvantage is the fact that it is performed on late software development stages. Therefore found security vulnerabilities may require significant costs for their elimination.

3.5 Gray box software security testing techniques

Gray box testing, in some sense, is a hybrid solution that involves elements of white box and black box testing techniques. Therefore it can be utilized as an additional tool for these kind of testing techniques. It's main advantages are availability and high degree of code coverage by tests [95]. The main disadvantage consists in the fact that gray box testing is quite difficult to perform [95].

3.5.1 Dynamic code analysis

Dynamic code analysis by it's nature is execution of tested software using special tools. The goal is to detect abnormal actions and to notify about them instantly [11]. General operational principle of these tools is to run tested software in special environment that is used as a layer between the tested application and it's environment. After that, the tools analyze all operations performed by the tested application. If source code of the application is available, information about location in source code of each anomaly is displayed. Gray box testing tools often can exactly recognize an anomaly as a consequence of an error.

As opposed to static analysis, dynamic analysis makes it possible for a tester to search security vulnerabilities which can be detected by interaction of tested software with user, environment, or it's own component. By performing dynamic code analysis, one can find the following kind of errors: buffer overflow, format string function vulnerabilities, pointer vulnerabilities, and others [114]. The following well-known

tools can be utilized: StackGuard [115], Libsafe [64], FormatGuard [62], Valgrind [116], Helgrind [75].

3.5.2 Source code fault injection

Source code fault injection is a method used to test fault-tolerance of software [117]. The method by it's nature consists in manual injection of various defects to source code of tested software. The aim of this is, for example, to verify correctness of exception processing strategy [11]. Manual fault injection makes it possible to check correctness of work of both whole tested application and it's component under conditions of fault existence in environment or in one of it's other component. Software fault emulation makes it possible to determine degree of fault-tolerance of tested software.

Precise fault emulation makes it possible to determine aftereffects of an error presence. Threrefore it can be said that support of precise fault emulation is the main task of this kind of testing [117]. Fault injection is useful in case of searching errors, such as incorrect usage of pointers and arrays, race conditions, and others. The method makes it possible to find erros at development stage.

By-turn, the method has some disadvantages [118]. It can not be utilized if source code of tested software is not available. Other disadvantages are related to possibility of normal work-load violation and changes in structure of original copy of tested software.

4 PEERHOOD NETWORK ENVIRONMENT

Inspiration usually comes during work, rather than before it.
Madeleine L'Engle

This section contains general information about PeerHood project. The concept, goals and key requirements to the software are considered. A high-level architecture of the existing PeerHood implementation is examined. A question about user data security and software security of the current version of the project is formulated.

4.1 Concept

The aim of PeerHood is to carry out personal communications between devices in a network neighbourhood regardless of underylying network technology. It is intended for usage by PTD, so it is supposed to use the software mainly in mobile environments. Therefore the PeerHood concept can be considered as a mobile P2P neighbourhood [3].

PeerHood is a middleware responsible to provide transparency of unitilization of user services, both remote and local. User applications are located on a higher level in regard to PeerHood. Network plug-ins are located on a lower level in regard to PeerHood. They perform operations specific to a network technology directly. The concept is schematically represented on Fig. 4.

PeerHood carries out tracking of other devices in a network neighborhood in proactive manner switching between network technologies supported by a given mobile device. Also, is has the following function characteristics [14]:

- detection of other devices using different network technologies
- discovery of services from other devices
- advertising own local services to other devices
- monitoring status of devices in network neighborhood

The development of a project based on the concept described above has been being conducting in Communications Software Laboratory of Lappeenranta University of

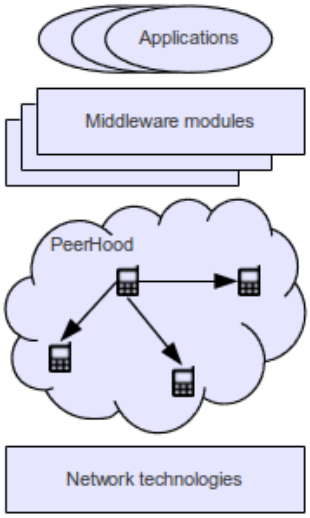

Figure 4. PeerHood concept [3]

Technology for several years. Initially the project was being developed with the participation of Nokia company [119]. A problem related to social communications performing using PeerHood was studied [82]. User applications for UMSIC project [120] were implemented [121]. At this moment PeerHood is a free software, it is distributed under GPL 2 license [122] [123].

Current PeerHood implementation can be utilized both on desktops and mobile devices. It is obtained by the fact that it's components are based on Qt framework [17], which is intended to develop cross-platform software.

4.2 Goals and key requirements

The motivation to develop this project is a need to solve a variety of problems. According to PeerHood specification, it's development is intended to achieve the following goals [124]:

Proactivity Proactive service discovery on mobile devices.

Connectivity To provide transparent communication between application regardless of underlying network technology.

Reactivity To provide information events related to services, connections or device resources.

To achieve the goals listed above it is necessary to solve a set of tasks. They are the following [4]:

- To provide a communication environment where communication between devices is based on P2P approach. It allows devices to communicate without utilization of central servers
- To provide a special software library to allow applications use a network technology through unified interface. It leads to facilitation of development of new applications because they does not depend on underlying network technologies.

It is necessary for PeerHood to have such functional characteristics which enable support of proactivity, connectivity, and reactivity. Supporting them by PeerHood leads to proper operating of user applications. A set of requirements to the project can be referred as these characteristics [125]. A set of functional requirements to the PeerHood project are listed below [124] [126]:

Device discovery System must be able to discovery other PeerHood capable devices within range and the same device neighborhood. Device detection can depend on used network technology.

Service discovery System must be able to discover services from the local device and other PeerHood devices in the device neighborhood. System must have capability to read service attributes as well with service discovery.

Service sharing PeerHood must provide mechanism to register services and use them by applications or middleware components. Services can locate services working on a local or a remote device. The PeerHood system must advertise registered services to other devices in a PeerHood neighborhood.

Connection establishment PeerHood must provide ability of connect to one or more other PeerHood devices in a PeerHood neighborhood. Connect establishment must be transparent for using underlying network technology.

Active monitoring of a device PeerHood must provide way to set a selected device in a neighborhood under active monitoring. In the active monitoring state, a PeerHood client is notified when the device under monitoring is out of range or when it comes back in the range. Proper response time and range are network technology dependent attributes.

Data transmission between devices PeerHood must provide data transmission between connected PeerHood devices. PeerHood should not take care of data being transferred. PeerHood user must take care of data endianess and word length of data.

Seamless connectivity PeerHood should provide way to change used active network technology automatically if established connection weakens or breaks. PeerHood should always provide the best possible connections for a user. Established connection should be possible to monitoring for detecting connection changes, which might cause change of used network technology.

A new additional functional requirement was added to the specification (but it is not supported by the current implementation of PeerHood) [14]:

User control PeerHood could provide ability to control the following PeerHood functionalities: whether PeerHood is active, a list of provided services, a list of accepted services.

Also, there is a set of non-functional requirements to the PeerHood project. They are listed below [124]:

Network management PeerHood should be able to manage a specific network and events from the network. In addition, PeerHood should check availability of network and get notifications of changes of the network.

Component management PeerHood should provide events to PeerHood client of changes and suspensions of discovering functionalities. PeerHood operates on mobile devices where memory and power consumptions have to take care. Due to that, used device environment is dynamic. As, if network interface might go power saving state or it can be closed for freeing memory to other applications.

Communication concurrency base PeerHood must support concurrent execution, in that multiple connections are used and they need to get execution time evenly. The only exception for use of multiple simultaneous connections is if used network technology limits multiple connections on the hardware level.

Event interface PeerHood must provide event interface for be able to notify dynamic changes to PeerHood client and itself.

Plugin architecture for networks PeerHood must provide interface for its functionalities to plugins. Network plugins implements abstractions of connectivity and device monitoring functionalities. In addition, plugins handles device detection and service sharing.

As it was noted above, all these requirements should be met to achieve the goals of PeerHood. Functional requirements are required to define a list of features which should be supported by the developing software, and a list of it's future components as well. Whereas development of non-functional requirements in the first place defines an architecture of the developing software [127]. The architecture of PeerHood is considered in the next subsection.

4.3 Architecture

From architectural point of view, PeerHood is comprised of the following components: daemon, shared library and a set of network plug-ins. They are represented on Fig. 5. The components together are middleware that provides communications in mobile P2P environment regardless of underlying network technology.

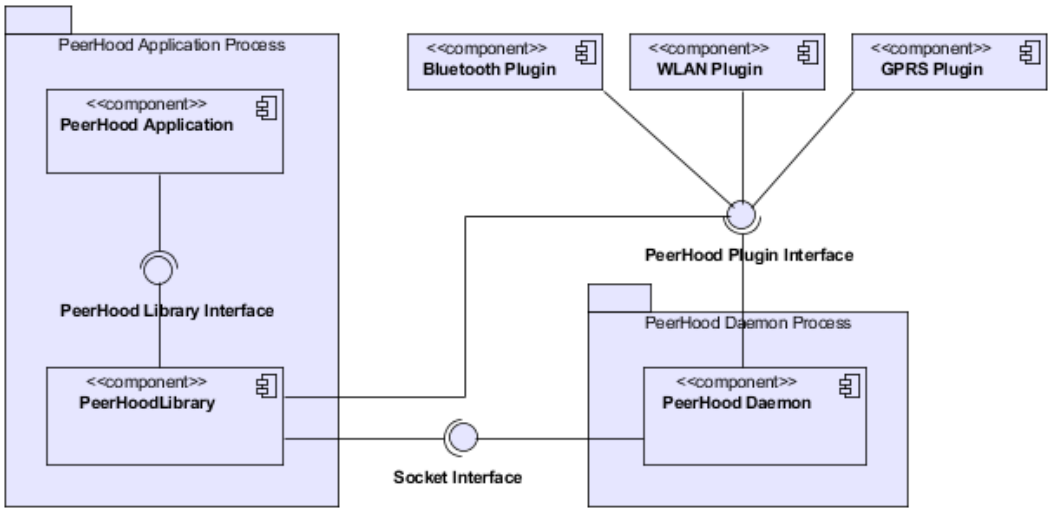

Figure 5. PeerHood components [14]

PeerHood daemon is an independent process which is directly responsible for providing communication between devices using a given network technology. It is accomplished by utilization of network plug-ins. PeerHood library is a component which is responsible for interaction with user applications via specified API and has a role to play as glue between daemon and applications. That is the way the process of communication between devices is encapsulated for applications.

4.3.1 Daemon

Daemon is a background process that is intended for discovering other devices and services in neighborhood. The operation is quite costly and consumes a lot of resources and hence it is implemented as a separate component. In this case during initializing of an application it is not necessary to gather information about neighborhood, so time needed for an application to start to work is reduced. The component has the such name because it works in background and fully corresponds to the POSIX "daemon" term [128].

