NTNU
Innovation and Creativity

Evaluating Security in Open Source Consumer Applications

Carlos Ballester Lafuente

Master of Science in Computer Science
Submission date: May 2007
Supervisor: Torbjørn Skramstad, IDI
Co-supervisor: Gunnar René Øie, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The aim of this Master Thesis is to develop a software security guideline that will be used for evaluating methods and measuring security in open source projects with a high security implication. One such example is healthcare applications, where the privacy and security are crucial factors.
After developing this guideline, it will be tested against a healthcare software open source project (Indivo) that is being currently developed for the management of patient health records. Security and trust are essential for such an application.
This software involves several usage contexts, several kind of users (patients, doctors, pharmacists) plus system administrators, and it has several components both self-developed and COTS (commercial off-the-shelf).
Implications for other products designed and used in similar fashion will be explained.
Results of evaluation with the guideline will be presented, and if severe security issues are found, they will be reported to the developers.

Assignment given: 26. January 2007
Supervisor: Torbjørn Skramstad, IDI

For my parents, without them I would be nothing,
for my family
and for my friends.

Abstract

Introduction

The aim of this Master Thesis is to develop a software security guideline that will be used for evaluating methods and measuring security in open source projects with a high security implication such as healthcare applications for example, where the privacy and security is a crucial factor.

Background Theory

First section of thesis is focused on presenting the appropriate background theory that will be needed for a good understanding of the rest of the thesis, like vulnerabilities, common security attacks, definition of the client-server technology, risk analysis and specific theory about Indivo and the healthcare field.

Methods

The method chosen to develop the guideline was the waterfall model as time was quite limited and only one iteration could be done. That's why no other methods like the spiral model were used, as they require several iterations until achieving functionality.

Results

After applying the guideline, several vulnerabilities were found, like session hijacking or capturing login information on real time. The guideline proved to be useful in revealing serious security issues that should be fixed, and into describing the purpose and the logic of decisions made in early stages like organizational or design stage.

Conclusion

Both the development of the Security Guideline and the posterior testing of the guideline were considered successful, as a working methodology was established and several security issues were revealed in Indivo.

Preface

This report is a Master's Thesis at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).
The thesis is part of a research project carried out by the NSEP about an Open Source Consumer Application for the healthcare field, which is used to manage patient health records, called Indivo.
Parallel to this thesis, there is other research work being done about access control over the same application.

I would like to thank my supervisor, Gunnar René Øie, for the help and guidance throughout the entire thesis. I would also like to thank Professor Torbjørn Skramstad, Professor Øystein Nytrø, research fellow Lillian Røstad for all the good information given about Indivo and to William Simons and Tim Tisdall as well as the rest of Indivo mailing lists participants for kindly answering my questions about problems with Indivo.

Trondheim, June 2007

Carlos Ballester Lafuente

Contents

1. Introduction 9
1.1 Motivation 9
1.2 Project Context 10
1.3 Problem Definition 10
1.4 Report Outline 10
2. Background Theory 12
2.1 Computer Security 12
2.2 Vulnerabilities 13
2.3 Security Attacks 14
2.3.1 Buffer Overflow Attacks 15
2.3.2 DoS – Denial of Service 16
2.3.3 IP Spoofing & Man-in-the-Middle 16
2.3.4 SQL Injection 17
2.3.5 Cross Site Scripting (XSS) 18
2.3.6 Directory Transversal 19
2.3.7 Session Hijacking 20
2.4 Web Services 21
2.4.1 XML 21
2.4.2 SOAP 22
2.4.3 WSDL 22
2.4.4 UDDI 22
2.5 Open Source and Closed Source 23
2.5.1 Open Source 23
2.5.2 Closed Source 23
2.6 Distributed Object Platform 24
2.6.1 CORBA 24
2.6.2 RMI 25
2.7 Client – Server Architecture 25
2.7.1 Two Tier Architecture 26
2.7.2 Three Tier Architecture 26
2.8 Cryptography 27
2.8.1 Symmetric Cryptography 28
2.8.2 Public Key Cryptography 28
2.9 Security Evaluation Standards 29
2.9.1 Common Criteria 30
2.10 Risk Management 31
2.11 Healthcare Related Theory 32
2.11.1 EHR 32
2.11.2 Indivo 33

3. Research Method 35

3.1 Background Theory Research Method 35

3.2 Guideline Research Method 36

3.3 Testing Methodology 37

4. Security Guidelines 38

4.1 General Guidelines 39
4.1.1 Secure the weakest link 39
4.1.2 Defense in Depth 40
4.1.3 Fail Securely 40
4.1.4 Use the Least Privilege 41
4.1.5 Compartmentalize 41
4.1.6 Keep it Simple 41

4.2 Organizational Guidelines 42
4.2.1 Who Should do Software Security 42
4.2.2 Setting out the Testing Plan 43
4.2.3 Keeping up with the testing effort: Retesting 44
4.2.4 What to Retest 44

4.3 Design Guidelines 45
4.3.1 Choosing a Programming Language 46
4.3.2 Choosing a Distributed Object Platform 47
4.3.3 Choosing an Operating System 49
4.3.4 Closed Source vs. Open Source 50
4.3.5 Risk Management 52

4.4 Implementation Guidelines 55
4.4.1 Code Review 55
4.4.2 Using COTS 57
4.4.3 Using and Reusing Trusted Components 57

4.5 Post Implementation Guidelines 58
4.5.1 Penetration Testing 58
4.5.2 Configuration Management 62

5. Practical Application and Results of the Guideline 63

5.1 General Guidelines in Indivo 63

5.2 Organizational Guidelines in Indivo 64
5.2.1 Selecting the security person 64
5.2.2 Setting the Test Plan 65
5.2.3 When and what to Retest 67

5.3 Design Guidelines in Indivo 68
5.3.1 Choosing the Programming Language 68
5.3.2 Choosing a Distributed Object Platform 69
5.3.3 Choosing an operating System 69
5.3.4 Risk Analysis 70

5.4 Implementation Guidelines in Indivo 73
5.4.1 Code Review 73
5.4.2 Commercials Off-the-Shelf 75

5.5 **Post Implementation Guidelines in Indivo** 77
5.5.1 Penetration Testing 77

6. Discussion 90

7. Conclusion and Further Work 96

7.1 **Further Work** 96

Appendix A: References 98

List of Figures

Figure 1: Buffer Overflow [26] 15
Figure 2: Session Fixation Attack [27] 20
Figure 3: Web Services. [28] 21
Figure 4: CORBA Server [29] 24
Figure 5: Java RMI [30] 25
Figure 6: Two Tier Architecture [31] 26
Figure 7: Three Tier Architecture [32] 27
Figure 8: Public Key encryption/decryption [33] 29
Figure 9: Risk Management [16] 32
Figure 10: Indivo 3-tier architecture [14] 33
Figure 11: Waterfall model for the guideline 36
Figure 12: Configuration for FindBugs 73
Figure 13: FindBugs result report. 74
Figure 14: Server Fingerprint. 78
Figure 15: Port Scan results. 78
Figure 16: Web Scarab. 79
Figure 17: Numerical analysis for session IDs. 80
Figure 18: Distribution analysis for session IDs. 80
Figure 19: Changing an Indivo request 81
Figure 20: Accessing an administrative interface. 82
Figure 21: Indivo Error message. 83
Figure 22: Accessing unintended files. 84
Figure 23: Capturing login information. 85
Figure 24: Editing request to forge a stolen SessionID. 86

List of Tables

Table 1: Vulnerability severity ranking, impact and control for Indivo. 72

1. Introduction

An Open Source consumer application is a piece of software whose source code is made available to the public through a special kind of license that permits users to study, modify and make improvements in the application, as well as to redistribute it in its original shape or in a modified way.
Open Source consumer applications, as all the rest of the software, are exposed to a high amount of security threats, but this becomes a bigger problem for this kind of software because as the code is available to everyone, it is easier to search and to find flaws and security holes on them.

1.1 Motivation

Each day software security becomes more and more important. Usually, people tends to put more efforts or to emphasize on network security, but we should realize that no matter how strong our network security is, if the software running inside our machines is not secure (i.e. it has some kind of bug, flaw that allows the code to be exploited), attackers will always find the way to break security. Lets say you can try to protect your network with a really good firewall with a really methodical configuration; even if the firewall is 100% perfect, if some of the software allowed to connect through the firewall is not secure, attackers will have an easy way to get into, without you even noticing it, not to say that the firewall itself can have some security flaw.

There are several reasons why developing a guideline about software security is interesting:

1. Security is the main system's feature to protect confidential data, only developing and having secure software, confidential data will be as safer as possible. And that's really a crucial factor for a lot of software nowadays.

2. It should teach software designers that software security is an important thing to take care about from the very beginning, and not only trying to fix security holes with patches, after software has already been released.

1.2 Project Context

The thesis is part of a research project carried out by the NSEP about an Open Source Consumer Application for the healthcare field, which is used to manage patient health records, called Indivo.
Parallel to this thesis, there is other research work being done about access control over the same application.
Because of the data that this kind of program deals with (personal health record information), security on this context is a crucial factor.

1.3 Problem Definition

The aim of the thesis is to develop a guideline for software security that will be used for evaluating methods and measuring security in open source projects with a high security implication.
The guideline will not only cover evaluating and measuring security after the development of the product, but also in the very early stages of its development.
Developing secure software is something that must be carried out since the very first stage of development of a piece of software.
Checking security only afterwards and patching the product, is not only an extra effort that can be saved if security is present in the early life of a product, but also it can introduce new security flaws in the product that would need subsequent patches.
After the guideline will be fully developed, it will be tested against an Open Source Software Application for the healthcare field (Indivo), which is still in development stage.

1.4 Report Outline

The structure of the report is as follows:

- Chapter 2 presents background theory about security, common technologies used in application development and specific Indivo and healthcare theory.

- Chapter 3 gives a brief presentation and summary of the methods used in the development of this thesis.
- Chapter 4 develops the main point of this thesis, the Security Guideline.
- Chapter 5 applies the Security Guideline to a real application: Indivo.
- Chapter 6 is a discussion of the thesis and of the results obtained when testing the guideline.
- Chapter 7 gives conclusions for the thesis and proposes further work that can be done.
- Appendix A contains the references used in the development of the thesis.

2. Background Theory

This chapter gives a theoretical background for the thesis, defining the most important concepts and areas that are needed for a good understanding of subsequent chapters.

First of all, I give an overview about what Computer Security is. After what is a vulnerability is defined and after having this information, I explain what Security Attacks are, and the possible types of attacks.

The chapter continues with a fast overview of Web Services, and after this, the two ways of developing software are presented: Open source and Closed source.

I then present some of the most important Distributed Object Platforms, to continue after this with an overview of the Client - Server architecture. I continue describing what Cryptography is and we give a short view on cryptographic methods.

After this, I describe what a Security Evaluation Standard is and why not to use Common Criteria on the security evaluation of this thesis is discussed. This chapter continues introducing some theory and concepts of what Risk Management is and its importance in security evaluation.

Last but not least, I introduce some important theory background about healthcare and Indivo healthcare application for a better understanding of the analysis part of the thesis.

2.1 Computer Security

"The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards - and even then I have my doubts". - Eugene H. Spafford [1]

Computer security is defined as the ability of protecting information and system resources respecting crucial concepts as confidentiality, integrity, authentication, and availability. This definition not only includes the information "per se", but also other computer elements such as disks, CPU and programs.

Confidentiality ensures that information is not accessed by unauthorized persons. Usually this is obtained by the use of access control and cryptography.

Integrity refers to the fact that information is not modified by unauthorized persons in a way that is not possible to detect by rightful owners/recipients of that information.

Authentication is any process by which you verify that someone is who they claim they are. This usually involves a username and a password, but can include any other method of demonstrating identity, such as a smart card, retina scan, voice recognition, or fingerprints [2].

Availability refers to the ability of using the information or resource desired in a given moment. The loss of availability, if caused by an attack, is often referred as DoS – Denial of Service.

Poor computer security is caused by vulnerabilities in software. This allows a malicious user to exploit the vulnerability through an attack.

2.2 Vulnerabilities

A security vulnerability is a flaw in a product that makes it infeasible - even when using the product properly - to prevent an attacker from usurping privileges on the user's system, regulating its operation, compromising data on it, or assuming ungranted trust as said in Microsoft TechNet [3].
Often, the word vulnerability and attack are used without difference, but note that an attack is possible because of a vulnerability; they are different things and we need to keep them separately.
Simple said vulnerabilities provide the entry point for an attacker to launch an attack.

There are many kind of vulnerabilities, and description of every vulnerability is out of the scope of this thesis. The following examples of vulnerabilities are only for better understanding of the theory for the reader.

Cross Site Scripting Flaws occur whenever an application takes user supplied data and sends it to a web browser without first validating or encoding that content.

Injection Flaws occurs when user supplied data is sent to an interpreter as part of a command or query. The attacker's hostile data tricks the interpreter into executing unintended commands or changing data.

Improper Error Handling occurs when applications can unintentionally leak information about their configuration, internal workings, or violate privacy through a variety of application problems. Attackers use this weakness to violate privacy, or conduct further attacks.

Unvalidated Input occurs when information from web requests is not validated before it is used by a web application. Attackers may exploit vulnerabilities of this kind to attack back end components through a web application.

These are only few vulnerabilities of the many existing. These vulnerabilities and many more existing, provide an "attack launch point" for the malicious user.

A good and very complete overview of application vulnerabilities can be found in OWASP website [4].

2.3 Security Attacks

A security attack is an action towards a system to exploit a vulnerability. Usually the aim of an attack is to gain access to confidential information that the attacker is not allowed to see or to get, or to gain the control of a system in an illegitimate way.
Knowing how attacks work is important for being able to understand how computer security works.
In the following part of this section I will describe how several important well known attacks work.

2.3.1 Buffer Overflow Attacks

"On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind." Aleph One [5]

Buffer overflows occur when a function in an application fails to check the size of the input data. If the input data size is bigger than the size the application has reserved for the data, it doesn't fits and overwrites other memory locations in the execution stack as can be seen in Figure 1.