By-turn, the daemon also gathers information about other devices and services in a neighborhood. It provides operation of local services and gives possibility to use them by other devices. It sends information about local services to other devices in a neighborhood. Data transmission, as well as registration and deregistration of local services by applications as well are realized as a shared library, which is another PeerHood component.

4.3.2 Library

The component has a role to play as glue between applications and daemon. It is a dynamic link library that is directly used by applications. Interaction with them is performed via specially designed API presented in Appendix 6.1. For applications the library is a medium that used to get information about other devices in neighborhood, to do data exchange with them, to utilize remote services, and to make local services available for other devices. Interaction of the library with daemon is performing via socket-based interface [129].

4.3.3 Network plug-ins

Network plug-in subsystem is intended for making support of new network technologies easier. Each plug-in performs operations specific for a given network technology. Also, it should have a specially designed API used by the daemon. Factically, plug-ins are dynamic link libraries which are loaded by daemon during PeerHood operating. Current PeerHood implementation has support of the following network technologies: Bluetooth [1], Wireless Local Area Network (WLAN) [130], and General Packet Radio Service (GPRS) [131].

4.3.4 User applications

Applications are located on the top of PeerHood architecture. It means that operations specific for a given network technology are performed by components located on lower layers. In such the way transparency of interaction between applications is accomplished.

To provide such interaction, "client – server" model is used. That is an application can both utilize a service provided by other device and provide an own service to be utilized by other applications. Interacting applications, meanwhile, can be physically located on both one or different devices. At the same time an application can both provide a service and utilize it. Finally, an application can perform tracking of other devices in a neighborhood. When a device appeares in a neighborhood or leaves it, the application gets a notice about it.

4.4 Security problem

At the present the problem related to information security is very important. It is especially actual in the context of PTD when private user data is involved in communication process. Hence, it is necessary to take into account security risks related to vioalation of integrity, confidentiality, and availability of the data.

There is a direction of further PeerHood development that is related to security assurance and is one of the most priority [124]. The main reasons of this are the following:

1. PeerHood has access to private user data stored on a device
2. PeerHood can transfer private data to other devices during communication
3. PeerHood is a network software, so data transfer security depends on security of underlying network technology
4. security of private user data also depends on PeerHood software security itself

According the PeerHood specification [124], implementation of a security mechanism for PeerHood is one of the most important tasks. The first step to implement the mechanism is to perform software security analysis of the project to find out, which kind of security vulnerabilities PeerHood has, which places are most susceptible from

security point of view, and so on. The next section of the thesis deals with this problem.

5 PEERHOOD ANALYSIS

It is easier to be critical than correct.
Benjamin Disraeli

This section deals with security analysis of PeerHood network environment. PeerHood specification analysis is made, PeerHood security risk modeling is performed. Software security testing of the project is executing. PeerHood source code revision intended to find software vulnerabilities is carried out.

5.1 Goals and task statement

Practical part of the work deals with security analysis of PeerHood network environment. It plays an important role because it allows give an answer to the question of the research: whether PeerHood contains software security vulnerabilities or not. Thus analysing PeerHood security it is necessary determine whether it is secure software or not (see Section 3.1).

Following Michael [93], it is necessary to conduct a number of activities to analyse software security. Concerning PeerHood they are the following:

1. specification analysis
2. estimation of security risks
3. carrying out the activities related to security testing
4. obtained results analysis

PeerHood specification analysis gives information about which factors may affect it's security, as well as composes CIA Triad principles concerning the project (see Section 2.1). It enables to get a set of formalized characteristics determining fault-tolerance of the project, and that means it's safety. Such characteristics compose requirements to the project. Thus all the following tasks aim at the characteristics analysis.

The results of the specification analysis enable to make PeerHood security risks estimation thoroughly. And on the basis of the results it is possible to compose a set of software security testing methods.

PeerHood security testing aim at check of a compliance of security requirements

to with their realization in the current implementation of the project. The testing results give a possibility to realize whether PeerHood contains security vulnerabilities or not. And it means to determine are PeerHood CIA Triad principles violated or not, can it operates properly under attack or not.

5.2 Specification analysis

PeerHood specification contains the following information. The project goals define that should the software must to do. The functional requirements define which functional characteristics should be have PeerHood to achieve it's goals. Non-functional requirements define a structure of the project.

5.2.1 PeerHood goals analysis

The PeerHood goal consists in personal communication support in a mobile P2P environment regardless of underlying network technologies. It is achieved at the expense of proactivity, connectivity and reactivity support (see Section 4.1).

Connectivity support composes a logical level of the applicatio that is responsible to seamless communication between mobile devices. According PeerHood concept, it is needed to link "Network technologies" elements with each other (see Fig. 4).

Reactivity support composes a logical level of the application that is responsible to communication between user services. According to PeerHood concept, it is needed to link "User applications" elements with each other (see Fig. 4).

Proactivity support composes a logical level of the application that is responsible to connect the levels described above. According to PeerHood concept, it is needed to link "Network technologies" elements with "Middleware modules" elements and "User applications" elements with "Middleware modules" (see Fig. 4).

Thus correctness of all PeerHood components functioning on each logical level defines a possibility of successful solution of all tasks targeted by the project. And their fault-tolerance defines a possibility of correct PeerHood functioning under attack.

5.2.2 PeerHood requirements analysis

Correctnes of each PeerHood component functioning is defined by meeting of all requirements related to the component, functional and non-functional. And all

requirements in the aggregate aim at the project goals achievement.

Connectivity is achieved by meeting the following requirements: seamless communication, connection establishment, data transfer between devices, architecture of network plug-ins, and network management. It's achievement conists in seamless data exchange between devices with a possibility of switching between available network technologies. It is achieved via network plug-ins, each of them is aimed at support of a specified network technology, it has special interface for performing network-specific operations (data transfer, connection establishment, and others).

Proactivity is achieved by meeting of the following requirements: device discovery, device detection, component management, and provision of the basis for simultaneous client serving. It's achievement consists in provision of up-to-date information about the network environment to one or more client. It is achieved via middleware modules that connect components of other levels. The modules provide information about the network environment to high level components and notify them about changes. Low level component in their turn are utilized by the modules to perform data transfer directly and to process received from they notifications about network events.

Reactivity is achieved by meeting the following requirements: service discovery, shared service utilization, user control, provision of client interface to receive events. It's achievement conists in shared both local and remote services utilization and the system control by user applications. It is achieved via high level modules that are aimed at execution of user operations within the concept of the services.

Realization of the fact how all PeerHood components are functioning enables to define a set of functional characteristics of the project. The charactestics affecting successul each PeerHood goal achievement are given in Table 1: proactivity, connectivity, and reactivity. Their analysis enables to define components of information security concering PeerHood.

From the security point of view, influence on each functional characteristic is arising depending on it's nature. It depends on the fact how the characteristic is related to each CIA Triad principle: availability, confidentiality, and integrity.

Availability is related to the following functional characteristics: retrieving of up-to-date information about devices and services in the network environment, simultaneous serving of several clients, as well as user control. It can be explained by the fact

Goal	Characteristics
Connectivity	data transfer, retrieving of up-to-date information about devices in the network environment (using different network technologies)
Proactivity	retrieving of up-to-date information about available devices in the network environment data exchange between devices, simultaneous serving of several clients (local or remote)
Reactivity	retrieving of up-to-date information about available devices in the network environment (local or remote), user data transfer, user control

Table 1. PeerHood functional characteristics

that fault-tolerance of these characteristics is related to the fact of a possibility of information retrieving, user control, simultaneous serving of several clients.

Confidentiality is related to the following functional characteristics: data transfer between both the system components and devices. According to PeerHood concept, sent user data are transferred on all levels of the application (middleware modules, network plug-ins, network). Thus there is a possibility to intercept the data.

Integrity is related to the all functional characteristics except user control and simultaneous serving of several clients. They are related to user data transfer or information about services and devices. Thus there is a possibility of their modification, fabrication, and other types of attacks.

It should be noted that each PeerHood component can potentionally have an access to system files, including user data. This facut in its turn also affects the project security because the data can be corrupted or hijacked. Thus a potentional access to system files is related to the problem of their integrity and confidentiality.

Violation of any CIA Triad principle considered above leads to PeerHood security non-assurance. There is a lot of security risks that affects the each principle.

5.3 Security risk modeling

PeerHood specification analysis enables to determine what composes a functionality of the project. But it is necessary to investigate what influences on it's fault-tolerance.

PeerHood security risk analysis allows to do it.

Knowing PeerHood functional characteristics it is necessary to determine how they are implemented in the project. According to it's architecture, PeerHood is comprised of the following components: a daemon, a shared library and a set of network plug-ins (see Section 4.3). How they are implemented and interact with each other it is shown on the PeerHood deployment diagram (Fig. 6).

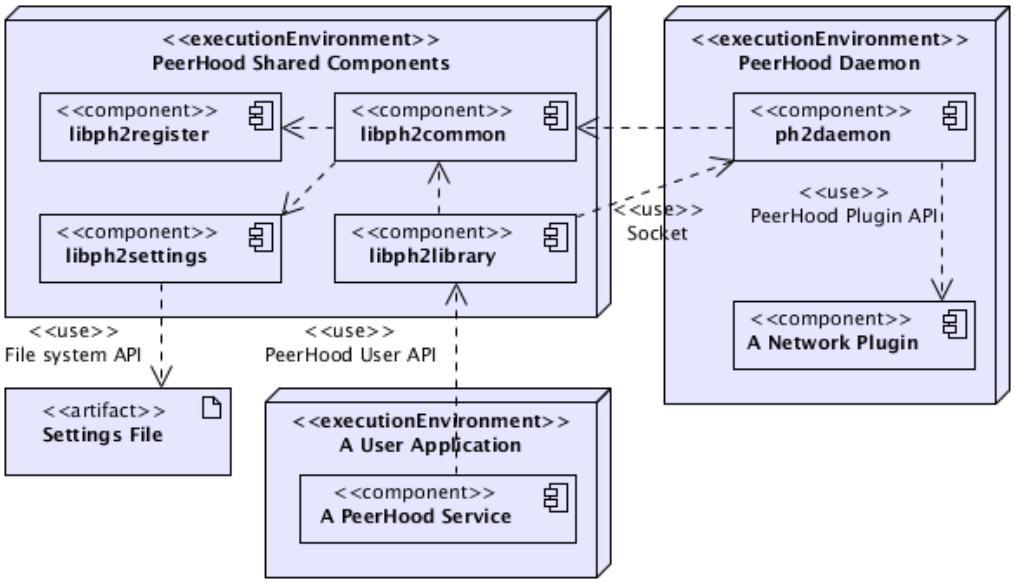

Figure 6. PeerHood deployment diagram

User applications via services interact with library by the use of specially designed API. On one hand, the library itself utilizes common components to register a service, to perform PeerHood user control, and to perform other activities. On another hand, the library interacts with the PeerHood local daemon. The daemon utilizes available network plug-ins to interact with other mobile devices and also it utilizes common components to process the configuration.

Thus it is possible to compose the following pairs of interacting with each other components. These are as follows:

- Network plug-in – Network plug-in
- Daemon – Network plug-in
- Daemon – Common components
- Daemon – Daemon

- Common components – File system
- User library – Common components
- User library – Daemon
- User application – User library

Each of the pairs is responsible for support of some functional characteristic, so it is associated with some PeerHood security component. Interaction analysis of each pair and security risks related to the pair are given below.

Network plug-ins are utilized to support a specified network technology and they are responsible for support of connection between devices, as well as data transfer between them. From the security point of view, the following security risks are related to this fact. Transferred data can my important, thus they can be attacked; they can be hijacked, corrupted, fabricated, and so on. They should be protected on the level of a utilizing network technology or a protocol, as well as by encryption. On another hand, it is possible to attack a utilizing network technology itself. Thus, if an implementation of such connection is not fault-tolerance, then it may lead to violation of all CIA Triad principles concerning PeerHood.

The daemon performs interaction with network plug-ins via API for data exchange. They data can be various: user data, information about available devices and services in a network environment. Thus each network plug-in should be trusted for the daemon. From the security point of view, there is a risk of malicious operations performed by a plug-in and targeted to the daemon. It can lead to violation of integrity and confidentiality principles concering PeerHood.

PeerHood is an configurable software. Configuration software facilities are provided by the common components. From the security point of view, data received from the components may contain incorrect values. It can influence on correctness of PeerHood functioning, so it can lead to violation of availabilty principle concerning PeerHood.

Interaction between daemons and between the user library and a daemon is performed via communication through a socket. Transferred data can contain both user data and information about available devices and services in a network environment. From the security point of view, interaction with a daemon functioning on another device can lead to performance of malicious operations targeted to the transferred data. Also the following alternative is possible. An application masking as a remote

daemon or a user library can attack the daemon itself to knock it out. Thus the all CIA Triad principles related to data can be violated concerning PeerHood.

As it was noted above, PeerHood software configuration facility is supported by the common components. The configuring process itself consists in read of a specified configuration file and provision of values storing in the file. In particular, a file system path to network plug-ins is obtained by this way. From the security point of view, there is a possibility to perform unathorized alteration of the configuration file, and it can affect PeerHood fault-tolerance and security. For example, there is a risk of a non-trusted plug-in loading, an the plug-in may perform some malicious operations on transferred data. Thus it leads to violation of integrity and confidentiality principles related to transferred data concering PeerHood.

In summary, one can conclude that the project has "vulnerable" places. The highest possibility of an attack performance is related to the places. They are shown as links between components on the PeerHood deployment diagram (Fig. 6). "Vulnerable" places are be the following:

- the configuration file
- the socket used by the daemon to serve several clients
- API for network plug-ins
- API for configuration facilities
- API for user applications

Since PeerHood is aimed at functioning in a mobile environment it is necessary to take into consideration possible security risks related to the environment. A mobile environment is characterized by a risk of performance of malicious operations related to transferred data, confidential user information leakage, or denial of service.

On the other hand, PeerHood network environment is based on the P2P communication concept. From the security point of view, availability of a service for all other devices in the environment is characterized by a threat of an attack targeted to transferred data, operating services, and the device itself.

One can conclude that taking into account mobility and P2P of the environment only underlines the significance of the security risks considered in this subsection. Thus this fact concerning PeerHood promotes to growth of a possibility of a transferred

data integrity and confidentiality violation, as well as availability of devices in the network environment.

To estimate PeerHood security risks it is necessary to perform it's security testing. It enables to estimate fault-toleance of each PeerHood component. Activities related to security testing are considered in the next subsection.

5.4 Security testing

A set of utilized software security testing methods depends on source code availability of a tested application and it's third-party components. PeerHood is an open source software, the source code is available on [123] (the *81eeb64* version of 11.08.2011 is used for the testing). Thus it is necessary to make a selection of appropriate testing methods of the ones considered in Section 3. Because PeerHood source code is available, in the first place it is necessary to examine applicability of black box testing methods.

5.4.1 Testing methods selection

Fuzz testing by its nature is quite cost-based, and the results can be obtained using easier methods, namely using entry points source code revision. For PeerHood the entry points are the "vulnerable" places listed in Section 5.3.

PeerHood reverse engineering conducting is unnecessary because of the source code availability, and also closed third-party components are not utilized by the project. PeerHood utilizes Qt framework [17] as a platform, thus it can be said that PeerHood security depends of security of this platform.

PeerHood black box debugging is also unnecessary for the same reasons. More comprehensive analysis of a way how an application interacts with software environment can be peformed using white box testing methods.

Utilization of vulnerability scanners is also unnecessary for the same reasons. More qualitative results can be obtained using source code revision of PeerHood "vulnerable" places. Source code revision aimed at error handling mechanisms analysis and checking the "security in depth" principle can be performed instead of this method. PeerHood is not so big project.

On the contrary, resource and component fault injection may be effective. Using

this method it is possible to estimate how much PeerHood is fault-tolerant to it's environment faults (resources, third-party component, software environment).

Penetration testing method is aimed at estimation of an application fault-tolerance. Risk estimation, "vulnerable" places analysis, and search of vulnerabilities put together a set of activities that are used for penetration testing performance. Thus it can be said that PeerHood security analysis implies penetration testing.

The same can be said about property based testing. PeerHood security testing is aimed at estimation of such characteristics as the CIA Triad principles concerning PeerHood. A detailed list of such characteristics is gived below:

- availability of data transfer
- confidentiality and integrity of transferred data
- availability, confidentiality, and integrity of up-to-date information about devices in the network environment
- availability of exchange of user data and information about available devices between devices in the network environment
- possibility of simultaneous several clients serving
- availability, confidentiality, and integrity of up-to-date information about available services in the network environment
- availability of exhange of information about available services between devices in the network environment
- possibility of user data exchange between devices
- confidentiality and integrity of user data transferred between devices
- possibility of the system control

Static and dynamic analysis conducting is aimed at search of vulnerabilities on source code level and at runtime respectively. Utilization of various tools promotes to more detail analysis of PeerHood. But utilization of these security testing methods does not enable to perform comprehensive analysis of the application. Thus it can be said that conducting of these methods is reasonable and necessary.

Using automatic security testing methods one can find only a limited number of vulnerabilities. Thus it is necessary to perform source code revision to carry out more comprehensive PeerHood analysis. As it was already noted above, it is especially urgent for the "vulnerable" places of PeerHood.

5.4.2 PeerHood code static analysis

The goal of static analysis is a search of source code level security vulnerabilities (see Section 3.3.1). It conists in utilization of automatic tools intended for search of programming errors affecting the application security. To carry out PeerHood static analysis the following tools are used: Cppcheck, RATS, GCC compiler features related to security.

GCC compiler allows to perform search of format string vulnerabilities, array bounds checking, detection of uninitialized data, and others types of errors at compile time [132]. To perform this operations, the tool is executed with the special `-Wall` argument which switches execution of such operations on. PeerHood compilation passes without any errors and warnings. This says that vulnerabilities are not found using this tool.

Cppcheck is intended for searching the following types of errors: array bounds crossing, exception safety, memory leakage, uninitalized variable values, and others [97]. Utilization of this tools concerning PeerHood has not found any errors (see Appendix 6.1). Only some reccomendations are given but they are not related to security.

RATS is intended for search of buffer overflow vulnerabilities, race conditions, and other errors [59]. This tools gives only several security warnings but source code revision of the appropriate blocks of the code does not detect any errors (see Appendix 6.1). The first warning is related to a possibility of overflow of the buffer that stores ICMP message data transferred by a network plug-in. Utilization of the buffer is carried out securely. The second warning is related to a possibility of a race condition state when several signals are registering via `signal()` function at the same time. But only signal is registered in the application. The last warning is related to non-safety of the `gethostbyname()` function because a returned by the function value can be fabricated from without. But software environment of the project (platform or OS) is responsible for the safety of this function, not PeerHood itself.

Thus PeerHood static analysis has not found any security errors on source code level. It can be explained by the fact that the application actively utilizes facilities of the Qt framework.

5.4.3 PeerHood code dynamic analysis

The goal of dynamic code analysis is a search of security vulnerabilities at runtime (see Section 3.5.1). It consists in utilization of automatic tools intended for abnormal activity detection, search of errors related to interaction between a tested application and it's software environment, and detection of other defects affecting security of the application. Using these tools the following runtime errors can be detected: buffer overflow, format string vulnerabilies, memory leakage, and others. To perform PeerHood dynamic analysis Valgrind and Helgrind tools are utilized.

Valgrind tool is intended to detect runtime errors listed above and errors related to memory allocation [116]. Utilization of the tools has not detected any abnormal activities or runtime errors (see Appendix 6.1). It should be noted that the tools has detected a potentional memory leakage. But the appropriate PeerHood source code revision has shown that the leakage is related to inner mechanism of Qt implementation.

Helgring tools is intended for synchronization error detection, including race conditions [75]. Utilization of the tools has not detected any errors related to incorrect use of synchronization primitives in the application (Appendix 6.1). The tool, in turn, has detected possible race conditions in many places. But they are related to inner mechanisms of Qt implementation.

Thus PeerHood dynamic analysis has not detected any runtime errors. However, potentional errors are detected in Qt framework that is used by PeerHood. Hence one can conclude that PeerHood security depends on security of this platform.

5.4.4 Fault injection

The goal of fault injection is a tolerance estimation of a tested application when it's component are functioning incorrectly or it's resources are not corrupted (see Section 3.4.2). It conists in fault injection into every component of a tested application, after that the appropriate interaction checking is carried out.

PeerHoos is comprised of the following interacting components: a set of network plug-ins, the daemon, common components, the user library, user applications, the configuration file (see Section 5.3). The interaction is peformed in these ways:

- dynamic loading of shared objects (libraries) and further interaction via API

(the daemon – network plug-ins, the daemon – common components, the user library – common components, a user application - the user library)

- interaction via socket (the daemon – the daemon, the user library – the daemon)
- interaction with file system (the user library – the configuration file)

Binary fault injection into a shared object (library) can be performed in two ways. The first one affects an interface part of the object. In this case an error occurs during dynamic binding of the object, this leads to it's failure. The second one affects only inner functioning change of this object, and this leads to it's incorrect functioning. It can affect a tested component, for instance, return of incorrect or even malicious data. To estimate tolerance of a tested component it is necessary to perform analysis of the API utilized to perform this interaction. Because PeerHood source code is available, the best way to perform this operation is to revise appropriate blocks of the source code.