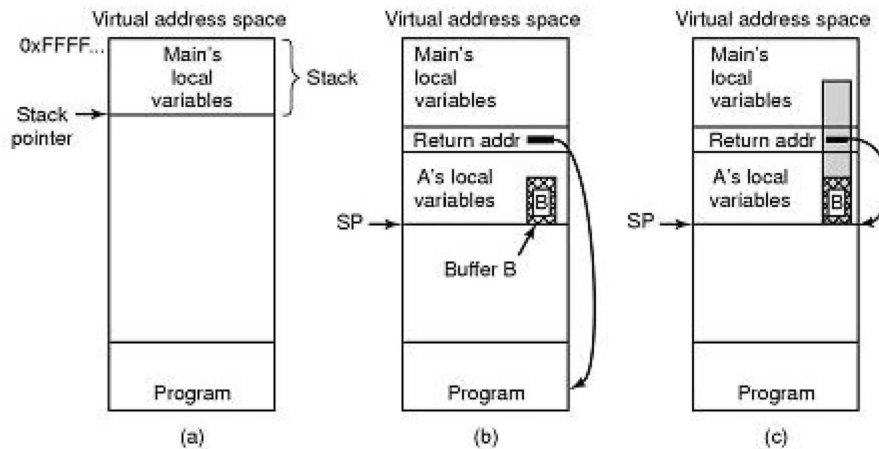

Figure 1: Buffer Overflow [26]

a) Program before invoking function A.

b) Function A is invoked and it books space for buffer B.

c) Input in buffer B is bigger than it can hold, resulting in other memory locations of execution stack being overwritten (grey color).

This situation by itself only will make the program to crash, nothing more.

The dangerous situation occurs when the attacker uses the overflow to overwrite the return address with the address of some other piece of code, that when executed, will result in giving the control of the machine to the attacker.

2.3.2 DoS – Denial of Service

Basis for this attack is that executing code takes time.
As Andrews and Whittaker [6] say, for each function called by the Web server, the application or the database a certain number of processor cycles are used in executing that function.
If that operation is long lived and the operating system cannot switch to any other task, the machine will be tied up servicing only that request. Even if the machine is multithread or multiprocess, if we can flood it with enough request to service long running operations, we can deny other users from accessing the service that the machine is offering.
This kind of attack is called DoS or Denial of Service.
The Denial of Service attack violates directly one of the principles of computer security, the availability.

2.3.3 IP Spoofing & Man-in-the-Middle

IP Spoofing consists in forging the IP address of the packets going from the attacker's machine to the victim's machine, so the victim will think that packets actually come from another machine different than the one they are coming really.
This attack is useful when trying to appear as a trusted host to get benefits from what that trust relation offers for the rightful machine.
The difficulty on IP Spoofing attacks relies in that the response is sent to the forged IP and not to the attacker's machine, so attacker should be quite skilful to get back that response, or to be in the same network that the victim's machine to be able to capture the packets that are being sent to the forged IP machine.
IP Spoofing is used as part of more sophisticated attacks like Man-in-the-Middle attacks, where the attacker's machine is in the middle of the communication line between two other hosts A and B (the victims).
By forging the IP address of the packets the attacker can make to believe the victim machine A that he is the victim machine B and to make that the victim machine B thinks that he is the victim machine A, thus being able to view, capture and modify every piece of information that will go from A to B without any substantial effort, becoming that way the "Man-in-the-Middle".

2.3.4 SQL Injection

This attack can be launched against any application that uses user-supplied input to dynamically construct SQL queries against a database.
The problem here relies in the attacker trying to trick the application into changing the nature of the query by entering SQL artifacts into that user input, giving him unrestricted access to the database.
This attack is more easy and clear to understand when looking through an example about how to conduct this attack:

Lets suppose that login into some system is achieved by performing a SQL query against a database with the login information that the user provides. That means that the SQL query is constructed dynamically with the input that user supplies, in this case a login username and a password.
The query that will retrieve user information with the database would look like this:

SELECT userdata FROM userinfo
WHERE userid = 'id'
AND password = 'pwd'

User controlled strings in that query are the ones between single quotes.
Normally, user would have to put both the userid and the password correctly for the SQL query to result in a successful login. But a skilled attacker can use SQL language to make that the query will give him access not even knowing the userid.
In SQL the double dash operator means a comment (--). Therefore we can alter the construction of the query by inserting specific SQL tokens into the user input fields. If in the userid field we would type *' OR 1=1 --* the following query will be generated:

SELECT userdata FROM userinfo
WHERE userid = ' ' OR 1=1

As anything "ORed" with a true statement like 1=1 (that is always true), is evaluated like true, we would obtain the userdata from every user at the database without having to know any single userid or password. Note that with

the double dash (--) we manage to make the password part of the query to be interpreted as a comment and thus not taking part in the real query that will be sent to the database [6].
In the same way, we can alter queries to create new users on a database, to drop tables of a database or whatever thing that SQL sentences can permit.

2.3.5 Cross Site Scripting (XSS)

This attack is based on the fact that websites often echo the data that is entered at some other place within the application. The problem is when that data doesn't contain simple HTML plaintext but some kind of script that will be executed into other machines visiting the site. Moreover, the biggest problem comes when that data that other people's machine will view is possible to be entered by a totally random person.

Let's take the example of the guestbooks at websites. Usually, users are free to write whatever they want, and other users will read that comments. But with HTML tags such as <script> one attacker can supply data that will be seen by other users in the shape of a script, which will be executed in the victim's machine.

The effects of a XSS attack can vary from simple annoyance to complete account compromise.
The most severe XSS attacks involve disclosure of a users session cookies. Other damaging attacks include the disclosure of end user files, installation of trojan horses, redirecting a user to some other page or site, and modifying presentation of content.

As Andrews and Whittaker explains in their book [6], retrieving a user's active cookie is as simple as embedding in some user data input, a script like this:

```
<script>
document.write("<img src=http://attacker.com/
px.gif?cookie="+document.cookie")
</script>
```

When another user will visit a page that displays that malicious input, the attacker's server log, where that image is hosted, will store his session id when the victim's browser is requesting the image so the attacker can get valid session identifiers in real time.

2.3.6 Directory Transversal

As OWASP website [4] explains, this category of attacks exploits various path vulnerabilities to access files or directories that are not intended to be accessed. This attack works on applications that take user input and use it in a "path" that is used to access a file system. If the attacker includes special characters that modify the meaning of the path, the application will misbehave and may allow the attacker to access unauthorized resources.
This type of attack has been successful on web servers, application servers, and custom code.

The most basic Directory Traversal attack uses the '../' special character sequence to alter the location of the request. By using this special character an attacker might be able to retrieve a file directly that he was not supposed to have access to.

Another approach is to target the application itself. Most commonly the use of parameters being passed by the application can be exploited. Such data can come from user input or application data being passed between pages. Let's take the example of: http://victim.com/getfile.asp?file=xxxfile.htm.
We can observe from the above that "getfile.asp" takes a parameter to navigate through the application; in this case, the file location is xxxfile.htm. We can use this knowledge to attempt the retrieval of getfile.asp's source code by submitting: *http://victim.com/getfile.asp?file=getfile.asp*.

We can combine these two approaches to try to access every file we want into the server.

2.3.7 Session Hijacking

The attacker's objective in session hijacking is to masquerade as another user by stealing that person's identifying credentials and using them for his own profit. The most common way of achieving this is to steal the user's session identifier by diverse methods. It is also possible not to steal but to give directly a user a "compromised session".

Attacks taking part on this particular attack can involve Cross Site Scripting to get session identifiers in real time as it was discussed previously, monitoring network traffic or guessing future session identifiers because of poorly implemented session handlers.

Another particular way of developing this attack is to do what is called "session fixation". This particular way of session hijacking attack, involves stealing the session ID before the legitimate user will ever get it. With this technique, the attacker is able to take the session from the legitimate user every time is profitable to do so. For this purpose, attacker should get a valid session ID and provide it to a user by giving a link to the web page with the session ID already included in the link. Because the session is a valid one, the victim doesn't notice the difference. However, the difference is that in this case, attacker can assume the victim's identity in any moment.

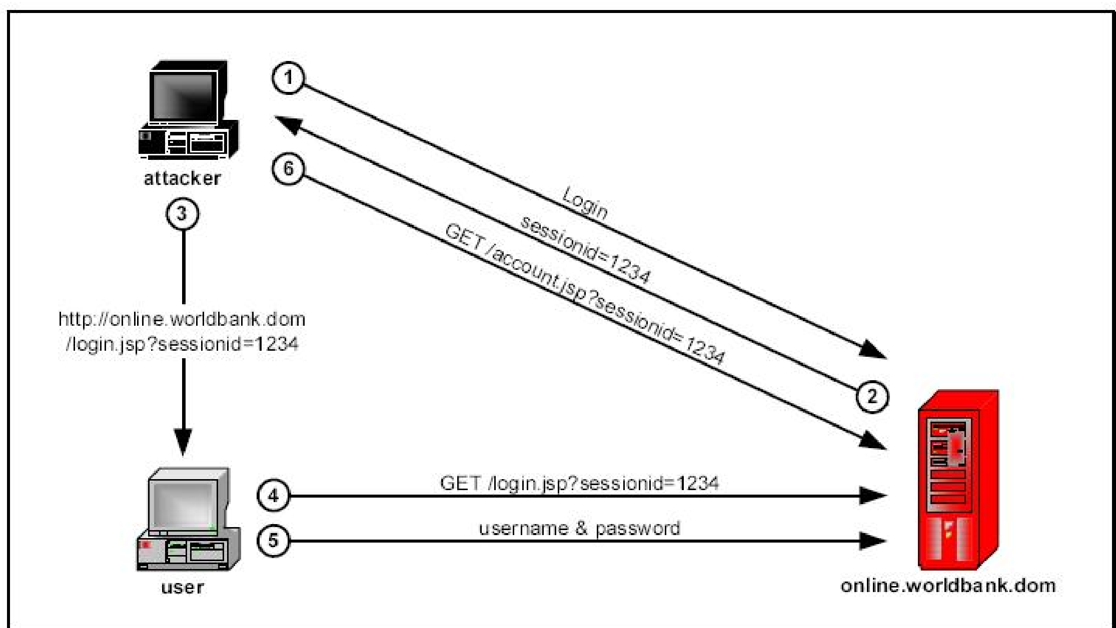

Figure 2: Session Fixation Attack [27]

2.4 Web Services

The W3C [7] defines a Web service as a software system designed to support interoperable Machine to Machine interaction over a network. Web services are frequently just Web APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services.
The W3C Web service definition encompasses many different systems, but in common usage the term refers to those services that use SOAP-formatted XML envelopes and have their interfaces described by WSDL.

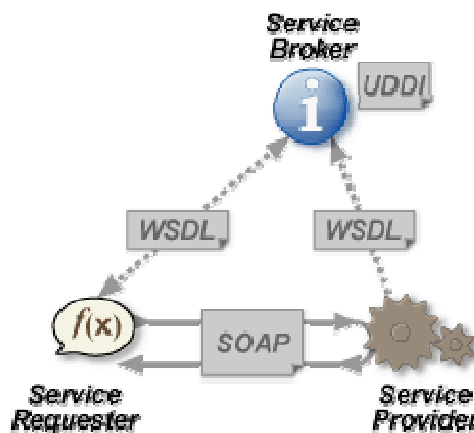

Figure 3: Web Services. [28]

Four technologies take part on web services: eXtensible Mark-up Language (XML), Simple Object Access protocol (SOAP), Web Services Description Language (WSDL) and Universal Discovery Description and Integration (UDDI).

2.4.1 XML

EXtensible Mark-up Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [7].

XML uses a similar tag structure as HTML; however, whereas HTML defines how elements are displayed, XML defines what those elements contain. While HTML uses predefined tags, XML allows tags to be defined by the developer of the page. Thus, virtually any data items, such as "id," "price" and "vendor" can be identified, allowing Web pages to function like database records. By

providing a common method for identifying data, XML supports business-to-business transactions and has become "the" format for electronic data interchange and Web services.

2.4.2 SOAP

Soap is a way to transport XML from one end point to another. It supports almost every standard transmission protocol such as TCP, HTTP, and SMTP.
The aim of SOAP is to provide an envelope to XML messages so they can be carried by a variety of transport mechanisms.
SOAP messages are divided into two parts, the header and the body.

2.4.3 WSDL

WSDL is an XML-based service description on how to communicate using web services. The WSDL defines services as collections of network endpoints, or ports. WSDL specification provides an XML format for documents for this purpose.
Messages are abstract descriptions of the data being exchanged, and port types are abstract collections of supported operations. In this way, WSDL describes the public interface to the web service.
WSDL is often used in combination with SOAP and XML Schema to provide web services over the Internet. A client program connecting to a web service can read the WSDL to determine what functions are available on the server. Any special datatypes used are embedded in the WSDL file in the form of XML Schema. The client can then use SOAP to actually call one of the functions listed in the WSDL [8].

2.4.4 UDDI

UDDI is a platform-independent, XML-based registry for businesses worldwide to list themselves on the Internet. UDDI is an open industry initiative enabling businesses to publish service listings and discover each other and define how the services or software applications interact over the Internet.

It is designed to be interrogated by SOAP messages and to provide access to Web Services Description Language documents describing the protocol bindings and message formats required to interact with the web services listed in its directory [8].

2.5 Open Source and Closed Source

"As far as firms are concerned, they should take strict measures to ensure that sensitive information is only transmitted via secure media.... If security is to be taken seriously, only those operating systems should be used whose source code has been published and checked, since only then can it be determined with certainty what happens to the data." [9]

Open source and closed source are two approaches to the control, exploitation and commercializing of computer software.
Open source approaches differ from the traditional model of software licensing by allowing other individuals and organizations to view and modify the source code, and in many cases, resell the software without providing royalties to the original authors of the software, or under some open source licenses without even requiring that they credit the original authors of the software.

2.5.1 Open Source

Open-source software is an antonym for closed source software and refers to any computer software whose source code is available under a license (or arrangement such as the public domain) that permits users to study, change, and improve the software, and to redistribute it in modified or unmodified form. It is often developed in a public, collaborative manner. It is the most prominent example of open source development [8].

2.5.2 Closed Source

Closed source refers to any program whose license does not meet the definition of Open-source software. Generally, it means only the binaries of a computer program are distributed and the license provides no access to the program's

source code, rendering modifications to the software technically impossible for practical purposes. The *source code* of such programs is usually regarded as a trade secret of the company [8].

2.6 Distributed Object Platform

These days, client/server applications are being built with software systems based on distributed objects. These technologies provide for remote availability of resources, redundancy and parallelism [10].

2.6.1 CORBA

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to interoperate [8].
CORBA inserts program code into a “pack” containing information about the capabilities of the code inside and how to call it. The resulting objects can then be called from other programs (or CORBA objects) across a network.
CORBA uses an interface definition language (IDL) to specify the interfaces that objects will present to the world. CORBA then maps from IDL to a specific implementation language like C++ or Java.