The same conclusion can be made concerning the components interacting via network socket (the daemon and the user library). Received in the such way data may be incorrect, thus it is necessary to estimate tolerance of these components. Because PeerHood source code is available, the easiest way to perform this operation is to revise the appropriate blocks of the source code.

Common components comprise configuration facilities. PeerHood stores it's own configuration in a specified file with a given format. Lack of this file or violation of it's format leads to PeerHood failure. The configuration file contains a file system path to network plug-ins loaded by the daemon. In the current daemon implementation there is no default value of this key, thus lack of the key leads to inability for network plug-ins loading and the daemon proper functioning.

The current PeerHood implementation does not comprise a special install script. At the present, deployment of the application conists in a copy of it's libraries, executing modules, and resources to a specified directory. The necessary from the security point of view access rights to these files are not assigned.

This can lead to PeerHood modules compromise. The consequence of this fact is a possibility of the application failure or a possibility of an attack performance on data transferred between the modules.

PeerHood also utilizes a text file to store it's own configuration there. The file is also readable and writable for other applications. Thus there is a risk related to an

unathorized alteration of the file. The goal of this alteration can be, for example, a change of file system path to network plug-ins location with further malicious plug-ins loading by the daemon.

Using this method concerning PeerHood, a vulnerability related to unsafe deployment of the project has been detected. Exploitation of the vulnerability by an attacker leads to violation of all PeerHood security components (see Section 5.4.1).

5.4.5 Source code revision

The goal of PeerHood source code revision is to search of vulnerabilities affecting it's security. According to [93] [133], such process is comprised of the following steps: revision goals establishment, selection of potential vulnerability classes, and further their detecion in source code. The PeerHood revision goal consists in estimation of it's security components listed in Section 5.4.1. And particular attention is given to these blocks of code that corresponds to "vulnerable" places at that (see Section 5.3). Seven Pernicious Kingdoms vulnerability categories can be used as the such vulnerability classes (see Section 2.6.2). These are as follows: input data checking and representation, improper API use, security mechanisms, time and state, errors, encapsulation, software environment [78].

Input data checking category consists in the following vulnerabilities: buffer overflow, code injection, format string vulnerabilities, integer overflow, invalid pointer value [78]. C/C++ arrays are not used in PeerHood source code with the exception considered in Section 5.4.2. More safe Qt alternatives are used instead of them: `QString`, `QList`, `QStringList`, `QByteArray` [134]. Format strings are also not used in the project. Source code revision has shown that the project does not contain integer overflow errors, as well as improper pointer usage.

The category related to security mechanisms contains vulnerabilities related to a full or particular lack of security mechanisms in a software (access control, authorization, authentification, protection of transferred data, and others) [78]. According to the specification (see Section 4), the mechanism are not yet implemented in PeerHood. But their implementation is planned and it is related to one of the further development directions of the project. Source code revision has shown that data are transferred in unencrypted form between mobile devices in the network environment. Thus it can be said that confidentiality and integrity of the data as PeerHood security

components can be violated.

The category related to errors contains the following vulnerabilities: double free of memory blocks, memory leakage, dereferencing of a NULL pointer [78]. The current PeerHood implementation is delivered with a set of tests, including those that are intended for memory leakage detection. They pass successfully, thus it can be said that PeerHood does not contain memory leaks. This is proved by dynamic analysis results (see Section 5.4.3). PeerHood source code revision has shown that errors related to NULL pointer dereferencing and double free of memory block are not found in the project.

The category related to software environment contains the following vulnerabilities: unsafe assignment of access rights to files used by a software, storing confidential information in unencrypted form [78]. The PeerHood vulnerability related to access rights set to it's configuration file has been considered in Section 5.4.4. But there is another one that is related to the fact that the configuration file stores information about name of the mobile device and it's identifier. These values are used to differentiate devices in a network environment. Unauthorized modification of these parameters can lead to incorrect device identification. In this case an attacker can pose as a normal user and perform appropriate malicious operations. The consequence of this fact is a possible violation of confidentiality related to transferred data.

The category related to encapsulation contains the following vulnerabilities: information disclosure, encapsulation violation related to classes of an software [78]. In the current PeerHood implementation, the daemon utilizes the hostname and the device identifier and transfers them to another devices in a network environment. Such behaviour from the security point of view can be considered as a fact of information disclosure because it helps an attacker to pose himself as another user in the network. The consequences of this fact has been already considered above in this subsection. Search of API classes encapsulation violation is performed further in this subsection.

The category related to improper API use deals with API contract elements violation [78]. Such operation can promote to incorrect functioning of an application, including unsafe behaviour. PeerHood API security analysis enables to estimate fault-tolerance of it's components. Additionally it promotes to a search of vulnerabilities related to information disclosure.

As it was noted above, PeerHood utilizes the more safe alternatives to C/C++ arrays: `QList`, `QString` classes and others [134]. According to Qt documentation [134],

these classes are not immutable, so their state can be changed at runtime. Including in a unauthorized manner, it may lead to some security problems. In Object-Oriented Programming (OOP) languages it is expressed as violation of class encapsulation. The violation is related to modification of a field of a class from outside of the class, not in a method or a constructor of the class. From the security point of view, such vulnerability can lead to confidentiality violation, if a field is storing some confidential information. Also this can lead to violation of integrity, when a value of the field is modified in a unauthorized manner. Finally, this can lead to violation of availability, when a block of memory holding the field is freed in an unauthorized manner.

PeerHood API contains a number of vulnerabilities related to it's class encapsulation violation. They are listed in Table 2. These vulnerabilities affect all the functional PeerHood characteristics except availability of simultenous several client serving and a capabiltiy of user control (see Section 5.4.1).

Class	**Methods/constructors**

`Service`	`const QString& name()`
	`const QStringList& attributes()`
	`const QStringList& attributes()`

`AbstractConnection`	`const QString remoteAddress()`
`AbstractCreator`	`const QString& connectionBase()`
`AbstractMonitor`	`const QString& connectionBase()` `const QString& address()`
`AbstractPinger`	`const QString& connectionBase()` `const QString& address()`

`Device`	`const QString& name()`
	`const QString& prototype()`
	`const QString& address()`
	`QList<Service>& serviceList()`

`AbstractAdverter`	`const QString& connectionBase()`

Table 2. PeerHood API classes with violated encapsulation

To provide fault-tolerance assurance for an application, it's components should perform input data checking. In case of API function arguments act as such input data. Lack of input data checking by a component may lead to violation of it's functioning. From the security point of view, this may cause availability violation when input data has unexpected or incorrect value and they cause the application

failure.

Concerning PeerHood such vulnerabilities affect it's functioning. They affect it's functional characteristics related to availability and capability (see Section 5.4.1). The methods related in PeerHood API classes are listed in Table 3. They do not verify values of their arguments. Source code revision has shown that PeerHood do not check input data, including transferred data. In particularly, deserialization of `Service` and `Device` classes is unsafe.

PeerHood user control consists in a capability of changing it's state, for example, transition of low power consumption state. From the security point of view, API components responsible for this functionality are implemented correctly. The same can be said about simultaneous serving of several clients. The feature is implemented correctly by means of synchronization primitives (`QMutex` [134]). Revision of the appropriate blocks of PeerHood source code responsible for utilization of resources has not detected any race condition.

5.5 Analysis of obtained results

PeerHood security analysis showed that the project contains security vulnerabilities. These are as follows:

- non-safe access rights assigned to the configuration file
- lack of security mechanisms
- data is transferred between devices in unencrypted form
- information about hostname and identifier of a device is transmitted during communication between devices
- violation of PeerHood API classes encapsulation
- lack of input data checking in all entry points of the project

The fact that PeerHood contains the security vulnerabilities listed above influences on it's functioning. It may lead to violation of the following PeerHood functional characteristics:

- data transferability between devices
- confidentiality and integrity of transferred data

Class	Methods/constructors
Service	void setPort(unsigned int) Service(const QString& name, const QStringList& attributes, unsigned int port, quint64 pid = 0, QObject parent=0)
AbstractConnection	AbstractConnection(const QString& connectionBase, const QHostAddress& address = QHostAddress::Any, QObject parent = 0) bool listen(quint16 port)
AbstractCreator	AbstractMonitor createMonitor(const QString& address) AbstractPinger createPinger(const QString& address)
AbstractMonitor	AbstractMonitor(const QString& address, const QString& connectionBase)
AbstractPinger	AbstractPinger(const QString& address, const QString& connectionBase)
Device	Device(QDataStream& stream, QObject parent = 0)
PeerHood	int registerService(const QString& name, const QStringList& attributes = QStringList(), unsigned int port = 0) bool unregisterService(const QString& name, unsigned int port = 0) void connectNotify(const char signal) void disconnectNotify(const char signal)

Table 3. PeerHood API classes that do not check input data

- availability, confidentiality, and integrity of up-to-date information about devices in the network environment
- possibility of user data exchange and exchange of information about available devices in the network environment
- availability, confidentiality, and integrity of up-to-date information about available services in the network environment
- possibility of exchange between devices of information about available services in the network environment
- possibility of user data transfer between devices
- confidentiality and integrity of user data transferred between devices

Violation of the functional characteristics listed above leads to violation of all components of PeerHood security: integrity, availability, and confidentiality. This in turn leads to apperance of a threat to violation of functional requirements to PeerHood.

Thus, violation of functional requirements to PeerHood leads to appearance of threats to non-conformances of the project goals. And this in turn negatively influences on it's fault-tolerance. Thus one can concludes that PeerHood is non-secure software.

6 CONCLUSIONS AND FUTURE WORK

Not only answers become obsolete, but even questions.
Ernest Hemingway

This work covers the security problem of user data utilized during carrying out personal communications in a mobile peer-to-peer environment. One of the aspects of it's security assurance is security of software operating on a mobile device. The PeerHood software was examined in this work. It is indended to carry out personal communications regardless of underlying network technology. The goal of the work was to assess the security of this project.

The first step to reach the goal was to study the software security problem in mobile peer-to-peer environment. The concept, factors, and threats of software security were examined. Known software security defects and vulnerabilities were considered in detail. Finally, security threats both typical for mobile and peer-to-peer environments were studied. The results of this study led to realize that the software security problem consists of and by which factors it is influenced in the context of a mobile peer-to-peer environment. Software security problem has been studied enough to understand it's importance and urgency.

The second step was to perform the following activities. The idea of secure software was studied. It helped to understand which characteristics software should have to ensure security of user data and to be a fault-tolerant software. Software security testing principles and methods have been studied enough to utilize them practically right.

The next step to reach the goal of the research was to study the PeerHood concept. The goals, requirements, and architecture of the software were examined. The step was one of the practical parts of the research and was required to further PeerHood security analysis. PeerHood has been studied rather comprehensively to realize it's concept and operation principles.

Finally, the last step provided the main practical part of the research. The PeerHood analysis was made by performing of risk-based and property-based software security methods, dynamic and static analysis of the project, source code revision as well. The results of the analysis showed that PeerHood has security vulnerabilities affecting the integrity and the confidentiality of data transferred between devices and the availability proper operating of the project. Therefore PeerHood can not be treated

as a secure and fault-tolerant software. The performed analysis is rather thorough to answer the research question, but it can't be considered as exhaustive because of specific character of the software security domain.

In this way, it can be said that the chosen approach resulted in the achievement of the goal of the work. However, only well-known and most frequent software security vulnerabilities were searching in PeerHood during making the analysis. Therefore, it is necessary to make more thorough security analysis of the project to get more accurate results. But even in this case it is impossible to achieve 100-% result because of the following specific character of software security testing. A vulnerability can exist in a project but could be not found by the testing.

6.1 Further work

The results of the research are planned to be used in further development of PeerHood. One of the further development directions is related to implementation of a security mechanism. In this way, it can be said that the research is the first step towards this direction. Also, it can be considered as the first step towards taking into account of secrurity in further development of PeerHood.

In spite of practical benefits of integration of the gained results it should be taken into consideration of their insufficiency. This is because of existence of other aspects of information security assurance apart from software security assurance. Finally, software security assurance is accomplished not only by software security testing but also by other activities carried out on all stages of software development lifecycle.

Currently there is a tendency of permanent growth of a number of discovered software security vulnerabilities. Hence, software security assurance process becomes more sophisticated. At the same time the gradual growth of popularity of personal trusted device usage is noted. In this way, there is the urgent need for further researches in the area of software security assurance in mobile peer-to-peer environment.

REFERENCES

[1] *The Official Bluetooth Technology Web Site*. Online page, from: http://www.bluetooth.com/, referred February 20, 2012

[2] R. Schollmeier, *A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications*, Proceedings of the First International Conference on Peer-to-Peer Computing 2001, Washington, DC, USA, 2001

[3] J. Porras and P. Hiirsalmi and A. Valtaoja, *Peer-to-peer Communication Approach for a Mobile Environment*, Proceedings of the 37th Hawaii International Conference on System Sciences, Big Island, Hawaii, USA, 2004

[4] A. Hämäläinen and J. Porras, *Enhancing Mobile Peer-to-Peer Environment with Neighborhood Information*, Proceedings of the 3rd Workshop on Applications of Wireless Communications, Finland, 2005

[5] *iOS: Mobile Operating System*. Online page, from: http://www.applce.com/ios/, referred February 20, 2012

[6] *Google Android Platform*. Online page, from: http://www.android.com/, referred February 20, 2012

[7] *Microsoft Reimagines Windows, Presents Windows 8 Developer Preview*. Online page, from: http://www.microsoft.com/presspass/press/2011/sep11/09-13FutureofComputingPR.mspx, referred February 20, 2012

[8] *Ubuntu Unity Project*. Online page, from: http://unity.ubuntu.com/, referred February 20, 2012

[9] J. Porras and P. Jäppinen and P. Hiirsalmi and A. Hämäläinen and S. Saalasti and R. Koponen and S. Keski-Jaskari, *Personal Trusted Device in Personal Communications*, 1st IEEE International Symposium on Wireless Communication Systems, Mauritius, 2004

[10] I. Mavridis and G. Pangalos, *Security Issues in a Mobile Computing Paradigm*, 1997. Online document, from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.573&rep=rep1&type=pdf, retrieved February 20, 2012

[11] K. M. Goertzel and T. Winograd, *Enhancing the Development Life Cycle to Produce Secure Software*, Technology Analysis Center (IATAC), USA, October 2008

[12] *PeerHood - Peer-to-Peer Neighborhood*, 2006. Online document, from: http://www2.it.lut.fi/project/ptd/peerhood.html, retrieved February 20, 2012

[13] K. M. Goertzel and T. Winograd, *Software Security Assurance. State-of-the-Art Report*, Technology Analysis Center (IATAC), Data and Analysis Center for Software (DACS), USA, July 2007

[14] K. Kolehmainen, *A Communication Middleware Quality Enhancement with Qt Framework*. Master's thesis, Lappeenranta University of Technology, Finland, 2010

[15] *The Source for Linux Information*. Online page, from: https://www.linux.com/, referred February 20, 2012

[16] International Organization for Standardization (ISO), *C++ Programming Language Standard (ISO/IEC 14882)*, 2003. Available at: http://www.open-std.org/jtc1/sc22/WG21/, retrieved February 20, 2012

[17] *Qt: Cross-platform Application and UI Framework*. Online page, from: http://qt.nokia.com/, referred February 20, 2012

[18] R. Stratonovich, *Theory of Information*, Soviet Radio, USSR, 1975, 424 pages

[19] *The Cambridge Dictionary Of Philosophy*, 2nd edition, Cambridge University Press, United Kingdom, 1999, 1032 pages, ISBN: 0-511-07417-4

[20] International Organization for Standardization (ISO), *ISO/IEC FDIS 17799. Information Technology — Security Techniques — Code of Practice for Information Security Management*, April 2005. Available at: http://www.iso.org/iso/catalogue_detail?csnumber=39612, retrieved February 20, 2012

[21] Legal Information Institute, *Title 44 of the United States Code*, 2010. Online document, from: http://www.law.cornell.edu/uscode/44/3542.html, retrieved February 20, 2012

[22] National Institute of Standards and Technology, USA, *Standards for Security Categorization of Federal Information and Information Systems (FIPS PUB 199)*, February 2004. Available at:

http://csrc.nist.gov/publications/fips/fips199/FIPS-PUB-199-final.pdf, retrieved February 20, 2012

[23] V. Domarev, *IT Security. The System Approach*, ITD DIS, Russia, 2004, 992 pages, ISBN: 966-7992-36-5

[24] Internet Engineering Task Force, *Internet Security Glossary, Version 2 (RFC 4949)*, August 2007. Available at: http://tools.ietf.org/html/rfc4949, retrieved February 20, 2012

[25] W. Stallings and L. Brown, *Computer Security. Principles and Practice*, Prentice Hall, USA, 2008, 880 pages, ISBN: 0-13-600424-5

[26] G. McGraw, *Software Security. Building Security In*, Addison Wesley Professional, USA, 2006, 448 pages, ISBN: 0-321-35670-5

[27] G. McGrow, *Software Security*, IEEE Security and Privacy, Vol. 2, No. 2, USA, March 2004

[28] G. Hoglund and G. McGraw, *Exploiting Software. How to Break Code*, Peter Collin Publishing, United Kingdom, 2004, 512 pages, ISBN: 0-201-78695-8

[29] F. P. Brooks, *The Mythical Man-Month. Essays on Software Engineering*, 20th edition, Addison Wesley Longman, USA, 1995, 322 pages, ISBN: 0201835959

[30] S. McConnell, *Code Complete. A Practical Handbook of Software Construction*, 2nd edition, Microsoft Press, USA, 2004, 960 pages, ISBN: 978073561967

[31] S. McConnell, *Less is more*, Software Development, Vol. 5, No. 5, USA, October 1997, pages 28–34

[32] National Institute of Standards and Technology, USA, *Glossary of Key Information Security Terms*, February 2011. Available at: http://csrc.nist.gov/publications/nistir/ir7298-rev1/nistir-7298-revision1.pdf, retrieved February 20, 2012

[33] A. K. Talukder and R. Yavagal, *Mobile Computing. Technology, Applications, and Service Creation*, McGraw-Hill, USA, 2007, 672 pages, ISBN: 0071477330

[34] M. G. Graff and K. R. van Wyk, *Secure Coding. Principles and Practices*, O'Reilly, USA, 2003, 224 pages, ISBN: 0-596-00242-4

[35] H. Langweg and E. Snekkenes, *A Classification of Malicious Software Attacks*, Proceedings of the 2004 IEEE International Performance, Computing, and Communications Conference, USA, April 2004

[36] J. Erickson, *Hacking. The Art of Exploitation*, No Starch Press, USA, 2003, 240 pages, ISBN: 1-59327-007-0

[37] C. Anley and J. Heasman and F. F. Linder and G. Richarte , *The Shellcoder's Handbook, Second Edition. Discovering and Exploiting Security Holes*, Addison Wesley Professional, USA, 2007, 745 pages, ISBN: 978-0-470-08023-8

[38] S. Heelan, *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*. Master's thesis, University of Oxford, United Kingdom, 2009

[39] C. Landwehr and A. Bull and J. McDermott and W. Choi, *A Taxonomy of Computer Program Security Flaws*, ACM Computing Surveys, Vol. 26, No. 26, USA, September 1994, pages 211–254

[40] O. Alhazmi, *Assessing Vulnerabilities in Software Systems: A Quantitative Approach*. Doctoral thesis, Colorado State University, USA, 2006

[41] M. Dowd and J. McDonald and J. Schuh, *The Art of Software Security Assessment. Identifying and Preventing Software Vulnerabilities*, Addison Wesley Professional, USA, 2006, 1200 pages, ISBN: 0-321-44442-6

[42] *National Vulnerability Database*. Online page, from: http://nvd.nist.gov/, referred February 20, 2012

[43] *CWE - Common Weakness Enumeration*. Online page, from: http://cwe.mitre.org/, referred February 20, 2012

[44] *BugTraq*. Online page, from: http://www.securityfocus.com/, referred February 20, 2012

[45] R. S. Seacord, *Secure Coding in C and C++*, Addison Wesley Professional, USA, 2005, 368 pages, ISBN: 0-321-33572-4

[46] International Organization for Standardization (ISO), *C Programming Language Standard (ISO/IEC 9899)*, 1999. Available at: http://www.open-std.org/jtc1/sc22/wg14/, retrieved February 20, 2012

[47] *TIOBE Programming Community Index for December 2011*, 2011. Online document, from: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, retrieved February 20, 2012

[48] *PHP: Hypertext Preprocessor*. Online page, from: http://www.php.net/, referred February 20, 2012

[49] *Python Programming Language*. Online page, from: http://www.python.org/, referred February 20, 2012

[50] *The Java Platform*. Online page, from: http://www.java.com/, referred February 20, 2012

[51] Carnegie Mellon University's Computer Emergency Response Team (CERT), *CERT Statistics (Historical)*, 2008. Online document, from: http://www.cert.org/stats/, retrieved February 20, 2012

[52] B. Chess, *Secure Programming with Static Analysis*, Pearson Education, USA, 2007, 619 pages, ISBN: 0-321-42477-8

[53] Y. Younan and W. Joosen and F. Piessens, *Code Injection in C and C++. A Survey of Vulnerabilities and Countermeasures*, Katholieke Universiteit Leuven, Belgium, 2004

[54] H. H. Thompson and S. G. Chase, *The Software Vulnerability Guide*, Charles River Media, USA, 2005, 369 pages, ISBN: 1-58450-358-0