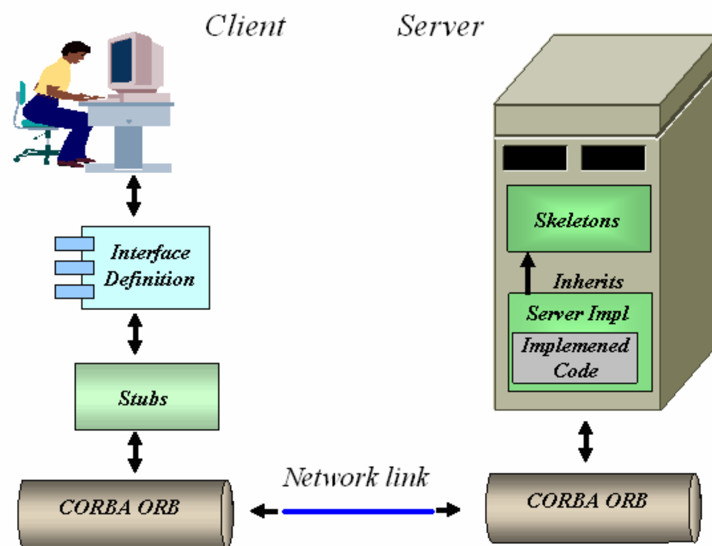

Figure 4: CORBA Server [29]

2.6.2 RMI

JAVA Remote Method Invocation (RMI), it's a mechanism offered by Java to invoke a method remotely.
Using RMI, a program can export an object. From that moment, the object is available for every client to connect to it and to use its methods.
The way RMI works is very similar to CORBA with the exception that RMI can be only used with Java coded servers. In the other hand, it provides passing object by reference and automated garbage collection, something that CORBA doesn't have.

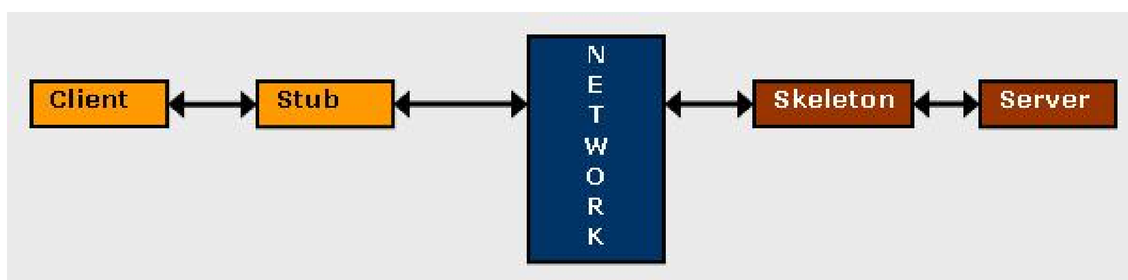

Figure 5: Java RMI [30]

2.7 Client – Server Architecture

Client - server is network architecture which separates a client (often an application that uses a graphical user interface) from a server. Each instance of the client software can send requests to a *server*. Specific types of servers include web servers, application servers, file servers, terminal servers, and mail servers. While their purposes vary somewhat, the basic architecture remains the same.

The term client/server was first used in the 1980s in reference to personal computers (PCs) on a network. The actual client/server model started gaining acceptance in the late 1980s.
The client/server software architecture is a versatile, message-based and modular infrastructure that is intended to improve **usability**, **flexibility**, **interoperability**, and **scalability** as compared to centralized, mainframe, time sharing computing.

A client is defined as a requester of services and a server is defined as the provider of services. A single machine can be both a client and a server depending on the software configuration [11].

2.7.1 Two Tier Architecture

As can be seen in figure 6, with two tier client/server architectures, the user system interface is usually located in the user's desktop environment and the database management services are usually in a server that is a more powerful machine that services many clients.
Processing management is split between the user system interface environment and the database management server environment. The database management server provides stored procedures and triggers.

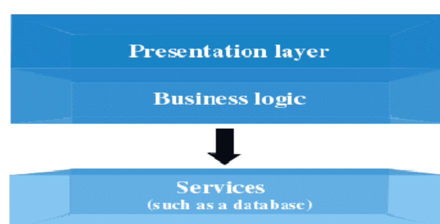

Figure 6: Two Tier Architecture [31]

2.7.2 Three Tier Architecture

This architecture consists, as shown in figure 7, of three logical tiers, namely the presentation tier, application tier and the data tier.

The first tier, the presentation tier, is responsible for presenting data to end users or systems. This tier includes web browsers and web servers. They may also include application components that create the page layout.

The second tier, the application tier, is the engine of a web application. It performs the business logic, like processing user input, making decisions,

obtaining more data, and sends data to the presentation tier which presents it to the user.

The third tier is the data tier, which is used to store things needed by the application, and acts as a repository for both temporary and permanent data.

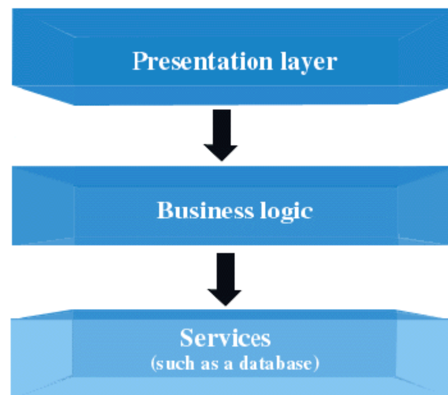

Figure 7: Three Tier Architecture [32]

2.8 Cryptography

As Kessler says [12], in data and telecommunications, cryptography is necessary when communicating over any untrusted medium, which includes just about any network, particularly the Internet.
Within the context of any application-to-application communication, there are some specific security requirements, including:

Authentication: The process of proving one's identity.

Privacy/confidentiality: Ensuring that no one can read the message except the intended receiver.

Integrity: Assuring the receiver that the received message has not been altered in any way from the original.

Non-repudiation: A mechanism to prove that the sender really sent this message.

Cryptography, then, not only protects data from theft or alteration, but can also be used for user authentication.

There are, in general, two types of cryptographic schemes typically used to accomplish these goals: secret key (or symmetric) cryptography and public-key (or asymmetric) cryptography.

In all cases, the initial unencrypted data is referred to as plaintext. It is encrypted into ciphertext, which will be decrypted into usable plaintext again.

2.8.1 Symmetric Cryptography

With symmetric cryptography, a single key is used for both encryption and decryption.

With this form of cryptography, it is obvious that the key must be known to both the sender and the receiver. The biggest difficulty with this approach, of course, is the distribution of the key.

Secret key cryptography schemes are generally categorized as being either stream ciphers or block ciphers.

Stream ciphers operate on a single bit (byte or computer word) at a time.

A block cipher is so-called because the scheme encrypts one block of data at a time using the same key on each block.

Among the symmetric cryptography algorithms, the most important are:

DES (Data Encryption Standard) with all its variants like tripleDES or DESX.

AES (Advanced Encryption Standard).

2.8.2 Public Key Cryptography

Public key cryptography, also known as asymmetric cryptography, is a form of cryptography in which a user has a pair of cryptographic keys - a public key and a private key.

The private key is kept secret, while the public key may be widely distributed. The keys are related mathematically, but the private key cannot be practically derived from the public key. A message encrypted with the public key can be decrypted only with the corresponding private key.

The biggest advantage is that private key never needs to be distributed, thus making a lot easier to keep it secret.

Anyone can encrypt a message using the public key, but only the holder of the private key related to the public one can decrypt the message. Secrecy depends on the secrecy of the private key.

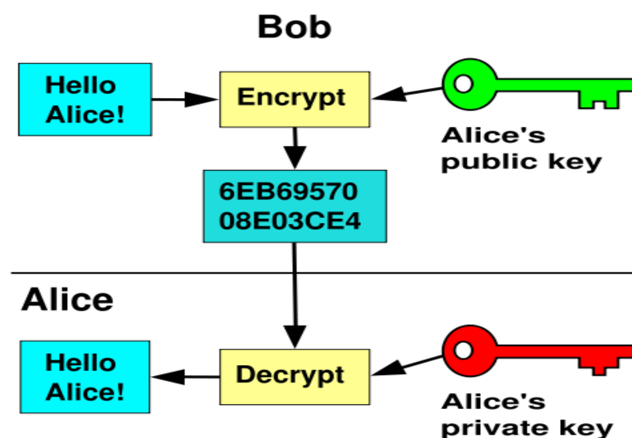

Figure 8: Public Key encryption/decryption [33]

Among the public key algorithms, probably the most famous is the RSA algorithm that has that name due to the initials of its creators, Rivest, Shamir and Adleman.

2.9 Security Evaluation Standards

For security evaluation purposes, entities and governments have created what are called security evaluation standards, which are used to certify that a piece of software is secure enough according to that standard.

In this section I will present the most important one, the Common Criteria, and after, why not to use such a standard in the development of our security guideline will be discussed.

2.9.1 Common Criteria

The Common Criteria (CC) is an international standard (ISO/IEC 15408) for computer security. Unlike standards such as FIPS 140, Common Criteria does not provide a list of product security requirements or features that products must contain.

Instead, it describes a framework in which computer system users can specify their security requirements, vendors can then implement and/or make claims about the security attributes of their products and testing laboratories can evaluate the products to determine if they actually meet the claims.

In other words, Common Criteria provides assurance that the process of specification, implementation and evaluation of a computer security product has been conducted in a rigorous and standard manner [8].

For the development of this guideline, Common Criteria will not be used because Open Source Applications are in constant development and change and can be used under many computer configurations, and Common Criteria only certifies that a product is secure under certain configuration specified by the vendor and for a specified severity of threats only.

As Viega and McGraw say [10], *"although the Common Criteria is certainly a good idea, security evaluation is unfortunately not as simple as applying a standard protection profile to a given target of evaluation. The problem with the Common Criteria is evident right in its name. That is, "common" is often not good enough when it comes to security."*

2.10 Risk Management

Risk management is the process of measuring, or assessing, risk and developing strategies to manage it. Strategies include transferring the risk to another party, avoiding the risk, reducing the negative effect of the risk, and accepting some or all of the consequences of a particular risk.

In ideal risk management, a prioritization process is followed whereby the risks with the greatest loss and the greatest probability of occurring are handled first, and risks with lower probability of occurrence and lower loss are handled later. In practice the process can be very difficult, and balancing between risks with a high probability of occurrence but lower loss vs. a risk with high loss but lower probability of occurrence can often be mishandled [8].

As said in [13], risk analysis can be divided in three processes: risk assessment, risk mitigation and evaluation and assessment.

Risk assessment is the first process in the risk management methodology. Organizations use risk assessment to determine the extent of the potential threat and the risk associated with an IT system.
The output of this process helps to identify appropriate controls for reducing or eliminating risk during the risk mitigation process.

Risk mitigation, the second process of risk management, involves prioritizing, evaluating, and implementing the appropriate risk-reducing controls recommended from the risk assessment process.
Because the elimination of all risk is usually impractical or close to impossible, using the least-cost approach and implement the most appropriate controls to decrease mission risk to an acceptable level, with minimal adverse impact on the organization's resources and mission, is the usual method.

The need for **evaluation and assessment** comes with the fact that in most organizations, the network itself will continually be expanded and updated, its components changed, and its software applications replaced or updated with newer versions. In addition, personnel changes will occur and security policies are likely to change over time.

These changes mean that new risks will surface and risks previously mitigated may again become a concern. Thus, the risk management process is ongoing and evolving.

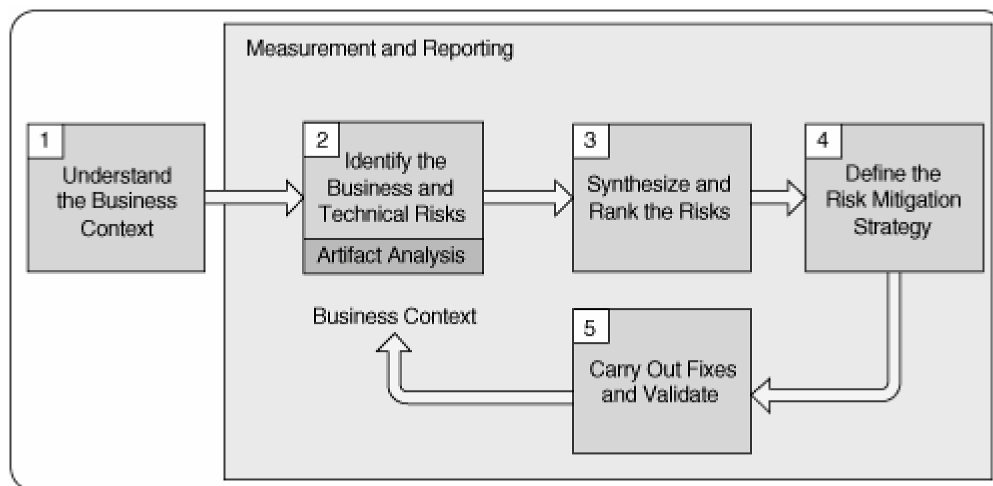

Figure 9: Risk Management [16]

2.11 Healthcare Related Theory

As the analysis part of this thesis is conducted against an Open Source Software Application for the healthcare field, which is used to manage electronic health records, some specific healthcare related theory is needed. In this section a fast overview on EHRs (Electronic Health Records) and on Indivo will be done.

2.11.1 EHR

An electronic health record (EHR) is a distributed personal health record in digital format. The EHR provides secure, real-time, patient-centric information to aid clinical decision-making by providing access to a patient's health information at the point of care.

An EHR is typically accessed on a computer or over a network. It may be made up of health information from many locations and/or sources, including electronic medical records (EMRs).

An EHR almost always includes information relating to the current and historical health, medical conditions and medical tests of its subject. In addition, EHRs may contain data about medical referrals, medical treatments, medications and their application, demographic information and other non-clinical administrative information [8].

2.11.2 Indivo

Indivo is a personally controlled health record system developed by the Children's Hospital Informatics Program (CHIP).

The Indivo system is based on a 3-tier architecture: Indivo client, Indivo server and the Store. A communication protocol is developed for client-server communication. The client interface is a PHP web based interface. Java RMI handles the communication between the Indivo server and the Store.