[55] Carnegie Mellon University's Computer Emergency Response Team (CERT), USA, *CERT C Programming Language Secure Coding Standard (Document No. N1255)*, September 2007. Available at: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1255.pdf, retrieved February 20, 2012

[56] M. Howard and D. LeBlanc and J. Vega, *19 Deadly Sins of Software Security. Programming Flaws and How to Fix Them*, McGraw-Hill/Osborne, USA, 2005, 304 pages, ISBN: 0072260858

[57] K. Lhee and S. Chapin, *Buffer Overflow and Format String Overflow Vulnerabilities*, Software – Practice and Experience, Vol. 33, No. 33, USA, April 2003

[58] T. Gallagher and B. Jeffries and L. Launder, *Hunting Security Bugs*, Microsoft Press, USA, 2006, 586 pages, ISBN: 073562187

[59] *RATS: Rough Auditing Tool for Security*. Online page, from: https://www.fortify.com/ssa-elements/threat-intelligence/rats.html, referred February 20, 2012

[60] *Flawfinder Home Page*. Online page, from: http://www.dwheeler.com/flawfinder/, referred February 20, 2012

[61] *PScan: A limited problem scanner for C source files*. Online page, from: http://deployingradius.com/pscan/, referred February 20, 2012

[62] *FormatGuard*. Online page, from: http://lwn.net/2001/0531/a/formatguard.php3, referred February 20, 2012

[63] *libformat*. Online page, from: http://www.securityfocus.com/tools/1818, referred February 20, 2012

[64] *Libsafe: Free Software Foundation*. Online page, from: http://directory.fsf.org/wiki/Libsafe, referred February 20, 2012

[65] F. C. Eigler, *Mudflap. Pointer Use Checking for C/C++*, GCC Developers Summit, Canada, May 2003

[66] Apple Inc., USA, *Secure Coding Guide*, February 2010. Available at: http://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/SecureCodingGuide.pdf, retrieved February 20, 2012

[67] U. Drepper, *Defensive Programming for Red Hat Enterprise Linux (and What To Do If Something Goes Wrong)*, IEEE Security and Privacy, USA, 2009

[68] R. Seacord and J. Rafail, *The CERT C Secure Coding Standard*, 2008. Online document, from: http://www.ioc.ornl.gov/csiirw/07/abstracts/Rafail-Abstract.pdf, retrieved February 20, 2012

[69] D. Wheeler, *Secure Programming for Linux and Unix HOWTO*, March 2003. Available at: http://www.dwheeler.com/secure-programs/, retrieved February 20, 2012

[70] M. Messier and J. Viega, *Secure Programming Cookbook for C and C++*, O'Reilly, USA, 2003, 784 pages, ISBN: 0-596-00394-3

[71] B. Eckel, *Thinking in Java*, 4th edition, Prentice Hall, USA, 2006, 1150 pages, ISBN: 0131872486

[72] R. H. Netzer and B. P. Miller, *What are Race Conditions? Some Issues and Formalizations*, ACM Letters on Programming Languages and Systems, Vol. 1, No. 1, USA, March 1992

[73] *ITS4: Software Security Tool*. Online page, from: http://www.cigital.com/its4/, referred February 20, 2012

[74] *PVS-Studio: Static Code Analyzer for C/C++/C++11*. Online page, from: http://www.viva64.com/en/pvs-studio/, referred February 20, 2012

[75] *Helgrind: a thread error detector*. Online page, from: http://valgrind.org/docs/manual/hg-manual.html, referred February 20, 2012

[76] *Intel Inspector XE*. Online page, from: http://software.intel.com/en-us/articles/intel-inspector-xe/, referred February 20, 2012

[77] Open Web Application Security Project (OWASP) Foundation, *The Ten Most Critical Web Application Security Vulnerabilities*, 2010. Online document, from: http://www.owasp.org/index.php/Top_10_2010, retrieved February 20, 2012

[78] K. Tsipenyuk and B. Chess and G. McGraw, *Seven Pernicious Kingdoms. A Taxonomy of Software Security Errors*, IEEE Security and Privacy, Vol. 3, No. 3, USA, November 2005, pages 81–84

[79] Fortify Software, *Fortify Taxonomy: Software Security Errors*, 2009. Online document, from: https://www.fortify.com/vulncat/en/vulncat/index.html, retrieved February 20, 2012

[80] C. Landwehr and A. Bull and J. McDermott and W. Choi, *A Taxonomy of Computer Program Security Flaws, with Examples*, Naval Research Laboratory, USA, November 1993

[81] The MITRE Corporation, *Top 25 Most Dangerous Software Errors*, 2011. Online document, from: http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf, retrieved February 20, 2012

[82] B. R. Karki, *Social Networking on Mobile Environment on top of PeerHood*. Master's thesis, Lappeenranta University of Technology, Finland, 2008

[83] N. Davies and G. S. Blair and K. Cheverst and A. Friday, *Supporting Adaptive Services in a Heterogeneous Mobile Environment*, WMCSA '94

Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications, Washington, DC, USA, 1994

[84] N. Asokan, *Security Issues in Mobile Computing*, 1995. Online document, from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.9758&rep=rep1&type=pdf, retrieved February 20, 2012

[85] W. Li and A. Joshi, *Security Issues in Mobile Ad Hoc Networks - A Survey*, The 17 White House Papers Graduate Research In Informatics, Sussex, United Kingdom, June 2004

[86] G. Bian and K. Nakayama and Y. Kobayushi and M. Maekawa, *Java Mobile Code Security by Bytecode Analysis*, ECTI Transactions on Computer and Information Theory, Vol. 1, No. 1, USA, May 2005

[87] A. D. Rubin and D. E. Geer, *Mobile Code Security*, IEEE Internet Computing, Vol. 2, No. 2, USA, November 1998, pages 30–34

[88] Open Web Application Security Project (OWASP) Foundation, *Top Ten Mobile Risks*, 2011. Online document, from: http://www.owasp.org/index.php/OWASP_Mobile_Security_Project, retrieved February 20, 2012

[89] Internet Engineering Task Force, *Security Issues and Solutions in Peer-to-Peer Systems for Realtime Communications (RFC 5765)*, February 2010. Available at: http://tools.ietf.org/html/rfc5765, retrieved February 20, 2012

[90] D. S. Wallach, *A Survey of Peer-to-Peer Security Issues*, Proceedings of the 2002 Mext-NSF-JSPS International Conference on Software Security. Theories and Systems, USA, 2002

[91] M. P. Barcellos, *Security Issues and Perspectives in P2P Systems: from Gnutella to BitTorrent*, 53nd Meeting of the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance, Natal, Brazil, 2008

[92] E. Damiani and D. Di Vimercati and S. Paraboshi and P. Samarati and F. Violante, *A Reputation-Based Approach for Choosing Reliable Resources in Peer-to-Peer Networks*, Proceedings of the 9th ACM Conference on Computer and Communications Security, USA, 2002

[93] C. C. Michael and W. Radosevich, *Risk-Based and Functional Security Testing*, 2005. Online document, from: https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/testing/255-BSI.html, retrieved February 20, 2012

[94] G. McGrow and B. Potter, *Software Security Testing*, IEEE Security and Privacy, Vol. 2, No. 2, USA, September 2004

[95] M. Sutton and A. Greene and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, Addison Wesley Professional, USA, 2007, 576 pages, ISBN: 0321446119

[96] *Splint: Static Code Analyser*. Online page, from: http://www.splint.org/, referred February 20, 2012

[97] *Cppcheck: A Tool for Static C/C++ Code Analysis*. Online page, from: http://cppcheck.sourceforge.net/, referred February 20, 2012

[98] G. Fink and M. Bishop, *Property-Based Testing; A New Approach to Testing for Assurance*, ACM SIGSOFT Software Engineering Notes, Vol. 22, USA, 1997, pages 74–80

[99] E. Eilam, *Reversing: Secrets of Reverse Engineering*, Wiley, USA, 2005, 624 pages, ISBN: 0764574817

[100] *IDA: Multi-processor Disassembler and Debugger*. Online page, from: http://www.hex-rays.com/products/ida/index.shtml, referred February 20, 2012

[101] *MSIL Disassembler (ildasm.exe)*. Online page, from: http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.71).aspx, referred February 20, 2012

[102] *Microsoft .NET Framework*. Online page, from: http://www.microsoft.com/net, referred February 20, 2012

[103] *OllyDbg: 32-bit Assembler Level Analysing Debugger*. Online page, from: http://www.ollydbg.de/, referred February 20, 2012

[104] *WinDbg: Multipurposed Debugger for Microsoft Windows*. Online page, from: http://msdn.microsoft.com/en-us/windows/hardware/gg463009, referred February 20, 2012

[105] *.NET Reflector*. Online page, from: http://www.reflector.net, referred February 20, 2012

[106] *Microsoft Windows*. Online page, from: http://windows.microsoft.com/, referred February 20, 2012

[107] *Windows SysInternals*. Online page, from: http://technet.microsoft.com/en-us/sysinternals/, referred February 20, 2012

[108] B. Chess and G. Snyder and T. R. Hein and B. Whaley, *UNIX and Linux System Administration Handbook*, 4th edition, Pearson Education, United Kingdom, 2010, 619 pages, ISBN: 0-13-148005-7

[109] J. A. Whittaker and H. H. Thompson, *Black Box Debugging*, Queue, Vol. 1, No. 1, USA, December 2003, pages 68–74

[110] *Nmap: Free Security Scanner for Network Exploration and Security Audits*. Online page, from: http://www.nmap.org/, referred February 20, 2012

[111] *Nessus: Web Application Security Scanner*. Online page, from: http://www.nessus.org/, referred February 20, 2012

[112] B. Arkin and S. Stender and G. McGraw, *Software Penetration Testing*, IEEE Security and Privacy, Vol. 3, No. 3, USA, January 2005, pages 84–87

[113] H. H. Thompson, *Application Penetration Testing*, IEEE Security and Privacy, Vol. 3, No. 3, USA, January 2005, pages 66–69

[114] J. Newsome and D. Song, *Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software*, 2005. Online document, from: http://www.ece.cmu.edu/~dawnsong/papers/taintcheck.pdf, retrieved February 20, 2012

[115] *StackGuard Tool*. Online page, from: http://www.securityfocus.com/tools/355, referred February 20, 2012

[116] *Valgrind*. Online page, from: http://valgrind.org/, referred February 20, 2012

[117] H. Madeira and D. Costa and M. Vieira, *On the Emulation of Software Faults by Software Fault Injection*, In Proceedings of the International Conference on Dependable Systems and Networks, New York, NY , USA, 2000

[118] M. Hsueh and T. K. Tsai and R. K. Iyer, *Fault Injection Techniques and Tools*, Computer, Vol. 30, No. 30, USA, April 1997, pages 75–82

[119] *Nokia Corporation*. Online page, from: http://www.nokia.com/, referred February 20, 2012

[120] *Usability of Music for the Social Inclusion of Children (UMSIC)*. Online page, from: http://www.umsic.org/, referred February 20, 2012

[121] J. Laakkonen, *PeerHood as UMSIC middleware module*. Master's thesis, Lappeenranta University of Technology, Finland, 2009

[122] *GNU General Public License, version 2*. Online page, from: http://www.gnu.org/licenses/gpl-2.0.html, referred February 20, 2012

[123] *PeerHood Source Code Repository*, 2011. Online document, from: http://gitorious.org/peerhood, retrieved February 20, 2012

[124] *PeerHood Specification*, 2009. Online document, from: http://www2.it.lut.fi/wiki/doku.php/peerhood/specification, retrieved February 20, 2012

[125] P. Krutchen, *The Rational Unified Process. An Introduction*, 2nd edition, Addison Wesley, USA, 2000, 320 pages, ISBN: 0-201-70710-1

[126] A. Hämäläinen and P. Hiirsalmi and J. Porras, *Comparison of Linux and Symbian Based Implementations of Mobile Peer-to-Peer Environment*, The 13th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2005), Split, Marina Frapa, Croatia, 2005

[127] A. Stellman and J. Greene, *Applied Software Project Management*, O'Reilly Media, USA, 2005, 336 pages, ISBN: 0596009488

[128] Institute of Electrical and Electronics Engineers, USA, *Portable Operating System Interface for Unix (POSIX.1-2008)*, 2008. Available at: http://standards.ieee.org/findstds/standard/1003.1-2008.html, retrieved February 20, 2012

[129] R. W. Stevens and B. Fenner and A. M. Rudoff, *UNIX Network Programming. The Sockets Networking API. Vol. 1*, 3rd edition, Addison Wesley, USA, 2004, 1024 pages, ISBN: 0-13-141155-1

[130] *IEEE 802.11, The Working Group Setting the Standards for Wireless LANs*. Online page, from: http://www.ieee802.org/11/, referred February 20, 2012

[131] *ETSI GPRS*. Online page, from:
http://www.etsi.org/WebSite/technologies/gprs.aspx, referred February 20, 2012

[132] *GCC: Warning Options*. Online page, from:
http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html, referred February 20, 2012

[133] National Institute of Standards and Technology, USA, *Source Code Security Analysis Tool Functional Specification Version 1.0*, May 2007. Available at:
http://samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268.pdf,
retrieved February 20, 2012

[134] *Qt Online Reference Documentation*. Online page, from:
http://doc.qt.nokia.com/, referred February 20, 2012

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#define UCHAR_MAX_LENGTH 3 /* UCHAR_MAX == 255 */
int is_prime(unsigned short number);

int main(int argc, char** argv) {
    unsigned int number_buf_size = (UCHAR_MAX_LENGTH + 1);
    char* number_buf =
        (char *) malloc(sizeof(char) * number_buf_size);
    printf("Enter a number (1-%d)", UCHAR_MAX);
    gets(number_buf);

    unsigned char number =
        (unsigned char) atoi(number_buf);
    char* result = is_prime(number)
        ? "is a prime" : "is not a prime";
    printf(number_buf);
    printf(" %s number\n", result);

    free(number_buf);
    return EXIT_SUCCESS;
}

int is_prime(unsigned int number) {
    int result = 1;
    int i;

    for(i = 2; i <= number / 2; i++) {
        if((number % i) == 0) {
            result = 0;
        }
    }

    return result;
}
```

peerhoodglobal.h

```
enum EConnectionState {
    OfflineState = 0,
    OnlineState,
    PowerSavingState,
    NormalPowerState,
    PassiveState
};
```

abstractconnection.h

```
class PH_COMMONSHARED_EXPORT AbstractConnection : public QObject
{
    Q_OBJECT
public:
  virtual ~AbstractConnection(){}
  virtual bool init() = 0;
  virtual const QString& name() = 0;
  virtual const QString& connectionBase() = 0;
  virtual EConnectionState state() = 0;

signals:
  void stateChanged(EConnectionState state, const QString&
     connectionBase);

public slots:
  virtual void changeState(EConnectionState state) = 0;
};
```

abstractcreator.h

```
class PH_COMMONSHARED_EXPORT AbstractCreator : public QObject
{
    Q_OBJECT
public:
    virtual AbstractConnection* createConnection() = 0;
    virtual AbstractAdverter* createAdverter() = 0;
    virtual AbstractPinger* createPinger(const QString& address) = 0;
    virtual AbstractMonitor* createMonitor(const QString& address) = 0;
    virtual const QString& connectionBase() = 0;
};
```

(continue)

device.h

```
PH_COMMONSHARED_EXPORT QDataStream& operator<<(QDataStream&, Device&);
PH_COMMONSHARED_EXPORT QDataStream& operator<<(QDataStream&, Device*);
PH_COMMONSHARED_EXPORT QDebug operator<<(QDebug, Device&);
PH_COMMONSHARED_EXPORT QDebug operator<<(QDebug, Device*);
PH_COMMONSHARED_EXPORT QDataStream& operator>>(QDataStream&, Device*);

class PH_COMMONSHARED_EXPORT Device : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString name READ name)
    Q_PROPERTY(int checksum READ checksum)
    Q_PROPERTY(QString prototype READ prototype)
    Q_PROPERTY(QString address READ address)
    Q_PROPERTY(bool hasPeerHood READ hasPeerHood)

protected:
    explicit Device(QObject *parent = 0);

public:
    explicit Device(QDataStream& stream, QObject *parent = 0);
    virtual ~Device();
    virtual const QString& name();
    virtual unsigned int checksum();
    virtual const QString& prototype();
    virtual const QString& address();
    virtual bool hasPeerHood();
    virtual const QList<Service*>& serviceList();

    Q_INVOKABLE virtual bool hasService(const QString& service);

protected:
    virtual void lockServiceList();
    virtual void unLockServiceList();

protected:
    QString         m_name;
    unsigned int    m_checksum;
    QString         m_prototype;
    QString         m_address;
    bool            m_hasPeerHood;
    QList<Service*> m_services;

private:
    PH_COMMONSHARED_EXPORT friend QDataStream& operator<<(QDataStream&,
```

(continue)

```
        Device&);
    PH_COMMONSHARED_EXPORT friend QDataStream& operator<<(QDataStream&,
        Device*);
    PH_COMMONSHARED_EXPORT friend QDebug operator<<(QDebug, Device&);
    PH_COMMONSHARED_EXPORT friend QDebug operator<<(QDebug, Device*);
    PH_COMMONSHARED_EXPORT friend QDataStream& operator>>(QDataStream&,
        Device*);
};
```

service.h

```
PH_COMMONSHARED_EXPORT QDataStream& operator<<(QDataStream&, Service&);
PH_COMMONSHARED_EXPORT QDataStream& operator<<(QDataStream&, Service*);
PH_COMMONSHARED_EXPORT QDataStream& operator>>(QDataStream&, Service&);
PH_COMMONSHARED_EXPORT QDataStream& operator>>(QDataStream&, Service*);
PH_COMMONSHARED_EXPORT QDebug operator<<(QDebug, Service&);
PH_COMMONSHARED_EXPORT QDebug operator<<(QDebug, Service*);

class PH_COMMONSHARED_EXPORT Service : public QObject
{
    Q_OBJECT
protected:
    static const QChar m_separator;

public:
    explicit Service(QObject *parent = 0);
    explicit Service(const Service* original);
    explicit Service(const QString& name, const QStringList& attributes
       ,
      unsigned int port, quint64 pid = 0, QObject* parent=0);
    virtual ~Service();

public:
    virtual const QString& name();
    virtual unsigned int port();
    virtual bool isAttribute(const QString& attribute);
    virtual const QStringList& attributes();
    virtual quint64 pid();
    virtual void setPort(unsigned int);

protected:
    QString m_name;
    quint16 m_port;
    QStringList m_attributes;
    quint64 m_pid;
```

(continue)

```
private:
    PH_COMMONSHARED_EXPORT friend QDataStream& operator<<(QDataStream&,
        Service&);
    PH_COMMONSHARED_EXPORT friend QDataStream& operator<<(QDataStream&,
        Service*);
    PH_COMMONSHARED_EXPORT friend QDataStream& operator>>(QDataStream&,
        Service&);
    PH_COMMONSHARED_EXPORT friend QDataStream& operator>>(QDataStream&,
        Service*);
    PH_COMMONSHARED_EXPORT friend QDebug operator<<(QDebug, Service&);
    PH_COMMONSHARED_EXPORT friend QDebug operator<<(QDebug, Service*);

};
```

abstractadverter.h

```
class PH_COMMONSHARED_EXPORT AbstractAdverter : public QObject
{
    Q_OBJECT
public:
    virtual ~AbstractAdverter(){};
    virtual bool start() = 0;
    virtual void stop() = 0;
    virtual bool isActive() = 0;
    virtual const QString& connectionBase() = 0;
};
```

abstractpinger.h

```
class PH_COMMONSHARED_EXPORT AbstractPinger : public QObject
{
    Q_OBJECT
public:
    explicit AbstractPinger(const QString& address, const QString&
       connectionBase);
    virtual ~AbstractPinger();
    virtual const QString& connectionBase();
    virtual const QString& address();
    virtual bool isInRange();
    virtual bool ping() = 0;

protected:
    QString m_connectionBase;
    QString m_address;
    bool m_isInRange;
};
```

(continue)

peerhoodsettings.h

```
class PHLIBRARYSHARED_EXPORT PeerHoodSettings : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString deviceName READ deviceName)
    Q_PROPERTY(int deviceChecksum READ deviceChecksum)
    Q_PROPERTY(QString peerhoodVersion READ peerhoodVersion)

public:
    explicit PeerHoodSettings(QObject *parent = 0);
    virtual ~PeerHoodSettings();
    const QString deviceName();
    int deviceChecksum();
    const QString peerhoodVersion();

private:
    PeerHoodSettingsPrivate* d;
};
```

peerhood.h

```
enum EDeviceStatus
{
    DeviceLost = 0x00,
    DeviceFound,
    WeakLink,
    VeryWeakLink
};

class PHLIBRARYSHARED_EXPORT PeerHood : public QObject
{
    Q_OBJECT
public:
    static PeerHood* instance();
    virtual ~PeerHood();
    virtual bool init();
    virtual const QList<Device*> deviceList();
    virtual const QList<Device*> deviceList(const QString& service);
    virtual const QList<Service*> localServiceList();
    virtual int registerService(const QString& name,
      const QStringList& attributes = QStringList(),
      unsigned int port = 0);
    virtual bool unregisterService(const QString& name, unsigned int
       port = 0);
    virtual AbstractConnection* connectToService(Device* device, const
```

(continue)