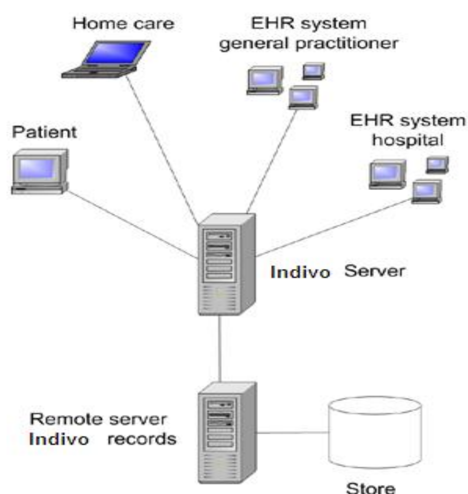

Figure 10: Indivo 3-tier architecture [14]

Some of the most notorious characteristics of Indivo are [14]:

- Personally controlled
- Distributed
- Web-based
- Ubiquitously accessible to the nomadic user
- Built to public standards
- Open-source

3. Research Method

In this chapter, I am going to describe the research methods employed to develop this mater thesis.
First of all, I will describe the research methodology for the theory background part, and after this, the methodology for the development of the security guideline will be introduced.
Finally I will describe the methods I will use to test the guideline against Indivo, an Open Source Consumer Application for the healthcare field.

3.1 Background Theory Research Method

For the Theory Background part of this thesis, first of all an initial schema of important theory concepts related with the content of this thesis was made.
After this, looking through the initial schema, the most important and relevant topics for the good understanding of the thesis were selected, discarding those that were too general or not that relevant.

After this first step, the remaining topics left in the schema were discussed with the thesis teacher and final adjustments on the schema were made, changing some topics and adding some relevant information that was not included yet.

Gathering of information for the theory background was made through the reading and summarizing of several topic-related books like [6] or [10] and through the search on Internet in general information websites or specialized security websites like OWASP or insecure.org.

Finally, like for every chapter of this thesis, the Background Theory part was sent to thesis teacher for feedback and readjustments.

3.2 Guideline Research Method

During the development of the guideline, work will be done according to the waterfall method, as shown in figure 11.

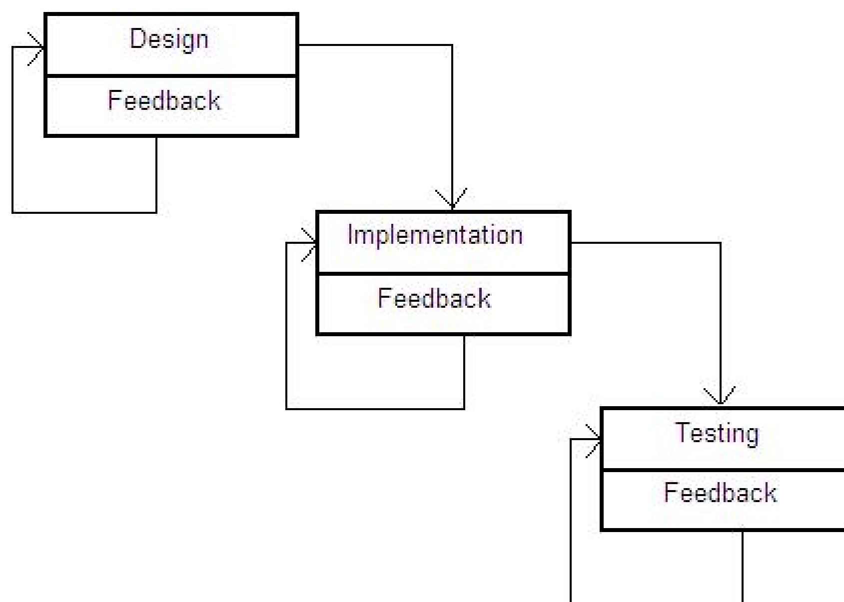

Figure 11: Waterfall model for the guideline

The waterfall model particularly expresses the interaction between subsequent phases in a development process. It is important to be aware that before moving on to a subsequent phase, feedback should be received to assess that everything is correct.

The reason why the waterfall model has been chosen for the development of the guideline, as opposed to various alternatives, is that there is time to do only one iteration, and in another model like spiral model, more than one iteration is needed.

For the design and implementation of the guideline several books and other relevant sources (internet, mailing lists, etc…) related to the subject of this thesis - the security in software - will be used to get the most important keys about building and testing secure software.

3.3 Testing Methodology

As said in earlier chapters, the testing of the guideline will be done against an Open Source Consumer Application for the healthcare area called Indivo.

As this application is already in beta testing stage, not all the guideline will be applied, the part of the guideline referring to the security during design stage will be skipped mostly on the testing.

For this stage of the thesis, it will be necessary to collect much information about vulnerabilities in specific pieces of software and about how to exploit those vulnerabilities.
All this research and information will be conducted searching in specialized pages such as Security Focus, Secunia or Cert website.

After the guideline will be tested against Indivo, results obtained will be presented and conclusions will be drawn.

4. Security Guidelines

"There are two ways of constructing secure software: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult." adapted from C. A. R. Hoare

In this chapter the main focus of this thesis, the Security Guideline, will be developed.

This security guideline is intended to evaluate security on Open Source Consumer Applications.
Overall, this guideline is a compilation of general and not so general "touch points", methods and techniques that are well organized in sections and that provide a good methodology for testing and evaluating security in Open Source Consumer Applications.
But this guideline can be used to evaluate security almost in every piece of software pretty much, open or not, despite the points that imply testing parts of the project that obviously are not available to every person willing to evaluate security on a closed source product.

Now, and before going deep into the guideline, I will present the structure that will conform it.
First of all general guidelines, that are common sense concepts, will be introduced.

After this, I will describe organizational guidelines, that involve the planning of the security evaluation tasks such as who is the most appropriate person to evaluate security, how to keep up with new vulnerabilities and fixes and so on.

Once organizational guidelines are done, it will be the turn for design guidelines to be taken into consideration. These will include orientation for design decisions like which programming language to choose, risk management and more.

Then, implementation guidelines will follow, including among several subjects code review or using and reusing trusted components.

Last but not least, post implementation guidelines will close this chapter, including the testing of the product part, or the importance of a correct installation of the product and the necessary technologies to make it run.

4.1 General Guidelines

This part of the guideline is intended to give few really general touch points on security.
That they are general doesn't mean that they are not important, and one should always check them in order to have a stable "starting point" to go further in the security analysis of an application.

As Viega and McGraw [10] say, there are some guiding principles that can help to improve the security outlook of an application. Following I will describe those principles in the order the authors use in the mentioned book.

4.1.1 Secure the weakest link

It is said that security is like a chain. Putting it simple, a software security system is only as secure as its weakest component, the same that a chain is only as strong as its weakest link.
If someone is preparing to attack an application or system, he will try to attack the weakest part of it, because it will be easier to break into it.

The weakest part of your system can be the software, or it can be its surrounding infrastructure such as other technologies needed to make that software run or some service you might be offering and that it is not secure.
The way to act is to fix first the weakest components, and not the ones that are easier to fix, like lots of people do.

Of course, this strategy can be applied forever, because after securing the weakest component, the next insecure item will become the weakest one. 100% security is never attainable, but securing the most obvious weak parts first is always a good practice to keep in mind.
Securing the weakest parts of your system first always pays off.

4.1.2 Defense in Depth

"Have a series of defenses so that if an error isn't caught by one, it will probably be caught by another". MacLennan

The idea behind defense in depth is to provide more than one defensive strategy, so if one of them will fail or will result inappropriate, a subsequent defensive layer will mitigate the attack.

It is important to make sure that various layers of security in a system works together in harmony to supplement each others functionality, but also to overlap, so that if one should fail, another layer will prevent a total compromise of the system, as said in OWASP [15].

4.1.3 Fail Securely

Any complex system has failure modes. Failure is unavoidable and should be planned in detail. What is avoidable are security problems related to failure. Many systems, when failing, exhibit insecure behaviors, and that's what should be tried to avoid.

Any security mechanism should be designed in such a way that when it fails, it fails closed. That is, it should fail to a state that rejects all subsequent security requests rather than allow them. [15].

When a system fails, it may be a problem that it reveals critical system information as the error messages can contain information of weaknesses on the system that can be used by an attacker such as revealing the name and fields of a database.
Detailed error messages should therefore never be sent to the client, but rather to a system administrator, or to a log.

4.1.4 Use the Least Privilege

The principle of least privilege is based in that only the minimum access necessary to perform an operation should be granted, and only during the minimum amount of time necessary.
When you give access to parts of a system, there is always the possibility that the privileges associated with that access will be abused.

4.1.5 Compartmentalize

The basic idea behind compartmentalization is to minimize the amount of damage that can be done to a system by breaking up the system into as few units as possible while still isolating code that has security privileges

This principle has to be used in moderation; segregating each little bit of functionality will make your system or application almost impossible to manage.

4.1.6 Keep it Simple

Complexity increases the risk of problems. Avoiding complexity, you avoid problems at the same time.

A complex design is never easy to understand, and is therefore more likely to include subtle problems.
Complex code tends to be harder to maintain as well. To stay out of this kind of trouble, design and implementation should be as straightforward as possible in a software development process.

Simplicity can be attained by making all security operations go through one or more choke-points where the traffic going through it will be checked, instead of implementing complex and elaborated security controls that subtracts usability to the final application.

4.2 Organizational Guidelines

This part of the guideline looks at the key decisions that need to be made while planning the organization of the project.

These decisions involve choosing the person or persons to carry out the security supervision and testing effort, setting out the testing plan and deciding when and under which circumstances to retest and what to include in the retesting effort.

4.2.1 Who Should do Software Security

As Gary McGraw says [16], as it stands in many organizations, software security is nobody's job.

Developers, architects, and other builders are often unaware of security and possess little in the way of software security knowledge. When their software suffers from security failure, they don't often feel responsible, arguing that security is up to the people in operations who install and operate the software they create.

When a security problem happens because of bad software, there really is nobody to hold responsible. The standard security people in operations are not really at fault (it's not their broken software), and neither are the software people (they're not security people). Organizationally, this is a textbook management problem.

In an “utopic” environment, software security should be everyone’s job. In a more realistic one, setting up a team that will be in charge of the software security can make a big difference to solve the problem.

Network security specialists are not the correct option, even if they know a lot about how software operations work, usually they don’t know enough about software itself to make a good software security practitioner. Put simple, they lack knowledge about compilers, language frameworks, software architecture, testing, and several other things necessary to be a solid software person.

The correct option in this situation is to use experienced software developers to put them at the front of a security team. Security is easier to learn than software development, and a person already being a software developer will have more acceptance among the other developers, as they tend only to pay attention to fellow developers than rather an outsider who has little idea about software developing.
Hiring external security consultants to form people in security issues is an option than despite being expensive, is worth to do because it always pays off.

As open source applications usually can rely not only on its development team but also on an interested community of people that are going to make use of it, and the code is available to everyone, another suitable option could be that people from that community with enough security interest and background took care of the security evaluation in a free and collaborative way.
If many people work on the security of an application individually, security issues that maybe someone will not find out, will be found by others, making this alternative also an interesting one.

4.2.2 Setting out the Testing Plan

A test plan can be as large as several hundred pages or as simple as a single piece of paper.

Doing a really documented test plan can be good as it contains a comprehensive analysis of the system and it can be really helpful, but it can also be the most consuming time task in the project, slowing down the process and getting obsolete quite fast as keeping up-to-date with a really huge test plan requires a lot of efforts.

As Steven Splaine says [17], the testing team should decide at what level of detail a test plan ceases to be an aid and starts to become a net drag on the project's productivity.

The security team should be able also to change the testing plan in light of newly discovered information such as previous test results that can reveal some

other problems, allowing this way the testing effort to focus on more important and in-need-of-attention areas.

One should see the testing effort as an iterative task in which the last iterations are not defined at all, and they will get defined once the results of previous iterations will appear, making this way the effort more effective.

4.2.3 Keeping up with the testing effort: Retesting

Many security practitioners make really good test plans, but forget about a really important part of the testing effort, the retesting.
Applications require continuous testing when they go live, because of the frequent changes that happens in applications and the technology surrounding them, and because of the often discovery of new vulnerabilities.

As Steven Splaine says [17], even if the functional requirements of a system remain unchanged, a system that was deemed secure last week may become insecure next week.

This can happen because of an unknown exploit becomes suddenly know to the attacker community, because additional devices are added to the system, but can be miss configured, because of installing a patch or a service pack that might change settings or open new security holes, because passwords maybe expiring and being reseted to shorter and easier ones, or even because such a trivial thing as people getting not sensitive to security alerts due to the high amount of false alarms.
It is when this kind of events occurs that retesting should be carried out.

Security testing should not be regarded as a one-time event, but rather as a recurring activity that will keep going on as long as the system remains active.

4.2.4 What to Retest

The security team should consider each test that has been used previously during the application’s initial testing effort as a possible candidate for

conforming the set of tests that should be carried on in a regular basis after the system goes live.

"Regression tests are usually intended to be executed many times and are designed to conform that previously identified defects have been fixed and stay fixed, that functionality that should not have changed has indeed remained unaffected by any other changes to the system, or both". - Steven Splaine

The determination about if to repeat or not certain test depends really much on how the previous problems that the test revealed were fixed and in the likelihood of that problems reappearing.

4.3 Design Guidelines

After describing the organizational guidelines, we go on into a new section, the design guidelines.

This part of the guideline focuses on the security efforts that should be done during the design stage of the application.

This includes which programming language to use, which distributed object platform to choose if needed, which operating system to use, discusses about choosing to develop the application as an open source project or as a closed source one and the advantages or disadvantages that it involves and it includes also a crucial part of the software security, the risk management stage.

The decision of including the risk assessment in this section has been taken due to the convenience that risk assessment should be done in the earliest stage of the product development that is possible, and the design stage is the most appropriate one.

4.3.1 Choosing a Programming Language

As Viega and McGraw say [10], the single most important technology choice most software projects face is which programming language or set of languages to use for implementation.

The choosing of a specific language can be determined by several factors or requirements like needing efficiency or representation power.
This ends up usually in choosing C or C++ because of efficiency requirements, not considering that other languages can be more secure for implementing the application and still to meet the efficiency requirements.

Other big reason for choosing certain languages is based on the comfort or familiarity that the software developer has with that language.
But making those choices usually indicates really a poor view and not much worries about security, and not much people takes a while to think about the benefits that using other languages can bring to the project.

One of the biggest mistakes companies make when choosing the programming language for a product is not to consider the impact that the language will have over the software security.

This section doesn’t intends to give the definitive answer about which language to use, first of all because that could be considered advertising a specific language, secondly because security of a programming language depends a lot on how the language is used (i.e. some software developer can use a language with built-in security but due to the lack of knowledge not to use the feature, and other software developer can perhaps use a “less secure” language but due to great knowledge on it and being really careful, to develop a more secure application) and also because the aim of this section is to give a general advice about what to look at when choosing the programming language to develop an application.