```
        QString& servicename);
    virtual AbstractConnection* connectToService(Service* service);
    virtual bool hasPendingConnections(int servicePort);
    virtual AbstractConnection* nextPendingConnection(int servicePort);
    virtual bool monitorDevice(Device* device);
    virtual void unMonitorDevice(Device* device);
    virtual bool signalMonitorDevice(Device* device);
    virtual void signalUnMonitorDevice(Device* device);

signals:
    void newConnection(int servicePort, int connectionId);
    void deviceStatusChanged(PH::EDeviceStatus status, QString address)
        ;

protected:
    explicit PeerHood(QObject *parent = 0);
    void connectNotify(const char* signal);
    void disconnectNotify(const char* signal);

protected:
    PeerHoodPrivate* d;

private:
    Q_DISABLE_COPY(PeerHood)
};
```

(continue)

abstractmonitor.h

```
class PH_COMMONSHARED_EXPORT AbstractMonitor : public QObject
{
    Q_OBJECT
public:
    explicit AbstractMonitor(const QString& address, const QString&
       connectionBase);
    virtual ~AbstractMonitor();

public:
    virtual bool isInRange();
    virtual int signalStrength() = 0;
    virtual bool startMonitoring() = 0;
    virtual const QString& connectionBase();
    virtual const QString& address();

public slots:
    virtual void stopMonitoring() = 0;

signals:
    void signalStrengthChanged(const QString& address, const QString&
       base, int strength);

protected:
    bool m_isInRange;
    QString m_address;
    QString m_connectionBase;
};
```

```
$ cppcheck -I ./common/

Checking common/abstractconnection.cpp...
Checking common/abstractconnection.cpp: COMMON_DEBUG_LOGGING...
Checking common/abstractconnection.cpp: PEERHOOD_COMMON_LIBRARY...
1/40 files checked 2% done

...

Checking common/connectionmanager.cpp...
Checking common/connectionmanager.cpp: BEARER_FROM_QT_MOBILITY...
Checking common/connectionmanager.cpp: COMMON_DEBUG_LOGGING...
Checking common/connectionmanager.cpp: PEERHOOD_COMMON_LIBRARY...
[./common/connectionmanager.cpp:92]: (style) Redundant condition. It is
    safe to deallocate a NULL pointer
6/40 files checked 12% done

...

Checking plugins/wlanplugin/wlanpinger.cpp...
Checking plugins/wlanplugin/wlanpinger.cpp: PEERHOOD_COMMON_LIBRARY...
Checking plugins/wlanplugin/wlanpinger.cpp: Q_OS_LINUX...
[./plugins/wlanplugin/wlanpinger_unix.cpp:1]: (style) The function '
   setSocket' is never used
[./plugins/wlanplugin/wlanpinger_unix.cpp:90]: (style) The function '
   SocketGuard::getSocket' can be const
32/40 files checked 80% done

...

Checking settings/settings_p.cpp...
40/40 files checked 100% done
```

```
$ rats .

Analyzing ./library/serviceconnectionhandler.cpp
Analyzing ./library/serviceconnectionacceptor.cpp
Analyzing ./library/serviceconnector.cpp
Analyzing ./library/devicemonitorworker.cpp
Analyzing ./library/peerhoodsettings.cpp
Analyzing ./library/peerhood.cpp
Analyzing ./daemon/daemon.cpp
Analyzing ./daemon/main.cpp
Analyzing ./daemon/daemonserver.cpp
Analyzing ./daemon/daemonclientpeer.cpp
Analyzing ./plugins/wlanplugin/wlanpluginadverter.cpp
Analyzing ./plugins/wlanplugin/wlanservicepublisher.cpp
Analyzing ./plugins/wlanplugin/wlanpinger.cpp
Analyzing ./plugins/wlanplugin/wlanpinger_unix.cpp
Analyzing ./plugins/wlanplugin/wlaninquirer.cpp
Analyzing ./plugins/wlanplugin/wlanplugincreator.cpp
Analyzing ./plugins/wlanplugin/wlanmonitor.cpp
Analyzing ./plugins/wlanplugin/wlanplugin.cpp
Analyzing ./plugins/localhostplugin/localhostplugincreator.cpp
Analyzing ./plugins/localhostplugin/localhostplugin.cpp
Analyzing ./register/register.cpp
Analyzing ./settings/settings_p.cpp
Analyzing ./settings/settings.cpp
Analyzing ./common/serviceinforeaderwriter.cpp
Analyzing ./common/abstractreaderwriter.cpp
Analyzing ./common/connectionpluginloader.cpp
Analyzing ./common/abstractmonitor.cpp
Analyzing ./common/device.cpp
Analyzing ./common/factory.cpp
Analyzing ./common/deviceinforeaderwriter.cpp
Analyzing ./common/datamanager.cpp
Analyzing ./common/daemonclientservice.cpp
Analyzing ./common/abstractconnection.cpp
Analyzing ./common/abstractpinger.cpp
Analyzing ./common/localdevice.cpp
Analyzing ./common/daemonconnector.cpp
```

(continue)

```
Analyzing ./common/service.cpp
Analyzing ./common/abstractpluginloader.cpp
Analyzing ./common/daemonclient.cpp
Analyzing ./common/connectionmanager.cpp
./plugins/wlanplugin/wlanpinger_unix.cpp:142: High: fixed size local
   buffer
./plugins/wlanplugin/wlanpinger_unix.cpp:187: High: fixed size local
   buffer
Extra care should be taken to ensure that character arrays that are
   allocated
on the stack are used safely.  They are prime targets for buffer
   overflow
attacks.

./plugins/wlanplugin/wlanpinger_unix.cpp:148: High: gethostbyname
DNS results can easily be forged by an attacker (or
arbitrarily set to large values, etc), and should not be trusted.

./daemon/main.cpp:45: Medium: signal
When setting signal handlers, do not use the same function to handle
   multiple signals. There exists the possibility a race condition
   will result if 2 or more different signals are sent to the process
   at nearly the same time. Also, when writing signal handlers, it is
   best to do as little as possible in them. The best strategy is to
   use the signal handler to set a flag, that another part of the
   program tests and performs the appropriate action(s) when it is set
   .
See also: http://razor.bindview.com/publish/papers/signals.txt

Total lines analyzed: 5932
Total time 0.007901 seconds
750791 lines per second
```

```
$ valgrind ./ph2daemon

==1874== Memcheck, a memory error detector
==1874== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al
   .
==1874== Using Valgrind-3.6.1-Debian and LibVEX; rerun with -h for
   copyright info
==1874== Command: ./ph2daemon
==1874==
==1874==
==1874== HEAP SUMMARY:
==1874==     in use at exit: 446,530 bytes in 4,104 blocks
==1874==   total heap usage: 30,705 allocs, 26,601 frees, 2,088,694
   bytes allocated
==1874==
==1874== LEAK SUMMARY:
==1874==    definitely lost: 0 bytes in 0 blocks
==1874==    indirectly lost: 0 bytes in 0 blocks
==1874==      possibly lost: 199,071 bytes in 2,438 blocks
==1874==    still reachable: 247,459 bytes in 1,666 blocks
==1874==         suppressed: 0 bytes in 0 blocks
==1874== Rerun with --leak-check=full to see details of leaked memory
==1874==
==1874== For counts of detected and suppressed errors, rerun with: -v
==1874== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 106 from
   15)
```

```
$ valgrind --tool=helgrind ./ph2daemon

==1979== Helgrind, a thread error detector
==1979== Copyright (C) 2007-2010, and GNU GPL'd, by OpenWorks LLP et al
   .
==1979== Using Valgrind-3.6.1-Debian and LibVEX; rerun with -h for
   copyright info
==1979== Command: ./ph2daemon
==1979== Parent PID: 1978
==1979==
==1979== Thread #1 is the program's root thread
==1979==
==1979== Thread #2 was created
==1979==    at 0x46320B8: clone (clone.S:111)
==1979==
==1979== Possible data race during write of size 1 at 0x48a247c by
   thread #1
==1979==    at 0x4332DC9: QObjectPrivate::setThreadData_helper(
   QThreadData*, QThreadData*) (in /usr/lib/i386-linux-gnu/libQtCore.
   so.4.7.4)
==1979==    by 0x4332FE0: QObject::moveToThread(QThread*) (in /usr/lib/
   i386-linux-gnu/libQtCore.so.4.7.4)
==1979==    by 0x4103032: QNetworkConfigurationManagerPrivate::
   updateConfigurations() (in /usr/lib/i386-linux-gnu/libQtNetwork.so
   .4.7.4)
==1979==    by 0x410375C: QNetworkConfigurationManagerPrivate::
   initialize() (in /usr/lib/i386-linux-gnu/libQtNetwork.so.4.7.4)
==1979==    by 0x4579112: (below main) (libc-start.c:226)
==1979==  This conflicts with a previous read of size 1 by thread #2
==1979==    at 0x4349308: ??? (in /usr/lib/i386-linux-gnu/libQtCore.so
   .4.7.4)
==1979==    by 0x476388B: g_main_context_prepare (in /lib/i386-linux-
   gnu/libglib-2.0.so.0.3000.0)
==1979==    by 0x4764636: ??? (in /lib/i386-linux-gnu/libglib-2.0.so
   .0.3000.0)
==1979==    by 0x4764C29: g_main_context_iteration (in /lib/i386-linux-
   gnu/libglib-2.0.so.0.3000.0)
==1979==    by 0x4349AD9: QEventDispatcherGlib::processEvents(QFlags<
   QEventLoop::ProcessEventsFlag>) (in /usr/lib/i386-linux-gnu/
   libQtCore.so.4.7.4)
```

(continue)

```
==1979==    by 0x431A1DC: QEventLoop::processEvents(QFlags<QEventLoop::
   ProcessEventsFlag>) (in /usr/lib/i386-linux-gnu/libQtCore.so.4.7.4)
==1979==    by 0x56C6387: ???
==1979==  Address 0x48a247c is 44 bytes inside a block of size 56 alloc
   'd
==1979==    at 0x402848F: operator new(unsigned int) (vg_replace_malloc
   .c:255)
==1979==    by 0x421E003: ??? (in /usr/lib/i386-linux-gnu/libQtCore.so
   .4.7.4)
==1979==    by 0x421E0AB: QThread::QThread(QObject*) (in /usr/lib/i386-
   linux-gnu/libQtCore.so.4.7.4)
==1979==    by 0x4579112: (below main) (libc-start.c:226)
==1979==

...

==1979==
==1979==
==1979== For counts of detected and suppressed errors, rerun with: -v
==1979== Use --history-level=approx or =none to gain increased speed,
   at
==1979== the cost of reduced accuracy of conflicting-access information
==1979== ERROR SUMMARY: 170 errors from 41 contexts (suppressed: 882
   from 81)
```