Of course, if having good knowledge about several programming languages and knowing how to take profit of their security features, the wiser choice would be to pick one with built-in security, exception handling and that sort of features

that when used properly would make your application more secure, like Java or C#, in detriment of C, C++ or others that doesn't provide that features.
For a more detailed language comparison, you can take a look in [18].

In conclusion, when it comes to choose a language to develop an application it is important to think carefully which one offers better security features, but as important as this is also to know how to use properly the language selected and not to be less careful when the language implements aids like exception handling because of thinking that "the language will take care about it for me".

4.3.2 Choosing a Distributed Object Platform

As Viega and McGraw say [10], these days, client-server applications are being constructed with software systems based on distributed objects, such as CORBA, DCOM or EJB using RMI.
These technologies provide with remote availability of resources, redundancy and parallelism with much less effort than old-fashioned programming.

When thinking about software security, each of these technologies has different security features that should be considered when choosing which Distributed Object Platform to use.
The aim of this section is not to point out which one specifically to choose, as each application has different needs, but to make clear in which way each of these technologies provide security.

4.3.2.1 CORBA

CORBA implementations can come with security service or not, it greatly varies from one implementation to another, having the worst case, you will not have any security implemented, you may have security only half the way in some implementations, or to find an implementation with full security features implemented. Of course the recommendation is to get a fully secure implementation.

CORBA defines two levels of security, level 1 is intended for applications that need to be secure but where the code itself doesn't need to be aware of security issues, and level 2 that supports more advanced security features.
CORBA allows as the most significant feature secure communications using cryptography. CORBA also provides authentication services, which can be made transparent to the application.

Access to particular operations, such as the ones of an administrative interface for example, can be restricted in CORBA, and it has a wide set of possibilities when managing privileges in a distributed system, allowing great flexibility on what an object can do with those privileges.

4.3.2.2 DCOM

DCOM is the equivalent of CORBA but only working in Microsoft platforms.
Security features in DCOM are quite similar to CORBA ones even if they look totally different.

Authentication, data integrity, and secrecy are all wrapped up into a single property called the authentication level. Authentication levels only apply to server objects, and each object can have its own level set.

Authentication level range goes from 1 to 7, being 1 the lowest (no authentication or security at all), and 7 the highest security level (called packet privacy-level authentication, which fully encrypts all data and authenticates each packet). The range left in the middle of levels 1 and 7 increases step by step security until reaching level 7.
It is not the purpose of this guideline to describe each level on detail but only to give a general vision, the advice is to consult Microsoft support page [19] for a better and deeper view of those levels.

4.3.2.3 EJB and RMI

As Viega and McGraw say in [10], Enterprise Java Beans (EJB) are Java's version of a distributed object platform. EJB client/server systems make use of Java's Remote Method Invocation (RMI) implementations for communication.

While the EJB specification only provides for access control, most implementations usually provide encryption facilities that are configurable from the server environment.

EJB's access control system consists in moving access control decisions into the domain of the person assembling the application. That way someone not associated with development can specify the policy.

A critical security issue to note is that EJB implementations are built on top of RMI, and may inherit any problems associated with RMI implementations.
In the past, RMI had a poor reputation for security because RMI was configured to allow clients to download required code automatically from the server when it wasn't present. This feature was generally an all-or-nothing toggle and the negotiation was possible before a secure connection had been established.
It is for this reason that Viega and McGraw don't recommend in their book [10], the use of EJB and RMI at all.

But security in RMI has evolved as you can check in java.net Krishnan Viswanath's article [20], and these issues have been corrected, making it a totally valid choice as well as the rest of previously described technologies.

4.3.3 Choosing an Operating System

In this section, I will provide some advice about what to think on when choosing an operating system.
Yet again, it will not point out a specific operating system, because as before it happened with the programming language section, it is not only about the selected operating system but also a lot about how the selected system is configured and administrated.

As CERT® Coordination Center says [21], if you have knowledgeable staff, you may choose to use freely available OS versions so that you can maintain or fine tune the product to meet specific requirements. You might have more confidence in the modified OS because you were responsible for making changes or closely involved in the implementation of patches or workarounds. If you know about a vulnerability and understand the problem, you may want to

apply fixes immediately to the source code rather than wait for an upgrade or patch to be released through other channels.
If you select freely available OS versions and don't have the resources to maintain software in-house, it's important to know that you could be placing your system at a high risk of compromise.

If you do not have the time or expertise to modify and maintain an operating system in-house, you might choose a commercial vendor product. When you buy a commercial operating system, you can purchase a service contract to provide you with patches, upgrades, and other customer assistance. Alternatively, you could buy third-party service or select products from vendors who implement fixes and make patches publicly available.
When choosing an operating system, there are many things you need to consider like availability of source code vs. binaries, availability of technical expertise (internal and external), maintenance and/or customer support, customer requirements and usability and cost of software, hardware, and technical support staff.

No matter the choice you make, you should first carefully review and understand the needs of your organization or customer base in terms of resources, cost, and security risk. The best advice is to compare the available products and services to your needs, and then determine what product best matches your needs.

4.3.4 Closed Source vs. Open Source

“Given enough eyeballs, all bugs are shallow”. – Eric Raymond

The point about choosing to develop an application as open or closed source has been discussed a lot.
The truth is, however, that once again not any of the two approaches is definitely more secure and that secure applications can only be developed with good security practices and with being careful when developing it.

In one side, one might think that developing as open source makes applications more safe, as their code is reviewed by many people making this way easier to

discover possible vulnerabilities and to fix them in less time, making more difficult for the “bad guys” to exploit them, and also because if you understand the code, you can even fix the vulnerability by yourself without having to wait for a patch release.
But this fact is true also the other way around, as open source application’s code is available to everyone, it is also easier for the malicious user to discover vulnerabilities and thus, exploiting them in an easier way.

In the other side, closed source applications rely in what is called “security by obscurity”, which is a double edged weapon.
It is true that for a malicious user, is far more difficult to find a vulnerability without being able to take a look into the code, and it also makes difficult to develop an exploit for it, as a good understanding of what the code is doing is necessary to do that.
But it is also true that it is possible to use reverse engineering to discover how the code is working, and there is also the problem that once the vulnerability is found, users of the application are totally unprotected against attacks exploiting that vulnerability until the vendor of the application releases the fix for it.
Also, relying in security by obscurity might lead to mistakes like implementing self-made security solutions thinking that as no one knows how they work, no one will be able to break them. But that doesn’t mean that they are more secure that an already implemented one and well known, if it has proven to be secure already.
Put simple, it would not be clever to develop a cryptographic algorithm by yourself for an application, because even if no one knows how it works, it is likely that it will be not as safe as some of the already implemented algorithms like DES, as good cryptographic algorithms remain good even if people know how they work.

As Jason Miller says [22], although you can take an open source project, compare it against a closed source project, and say that one is more secure than the other based on some number of observations or measurements, this determination will probably be based on factors other than the nature of the project's open or closed source code.
Secure design, source code auditing, quality developers, design process, and other factors, all play into the security of a project, and none of these are directly related to a project being open or closed source.

Neither closed source nor open source is inherently more or less secure than the other, making a blanket statement such as that would be foolish. The best conclusion that one can make on this subject is that the two methodologies are not "better or worse", but instead, different from one another. I don't believe that you can answer that question any better.

4.3.5 Risk Management

As said in “A Risk Management Guide for Information Technology Systems” [13], Risk Management is the process that allows IT managers to balance the operational and economic costs of protective measures and achieve gains in mission capability by protecting the IT systems and data that support their organizations’ missions.

Risk management involves three stages: risk assessment, risk mitigation, and evaluation and assessment.

4.3.5.1 Risk Assessment

Risk assessment is the first process in the risk management methodology. Organizations use risk assessment to determine the extent of the potential threat and the risk associated with a project.
The output of this process helps to identify appropriate controls for reducing or eliminating risk during the risk mitigation process.

Risk assessment can be divided into several steps, although some of them can be carried out concurrently.

First step is used for system characterization. Characterizing a system establishes the scope of the risk assessment effort, delineates the operational authorization (or accreditation) boundaries, and provides information (e.g., hardware, software, system connectivity, and responsible division or support personnel) essential to define the risk.
The gathering of this information can be done through questionnaires, interviews to personnel and/or project’s documentation review.

After this step, the next one is to determine the possible threats, vulnerabilities and its sources by developing a list of system vulnerabilities (flaws or weaknesses) that could be exploited by the potential threat-sources.
Recommended methods for identifying system vulnerabilities are the use of vulnerability sources, the performance of system security testing, and the development of a security requirements checklist.

Next to this step comes to set the likelihood of that threat sources to exploit the vulnerabilities.
To get a global rating that indicates the probability that a potential vulnerability may be exploited by the correspondent threat source the following factors must be considered:

• Threat source motivation and capability
• Nature of the vulnerability
• Existence and effectiveness of current controls.

After considering these factors, a severity level rated from low to high will be assigned to the vulnerability.

The next major step in measuring level of risk is to determine the adverse impact resulting from a successful threat source exploiting a vulnerability.
This impact can be described in terms of loss or degradation of any, or a combination of any, of the following three security goals: integrity, availability, and confidentiality.

Last step of the risk assessment is to develop control recommendations that could mitigate or eliminate the identified risks, as appropriate to the organization's operations, are provided. The goal of the recommended controls is to reduce the level of risk to the system and its data to an acceptable level.
The control recommendations are the results of the risk assessment process and provide input to the risk mitigation process, during which the recommended procedural and technical security controls are evaluated, prioritized, and implemented.
As not all the possible controls can be implemented to reduce loss, a cost/benefit analysis should be conducted for the proposed controls, to

determine which of the controls can be justified by the reduction in the level of risk.

4.3.5.2 Risk Mitigation

Risk mitigation involves prioritizing, evaluating, and implementing the appropriate risk-reducing controls recommended from the risk assessment process.

Risk mitigation is also divided into several steps, but the implementation of the previously mentioned security controls is the task of the application's development team, and to carry out the cost/benefit study is a task for the management section of the project, so I will not go into more deep in this section.

4.3.5.3 Evaluation and Assessment

In most projects, the application itself will continually be expanded and updated, and its surrounding technologies will maybe change from time to time. In addition, personnel changes will take place eventually and security policies are likely to change over time.
These changes mean that new risks will appear and risks previously mitigated can become a real risk again. That's why the risk management process is always in a loop that goes on during all the lifecycle of the application.

Risk management should be conducted and integrated in the lifecycle of the application because it is a good practice and supports the project's business objectives or mission.
There should be a specific schedule for risk management, but the process should also be flexible enough to allow changes such as major changes to policies and/or new technologies.

4.4 Implementation Guidelines

Once finished with the design guidelines, we move into the implementation stage guidelines

This part of the guideline focuses on the security efforts that should be done during the implementation stage of the application.

In this section I will talk about code review, using COTS (Commercial Off-The-Shelf) and the convenience of using and reusing trusted components.

4.4.1 Code Review

"Debugging is at least twice as hard as programming. If your code is as clever as you can possibly make it, then by definition you're not smart enough to debug it." - Brian Kernighan

Programmers make little mistakes all the time like a missing semicolon or placing an extra parenthesis. Most of the time, such mistakes have no more repercussion than the compiler noting the error and after the programmer fixing the code. After this, the implementation process continues and subsequent similar errors are fixed in the same way. This quick cycle of feedback and response doesn't appear to happen with most security vulnerabilities, which can remain hidden for a long time before someone actually discovering them. And the longer a vulnerability remains hidden, the more expensive is to fix it properly.

To solve this problem is when code review comes into action.
There are two ways of reviewing code, manually and with the aid of automated reviewing tools.

Manual code review is an extremely time-consuming activity, and to do it effectively one must first know how security vulnerabilities look before being able to start to manually review the code, not to mention that applications often have more than one hundred thousand lines of code, making the task virtually impossible to perform or requiring more than one person to perform the review.

As McGraw says in [16], static analysis tools compare favourably to manual audits because they're faster, which means they can evaluate programs much more frequently, and they encapsulate security knowledge in a way that doesn't require the tool operator to have the same level of security expertise as a human auditor.
Just as a programmer can rely on a compiler to enforce the finer points of language syntax consistently, the operator of a good static analysis tool can successfully apply that tool without being aware of the finer points of security bugs.

Automated code scanning tools go through the code, searching for common patterns that might cause a vulnerability to appear, comparing the code with some set of fixed rules that they have (i.e. like searching for the function strcpy() in c coded applications as it is well known that the use of this function can lead to buffer overflows easily), and after performing the analysis they present a list of results that include all that pieces of code that matched their rules.

Using an automated scanning tool, although, doesn’t means to run it and to have the work already done.
The output generated by an automated scanning tool stills need the manual verification of a human, although the work is quite smaller as the places to look at are already listed, saving you the time that takes to go through all the code.
Usually the output of these tools needs to be checked as they usually produce what is called false negatives, that mean that the tool is not perfect enough to discover every single vulnerability, and false positives, that mean that some times what is reported as a vulnerability, depending on the context of the application is not such vulnerability.
There's no way for any tool to know automatically which problems are more or less important to you, so there's no way to avoid looking through the output and deciding which issues should be fixed and which ones carry an acceptable level of risk.
Apart from this, no automated scanning tool can tell you design issues such as if password’s security enforcement is good enough or not, just to put an example.

In conclusion, both self-expertise and a good code scanning tool are needed.

4.4.2 Using COTS

Commercial off-the-shelf (COTS) defines the software products or technologies that are already made and available for sale, lease, or license to the general public. They are often used as alternatives to in-house developments.

The motivation for using COTS components is that they will reduce overall application development costs and involve less development time because the components can be bought or used for free (if they are open source), instead of being developed from scratch.

But using COTS can be a double edged weapon, because apart from the good features that using COTS bring like reducing costs and development time, they can also introduce in your application or system security vulnerabilities, as no one assures you that a particular COTS that you are using is vulnerability free.
When using COTS, the best measure is to check as much as possible if the component looks to be secure and if it has been in use for some time without presenting security issues, basically, to check if it can be trusted or not.

And this is what the next section is about, using and reusing trusted components.

4.4.3 Using and Reusing Trusted Components

When it comes to use components that have not been developed in-house, it is clever to make sure before using them that they have been tested by people you trust or with a solid security reputation, and that there haven't been published documents with relevant vulnerabilities about them.

As said in OWASP [4], the more successful a component has proven itself to be over time, the better reason to reuse it. Many people developing computer systems have gone through the same kind of problems, and may have invested large amounts of time researching and developing robust solutions to them. In many cases, components have been improved through an iterative process.

Using and reusing trusted components makes sense both from a resource stance and from a security viewpoint. When someone else has proven they got it right it is beneficial to take advantage of it.

Let's put as example the case of cryptographic libraries or algorithms. Well-used libraries and algorithms are much more likely to be more robust than something implemented in-house, because people are more likely to have noticed implementation problems.

4.5 Post Implementation Guidelines

To finish this guideline, I will proceed to present the post implementation guidelines.

These guidelines are to be applied after the implementation of the application, and they will test the security through penetration testing, which is a good method to check the security of the application in its real environment.
Also the importance of a correct configuration of the system will be discussed.

4.5.1 Penetration Testing

As said in OWASP [4], having tested the requirements, analyzed the design and performed code review, it might be assumed that all issues have been caught.

Hopefully, this is the case, but penetration testing the application after it has been deployed provides a last check to ensure that nothing has been missed.

Penetration testing can be divided in several steps which will be explained next.

4.5.1.1 Information Gathering

The first phase in penetration testing is focused on collecting as much information as possible about the application. Information gathering is a necessary step of a penetration test.

Using tools like search engines, scanners, sending simple HTTP requests, or specially crafted requests, it is possible to make the application to discover information by sending back error messages or revealing the versions and technologies used by the application.

4.5.1.2 Authentication Testing

Second step in penetration testing should be authentication testing. Testing the authentication schema means understanding how the authentication process works and using that information to circumvent the authentication mechanism.

First thing that can be tested here is if there are default user accounts that usually some technologies just create for the first login and that should be changed the first thing always, but sometimes can remain there just by mistake and pairs of users/password that can be guessable by dictionary testing.
After this, next step is to try to gain authentication by brute force methods. This step usually takes too much time and can make the application to block the rest of the attempts after some invalid tries.
Next to this, directory traversal testing should be used to try to find a way to bypass the application and gain access to system resources. Typically, these vulnerabilities are caused by misconfiguration.
Another possible test to perform is to check how the application manages the forgotten password functionality or if the “remember me” function is present, as this might mean that password is being stored somewhere or that a cookie is being stored to remember your session.

4.5.1.3 Session Management Testing

Session management covers all controls on a user from authentication to leaving the application.
Checking how an application handles session management and understanding it is important as it can reveal vulnerabilities that can provide a way to hijack a session or to make a replay session attack.

One of the ways to achieve this is by session token manipulation, which consists in modify a session token (i.e. hidden field, cookie or sessionID) so the session management system will think that we are a user we are not, thus being able to access in an illegitimate way the information that only that user should be able.

Another is to check if session token is transmitted from the client to the server in an encrypted transport by default. If that transmission is not done over a secure protocol like SSL or it is not encrypted before being sent, it can be exposed to eavesdropping.

4.5.1.4 Data Validation Testing

As said in OWASP Testing Guide [23], the most common application security weakness is the failure to properly validate input from the client or environment. This weakness leads to almost all of the major vulnerabilities in applications, such as interpreter injection, locale/Unicode attacks, file system attacks and buffer overflows.

The data must be validated by the application before it's trusted and processed. The goal of this step is to test if the application actually does what is meant to do and does not do what its not.

To perform this step, many tests can be done, including every existing injection vulnerability test, overflows or cross site scripting.

Several of these attacks have been explained in the theory part of the thesis and it remains out of the scope of this guideline to explain every single attack that can be carried out.

4.5.1.5 Denial of Service Testing

What a Denial of Service (DoS) attack is has been already explained in the theory part of the thesis

.

There are types of vulnerabilities within applications that can allow a malicious user to make certain functionality or sometimes the entire application server unavailable. These problems are caused by bugs in the application, often resulting from malicious or unexpected user input.

The first DoS case to consider involves a common defense to prevent brute-force attacks on user passwords. The most usual measure against this is to lock an account from use after between three to five failed attempts to login. This means that even if a legitimate user tries to provide his valid password, he will be unable to login to the system until his account has been unlocked. This defense mechanism can be turned into a DoS attack against an application if there is a way to predict valid login accounts.
Another way to achieve a DoS is to find some buffer overflow that will crash the application, thus making the service unavailable.
Also, if the application allows the user to determine how many of an object to create on the application server and it doesn't sets any upper limit, a malicious user can make the server to run out of memory.
A DoS case can appear also if the user can provide the input for a loop control variable, as he can make an infinite or high resource consuming loop to run.

Summing up, every user action that can make the application to crash or to consume high amounts of resources can result on a DoS case.

4.5.1.6 Web Services Testing

"By 2005 Web services shall have reopened over 70% of the attack paths against internet-connected systems, which were closed by network firewalls in the 1990's" – Gartner

The vulnerabilities in web services are similar to other vulnerabilities such as SQL injection, information disclosure and leakage but they also have unique XML/parser related vulnerabilities.
The first of these vulnerabilities involves XML not functioning properly when it's malformed. The XML message must be well formed in order to be successfully parsed. Malformed messages may cause unhandled exceptions to occur.
Other possible attack might be for an attacker to craft an XML document that contains malicious elements in order to compromise the target system. In this way, the attacker can include in the message the code needed to exploit an SQL injection in the web service, for instance.

4.5.2 Configuration Management

Checking security on the application is important, but it would not be worth anything if the system where application is running is missconfigured.
Configuration management consists in checking that everything is configured as it should be, so system security will be as higher as possible.

Among the actions that should be performed are turning off all features by default, as they are well known by attackers and can be used to gain illegitimate access to the system, always change default passwords in system's running applications as much of them come with default passwords that are well known, to encrypt network communications to prevent data leakage, to check in a regular basis the latest security vulnerabilities published and to apply the latest security patches.

Every system is unique and has different applications running, making impossible to provide a "general secure configuration".

5. Practical Application and Results of the Guideline

In this chapter, the previously developed guideline will be applied to an Open Source consumer application called Indivo.

The aim of this section is to demonstrate how to apply the guideline to an open source consumer application. The results obtained from the testing of the guideline with Indivo will be used to discuss the application's possible security issues and to test the guideline itself to check if it is working as expected.

The guideline is intended to be applied from the very first stage of development of an application.

As Indivo is an application that is already in its beta-testing stage and already under development and continuous change, some parts of the guideline can't be applied as intended, because some decisions have already been taken.

The guidelines previous to the implementation guidelines just will be discussed in terms of correctness on the decisions already made by the developing team of Indivo.

Indivo is an Open Source Consumer Application for the medical field developed by the Children's Hospital of Boston. Its main purpose is to manage data from electronic health records. Because of the nature of this data and the high degree of privacy and confidentiality that it requires, security is a crucial factor on this application.

5.1 General Guidelines in Indivo

The security guideline on this thesis starts with some general advice called General Guidelines.

The security guideline on this thesis is not only intended to actively test the security of an application but also to give advice about good practices while developing applications.

These general guidelines are nothing that you can directly apply, but some basic principles that developers have to keep in their minds during all the application development.

That's why they can't be directly applied to Indivo as the rest of the guidelines, and they will be skipped during the practical part.

5.2 Organizational Guidelines in Indivo

In this part, the organizational guidelines previously developed will be applied on Indivo.

5.2.1 Selecting the security professional

As starting point, I will address the issue of who should take care of the security on Indivo. Obviously, in this particular case, I will be the one testing the security of Indivo, as that is the purpose of the practical part of this thesis.
As I said in this part of the guideline, the most suitable person for taking care of the security of an application should be a quite experienced software developer with security formation, as in software security, good understanding of software development process is important.
As open source applications usually can rely not only on its development team but also on an interested community of people that are going to make use of it, and the code is available to everyone, another suitable option could be that people from that community with enough security interest and background took care of the security evaluation in a free and collaborative way.
If many people work on the security of an application individually, security issues that maybe someone will not find out, will be found by others, making this alternative also an interesting one.

In this particular case, my background as student of master on computer science makes me to match both cases, as I am not only experienced in software developing and interested in security, but also I form part of that "community" developed around Indivo as one of the aims of this thesis is to test security on it.

5.2.2 Setting the Test Plan

In this subsection of the guideline the basic testing plan has to be presented and developed.
This section is not intended to make penetration testing directly but to explain the basis and the steps that will be followed in the whole testing effort.
In the case of being able to perform various iterations on the testing effort, the test plan can be changed according to newly discovered security issues that remained hidden in the first test plan development.
In this particular case, only one iteration is going to be performed, so the test plan will be set up only once and will remain unchanged.

5.2.2.1 Introduction

The goal of this test plan is to check whether the Open Source Consumer Application called Indivo presents or not security issues that might be dangerous for the confidentiality and privacy of the data it manages, and therefore, solved.
In order to achieve this goal, risk analysis, code review and penetration testing will be performed.

5.2.2.2 Features to be tested

- Client-side application security
- Client-side to server-side application communication security
- Server-side application security
- Support technologies security
- System and configuration security

5.2.2.3 Test Approach

The first step will be to make a risk analysis in order to determine which features carry higher risk for the application and which lower, so the testing can focus more on those that can represent a higher risk.

After getting the risk analysis results, the second step will be performing code review. In order to do this, I will use a code review application for Java code called FindBugs, which will reveal which parts of the code can represent potential dangers for the security of the application.

Last step will be to perform penetration testing. Penetration testing will be divided on two parts, one focused exclusively over Indivo, and the second focused on finding vulnerabilities in the surrounding technologies that Indivo needs to run.
The penetration testing involves:

On Indivo

Use a proxy (WebScarab - OWASP) to check for possible vulnerabilities as unsafe session IDs, trying to modify requests to server before they leave the client, modifying parameters sent to server, changing hidden fields, etc.

Perform checks to search for possible SQL or XML injections (both in the BerkeleyDB database and the MySQL database), perform checks for Cross Site Scripting and check for directory transversal vulnerabilities that could make an attacker to access files that he is not intended.

Use a sniffer or a packet capture tool to intercept communication between client and server to check if relevant data can be disclosed.

Use brute forcing and dictionary based attacks on the logging system to check if it's possible to repeatedly login with the same username and to break into the application.

On the surrounding technologies

Check vulnerability tracking lists such as CERT, Secunia, Securityfocus, etc, to find possible issues and vulnerabilities with the surrounding technologies that Indivo uses.
As Indivo needs certain versions of those technologies and most of them are not last versions, it might be possible to exploit old vulnerabilities on those products.

Check for vulnerabilities on following products:

- Apache Tomcat 5.5
- Apache 2.0
- Java 1.5
- PHP 5.1.2
- Java Bridge 3.0.7
- MySQL Server 5.0

5.2.3 When and what to Retest

If testing is an important part of the security practice, retesting has at least the same importance.
Something that has been tested and proved to be safe can become unsafe suddenly because a lot of factors including a change in the configuration or operating system, a new patch for the application, a new vulnerability that suddenly appears and before no one knew of, etc.

As Indivo is on beta testing at the moment, and a lot of things are still being implemented or recoded and a lot of functionality being added, retesting of Indivo should happen each time that a new version is released or major relevant changes are made. Put simple, at this stage of development, retesting should be performed each time that the application's development cycle starts a new iteration. This way, the accumulation of possible vulnerabilities is avoided, which is unlike to happen if retesting only happens when final product would be released.
In this stage, in my opinion everything included in the original test plan should be retested without exception and the newly found issues should be added to the original test plan, as the application is under constant implementation changes and that changes can affect not only a single part of the application but the whole application.

After Indivo will be already in the production stage (it will be fully operative and not in beta stage), retesting should happen with a fixed periodicity and should be more focused on finding out and testing the new vulnerabilities that have appeared on vulnerabilities tracking lists since last tests and in some important

security issues such as password's strength, and proper configuration of the system and the application. Also retesting should be carried out every time a new update or patch is released.

5.3 Design Guidelines in Indivo

The design guidelines are to be applied in the design stage of the application. As Indivo is already out of this stage because it is already in beta testing and implementation, the guidelines will not be applied as supposed but will be used to compare them with the decisions already taken in Indivo and to determine the correctness of those decisions.

5.3.1 Choosing the Programming Language

Choosing a programming language to develop an application is one of the most important decisions during the design stage.
Indivo server has been developed in Java, and that has several advantages from a security point of view.

Java provides the possibility of restricting application's resource access by the use of security policies.
Java provides mechanisms to enforce strong memory protection. These mechanisms remove the possibility of either maliciously or inadvertently reading or corrupting memory locations outside boundaries of the program. As a result, applications cannot gain unauthorized memory access to read or change contents.
Java provides full support for encryption and digital signatures, making use of powerful encryption technology.
Java provides good rule enforcement as it is completely object-based. By using objects and classes to represent corporate information entities, it is possible to explicitly state the rules governing the use of such objects.
Java does a fairly reasonable job making programmers to catch the errors and exceptions that might be thrown, mitigating this way unhandled exceptions that might make the application to crash.

Apart from all the security features, Java code is fully portable to different operating systems, making the choice of using Java for Indivo even more recommendable.

As for Indivo client interface, it has been developed in PHP. Maybe the most natural option would have been developing the interface in jsp, as the rest of Indivo is coded under Java.

PHP is a language that has not that many security features as jsp, but in the other hand is a really popular web development language and it is in continuous upgrade.

5.3.2 Choosing a Distributed Object Platform

Indivo uses Java RMI to encapsulate the communication between the server and the IO Store.

The option here is simple; there are only two distributed object platforms suitable in this case, Java RMI and CORBA.

As Indivo is already implemented using Java, the most natural choice in my point of view is to use Java RMI as distributed object platform.

In terms of security, both platforms provide good security measures that despite being implemented using different mechanisms, accomplish their goal in a proper way.

But it is more likely that using Java RMI, as the development team is already used to work under Java, and RMI is simpler to work with, since the developer does not need to be familiar with the Interface Definition Language of CORBA, fewer mistakes will be made during the implementation, thus reducing this way the possibility of implementation flaws.

5.3.3 Choosing an operating System

Indivo is designed to run under whatever operating system that can handle Java and Apache Tomcat and Apache Server, which, as far as I know every Operating System can.

I will not evaluate this point as to choose operating system is quite an open choice and the evaluation would be the same as the advice I gave on the guideline part. The wiser decision is to choose an operating system that fits the

needs of your company and that you have enough resources and experience to maintain and to keep fully under control.

5.3.4 Risk Analysis

The purpose of this risk analysis is to get a clear view of the potential threats/vulnerabilities Indivo can be exposed to, to rate them in a severity scale with a range from low to high impact and to propose a possible solution to minimize them or to make them disappear.

5.3.4.1 System Characterization

First step is to get the system characterization to establish in a better way the scope of the risk analysis.

Hardware

Acer Laptop with processor IntelCoreDuo@1,66MHz, 1 Gb of RAM. Direct connection to Internet through a Broadcom 10/100 net card and wireless card also operative.

Software

Windows XP Professional edition Service Pack 2, with Norton Protection (Antivirus + Firewall)
Indivo version 3.0 beta running under Apache Tomcat 5.5 and Apache Server 2.0, with Java 1.5, PHP 5.1.2, Java Bridge 3.0.7 and MySQL Server 5.0.

5.3.4.2 Vulnerability list

Second step is to get a list of possible vulnerabilities.

- Session ID’s might be guessable.
- Indivo should work over https but also works over http.

- Possibility of accessing server side files like .xml's files can disclose important information.
- Error messages can disclose internal database structure or server way of operating (product versions, other technologies server is using).
- Indivo uses surrounding technologies (PHP, Apache, Java, etc) that only can be exact versions (i.e. only PHP 5.1.2 can be used, not other).
- Possible attacks like SQL injection or XSS might be possible.
- Unsafe, unused or test code might be found among Indivo source code.
- Password complexity is not enforced, asking only for up to 5 or more character passwords.

5.3.4.3 Vulnerability severity rating, adverse impacts and control recommendations.

In this last step, I will present the severity rating of the vulnerabilities, its possible adverse impact and control recommendations to mitigate or eliminate the possible vulnerability.

Rating should be done also by risk probability, because risk is **severity * probability**.
Unfortunately to know the probability of a certain risk, more than one iteration is required, because it is not easy to measure the probability before effectively trying to exploit the vulnerability.
As the scope of this thesis involves only one iteration, only a severity rating will be done.

Threat	Effect	Possible Solution	Severity
Session ID's might be guessable	Possible risk of session hijacking	Session ID's should be unique, never reused, randomly generated and nearly impossible to guess from an obtained sequence of previous ID's	High
Indivo should work over https but also works over http	Transmission of data in plain text with the possibility of obtaining username/password	Use https protocol always to provide encrypted communication	High

	pairs by a sniffer or by a in-the-middle proxy		
Possibility of accessing server side files like .xml's files can disclose important information	Attacker can get important information about how the Indivo server operates	Keep server side important and not-relevant-for-anyone-else files inaccessible to other than the server machine	Medium
Error messages can disclose internal database structure or server way of operating (product versions, other technologies server is using)	Attacker can get extra entry points with the newly got information	Error messages should be totally neutral and not to reveal any important information	Medium
Indivo uses surrounding technologies (PHP, Apache, Java, etc) that only can be exact versions (i.e. only PHP 5.1.2 can be used, not other).	Being unable to update to more recent versions of this products might end up in well known vulnerabilities being used to attack the server.	Implement Indivo in a way that is compatible with new versions of the same technologies	High
Possible attacks like SQL injection or XSS might be possible	Attacker can maybe fake other user's identity or make other users to execute some unwanted actions	Check if Indivo is vulnerable to this attacks	High
Unsafe, unused or test code might be found among Indivo source code	Attacker can check the source code as it is an open source project which code is available to everyone, and exploit some vulnerability in the code.	Perform a source code review in order to check for possible unsafe or repeated code.	Medium
Password complexity is not enforced, asking only for up to 5 or more character passwords	Passwords might be easy to brute force, leading to unwanted access.	Ask not only for a minimum length (higher than 8 characters) but also for combinations of letters, numbers and special characters.	Medium

Table 1: Vulnerability severity ranking, impact and control for Indivo.

5.4 Implementation Guidelines in Indivo

In this part of the security guideline I will evaluate the correctness of the source code on Indivo server with the help of a source code review tool for Java code called FindBugs, and the COTS used on Indivo.

5.4.1 Code Review

To review the source code of Indivo server I have used an automated source code reviewing tool for Java called FindBugs. The usage of the tool is really easy and intuitive. The first step was defining a name for the project and to add both the jars that were part of Indivo and the source code files (.java) corresponding to that jars. In the next image you can see how I did this.

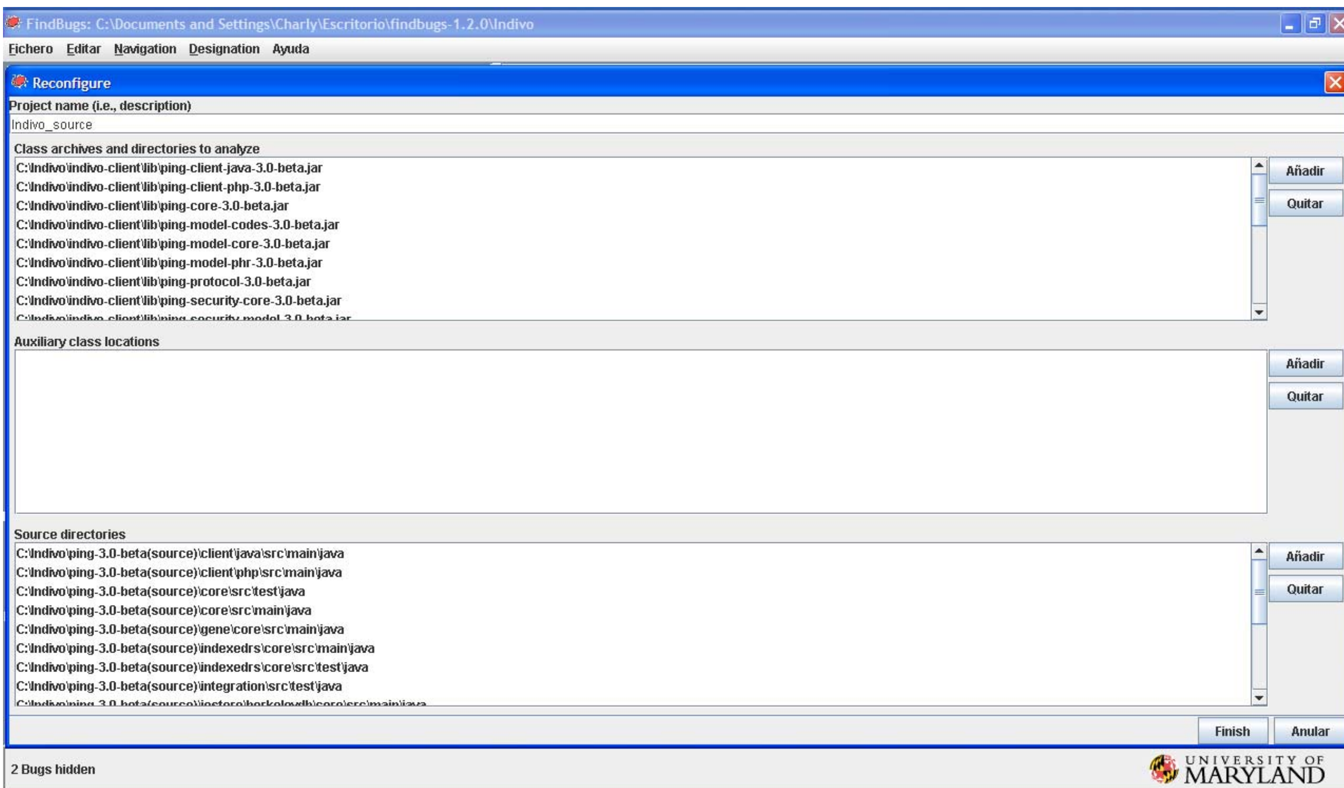

Figure 12: Configuration for FindBugs

After doing this, FindBugs proceeds to analyze all the classes inside the jars in order to find the relevant bugs and to match those bugs with the corresponding lines that match the bugs in the source code files.

Once this is done, a result screen with the bugs sorted by kind of bug is presented. When clicking on a specific bug, FindBugs shows the information about the bug, some advice on how to correct it and the source code lines where the bug was found highlighted. The results of Indivo source code review can be seen in the next image as they are presented by FindBugs.

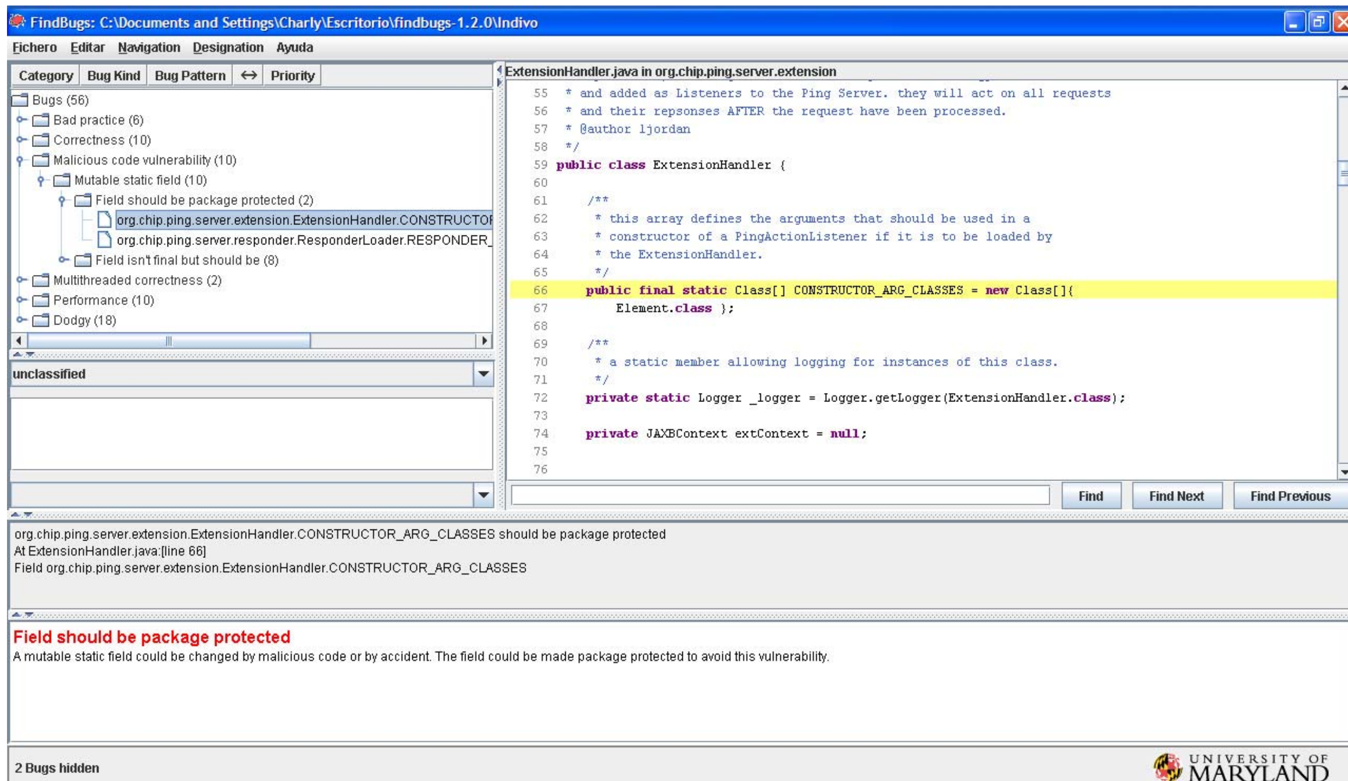

Figure 13: FindBugs result report.

After getting the results, I checked manually all the reported bugs, taking out the bugs that the program suggested that could be false positives or that were not security relevant but only performance relevant.

Following, I will comment briefly the most important bugs found after the code review.

Bad Practice

Under the category of dropped or ignored exception FindBugs gives to results of methods that ignore exceptions, thus making possible that an uncaught exception might appear making the application to crash.

Correctness

Under the category of infinite loops, FindBugs reports an infinite recursive loop that might cause a stack overflow with the risk of the program crashing.
Under the category of null pointer dereference FindBugs reports five results that might make the program to cause a NullPointerException making it to crash. Two of these results might be false positives.

Malicious Code Vulnerability

Under this section FindBugs reports eight possible fields that are not final but should be. A mutable static field that is not final could be changed by malicious code or by accident from another package. The field should be made final to avoid this vulnerability.

5.4.2 Commercial Off-the-Shelf

In this section, the security of the COTS that Indivo uses will be discussed, so a better understanding can be achieved.

The first COTS that I will analyze is the database system that Indivo store uses. This is an open source technology called BerkeleyDB and it has been developed by SleepyCat.
Berkeley DB is notable for having a simple architecture. Programs access the database using in-process API calls. It does not support SQL or any other query language, nor does it support table schema or table columns. A program accessing the database is free to decide how the data is to be stored in a record; BerkeleyDB puts no constraints on the record's data.
Choosing BerkeleyDB as main data storage has several advantages and it is a technology that has proven to be really safe. BerkeleyDB has not been reported any known vulnerability or security issue, and as it is accessed by API calls and doesn’t supports any query language, it is immune to injections.

Indivo uses Xajax 0.2.4 implementation for integrating AJAX on the PHP interface. Xajax is an open source PHP class library that allows you to easily create powerful, web-based, AJAX applications using HTML, CSS, JavaScript,

and PHP. Applications developed with Xajax can asynchronously call server-side PHP functions and update content without reloading the page.
Xajax 0.2.4 version was recently reported with one Cross-site Scripting vulnerability with bugtraq ID 24006.
Xajax is prone to an unspecified cross-site scripting vulnerability because the application fails to properly sanitize user-supplied input.
An attacker may leverage this issue to execute arbitrary script code in the browser of an unsuspecting user in the context of the affected site. This may help the attacker steal cookie-based authentication credentials and launch other attacks. The proposed solution is to upgrade to Xajax 0.2.5.

Indivo uses Log4j implementation for logging events at runtime. With log4j it is possible to enable logging at runtime without modifying the application binary. The log4j package is designed so that these statements can remain in shipped code without incurring a heavy performance cost. Logging behavior can be controlled by editing a configuration file, without touching the application binary.
No vulnerabilities for Log4j have been reported.

Indivo uses Sun XACML implementation for management of access policies.
XACML is an XML-based language for access control that has been standardized in OASIS. XACML describes both an access control policy language and a request/response language. The policy language is used to express access control policies (who can do what when). The request/response language expresses queries about whether a particular access should be allowed (requests) and describes answers to those queries (responses).
No vulnerabilities for Sun XACML have been reported.

Indivo uses various Apache commons-framework packages, like commons-logging, commons-beanutils, commons-collections, etc.
No vulnerabilities for these packages have been reported.

Indivo uses the Avalon framework included now in Apache Excalibur project.
The Avalon Framework consists of interfaces that define relationships between commonly used application components, best-of-practice pattern enforcements, and several lightweight convenience implementations of the generic components. No vulnerabilities for the Avalon framework have been reported.

5.5 Post Implementation Guidelines in Indivo

In this section penetration testing will be performed against Indivo as stated in the Test Plan design in the organizational guidelines.

5.5.1 Penetration Testing

Penetration testing will be divided into two sections:

- On Indivo
- On the surrounding technologies of Indivo

5.5.1.1 Testing Indivo

First step on penetration testing is to gather as much information as possible about the application. Of course, in this concrete case all the information is already known, but I will perform some tests to gather information about the server and applications working on it anyways. It is always a good way to see which information actually an attacker can get.

First of all, I will proceed to fingerprint the server. To fingerprint the server means to get information about the kind of server and the version. This can be achieved by sending an http request directly and looking at the response headers, or by the use of automated tools.

I will use a tool called HttpPrint to gather information. The result can be seen in the following image.

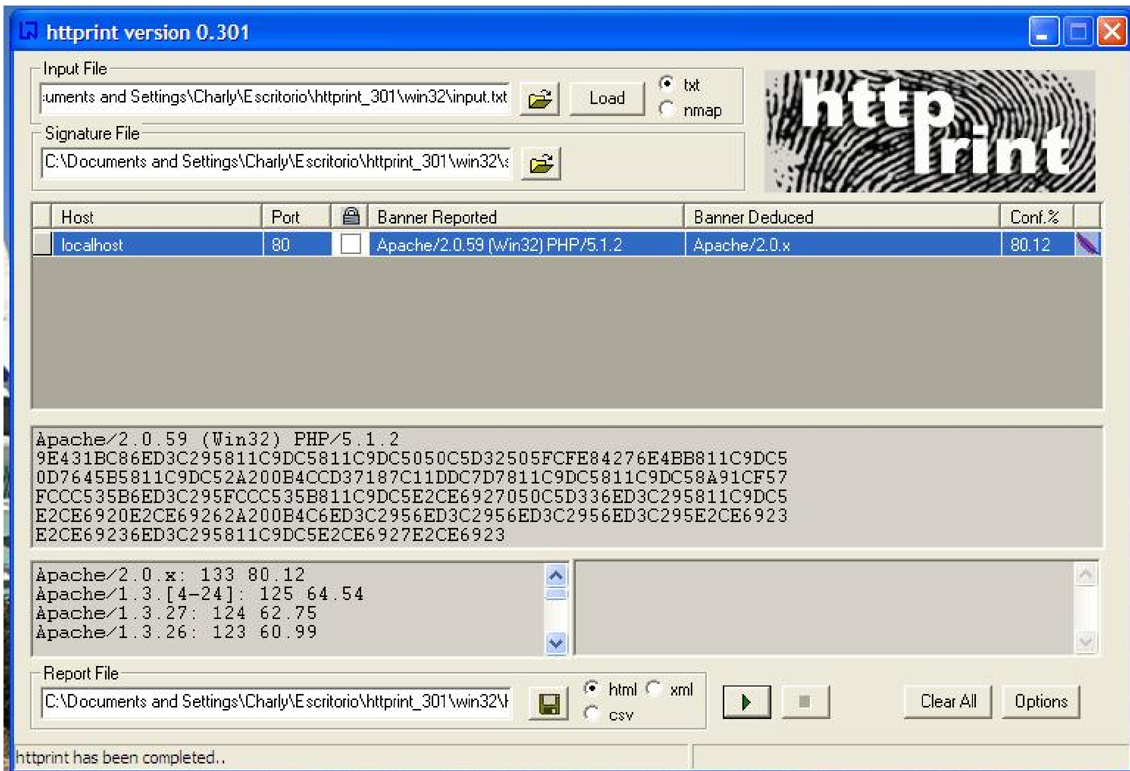

Figure 14: Server Fingerprint.

As can be seen in the image, the program gave as result that the server is an Apache server version 2.0.59 and moreover, that it is running PHP 5.1.2 and that it is running on a Windows system (Win 32).

This kind of information can provide an attacker of an entry point to our application by trying to exploit known vulnerabilities of the server version discovered.

By using a common port scanner, we can check for more applications running in the target machine. Using a regular port scanner, we obtain the following results.

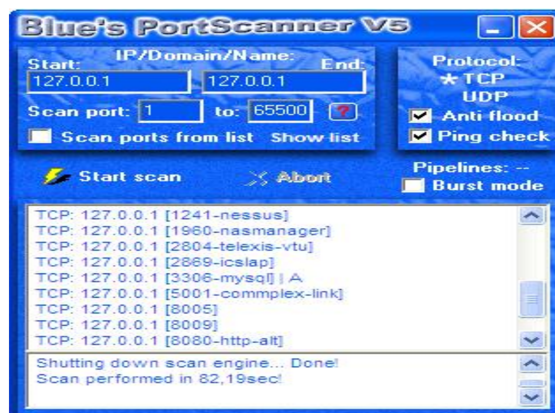

Figure 15: Port Scan results.

As it can be seen in the previous image, the scanner detects MySQL service running on port 3306 (default MySQL port), and an alternative http server on port 8080, that it is easy to assume it is Apache Tomcat, as it usually runs on that port.

Once this has been done, I will proceed to evaluate Indivo first using a proxy called WebScarab, developed by OWASP. A proxy is basically a program that acts as intermediate between the client and the server, allowing you to monitor all the traffic going from the client to the server and the other way round, letting you to modify that data either before being sent to the server or before reaching the client. It also has much other functionality like testing strength on session IDs or revealing all hidden fields of the client side application so you can freely modify them in order to obtain some undesired result.
In the following image, you can see an overview picture of WebScarab.

Figure 16: Web Scarab.

The first thing I will try is to analyze the strength of Session IDs because if session IDs are guessable in some way, security of Indivo could be compromised seriously.

To perform this task, I will use one of the functions of WebScarab, which permits to store several different session IDs and after that analyzing them and getting a statistic summary on their regularity and predictability. To feed WebScarab with enough IDs, I log in and out repeatedly forcing the session ID to regenerate and storing each ID in WebScarab.

After this process WebScarab perform an analysis and show the following results, captured in the following images.

Collection | Analysis | Visualisation

Session Identifier : localhost./PHPSESSID

Date	Value	Numeric	Difference
2007/05/22 17:59:37	19722405afc565e93dc333d92d553be7	45923888236852862919284885	
2007/05/22 18:03:46	178cb11f6c4e8a2b53947b3e6e81a9b1	43372566084618130343938038	-2551322152234732575346847
2007/05/22 18:07:30	c4dd49300b58f7f8594b727bfc9bd173	126281312182608224922061790	82908746097990094578123752
2007/05/22 18:09:40	fed9a1d7fa23be9ea0faf3e3c1bc185a	193134510124200061204476323	66853197941591836282414533
2007/05/22 18:11:36	ef91fe1b9ae4e778c7d1c4bc3f1437b4	166980473582796620465135321	-26154036541403440739341002
2007/05/22 18:12:40	23fd57254e6b358807b582a0718df7a9	66935339062353100552708149	-100045134520443519912427172
2007/05/22 18:29:49	82a8c719eaef7f08179aecd9277633c1	90232034257164254131211409	23296695194811153578503260
2007/05/22 18:30:46	01f68dc82e62128fa5b1480eb927002b	3146023880277280820838894	-87086010376886973310372515

Figure 17: Numerical analysis for session IDs.

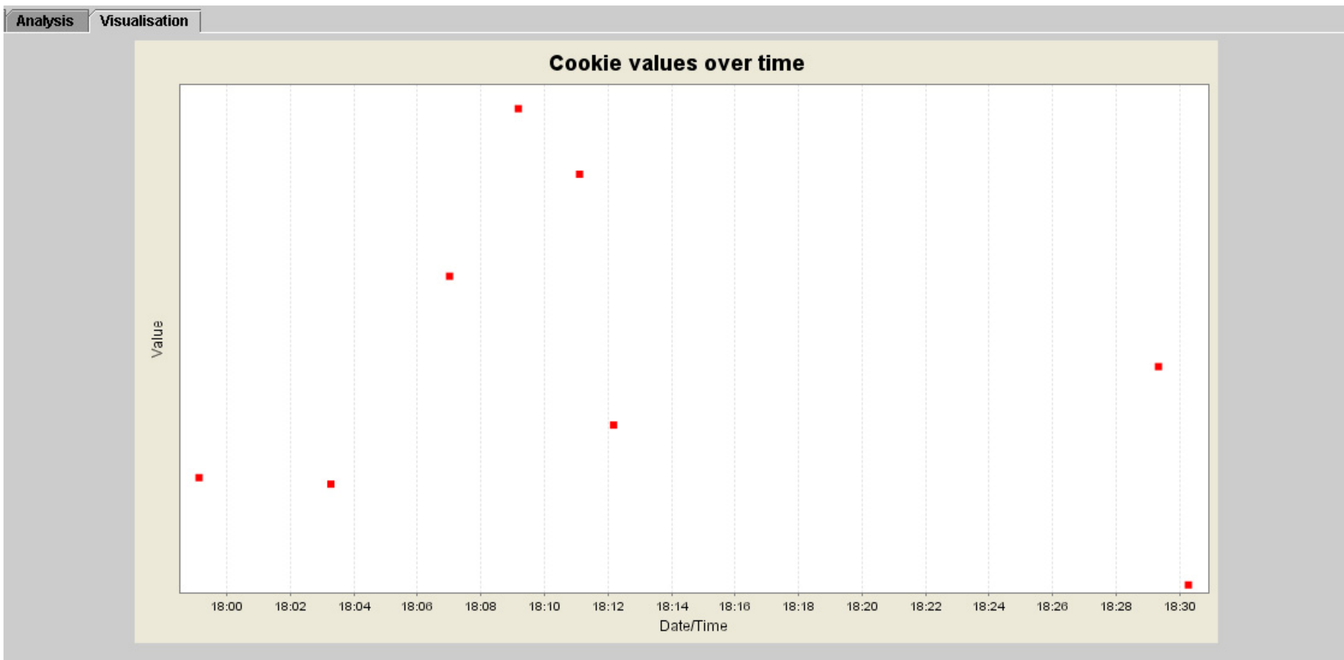

Figure 18: Distribution analysis for session IDs.

As it can be seen, Indivo session IDs are totally random and they don't follow any identifiable distribution pattern. This makes Indivo session IDs difficult to guess.

After analyzing session IDs, I will try to search for hidden fields in Indivo that can be changed in order to obtain some advantage. I will do this using WebScarab tool to automatically reveal hidden fields.
After trying different hidden fields found in “Edit Personal information” template, “Add annotation” template and others, no relevant hidden field that could be exploited was found.

After searching for hidden fields, next step to take will be to try to modify requests sent to Indivo server. Using WebScarab I will try to change those requests and the methods they ask for, to see if a regular user can access methods that he is not intended to use.
In order to find methods that a regular user is not able to use, I log in as administrator of the system and I try to create a new user, to see the method that is invoked. This method results to be “showEnrollmentForm”.
After this I try to modify a simple request to the server from a user of the group patients, and to add the previous method.

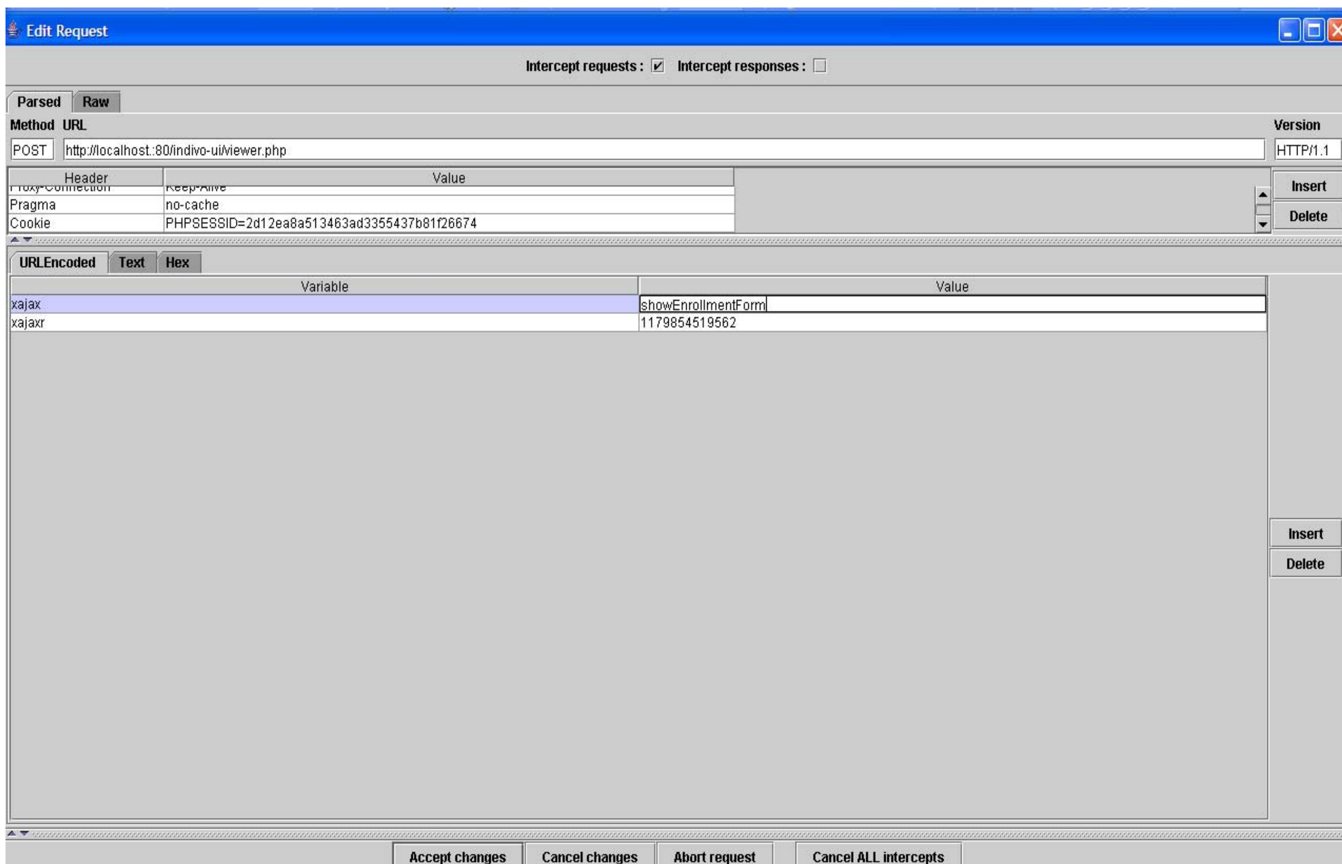

Figure 19: Changing an Indivo request.

The result of doing this is a regular user being able to access a private administrative interface, as shown in the following image.

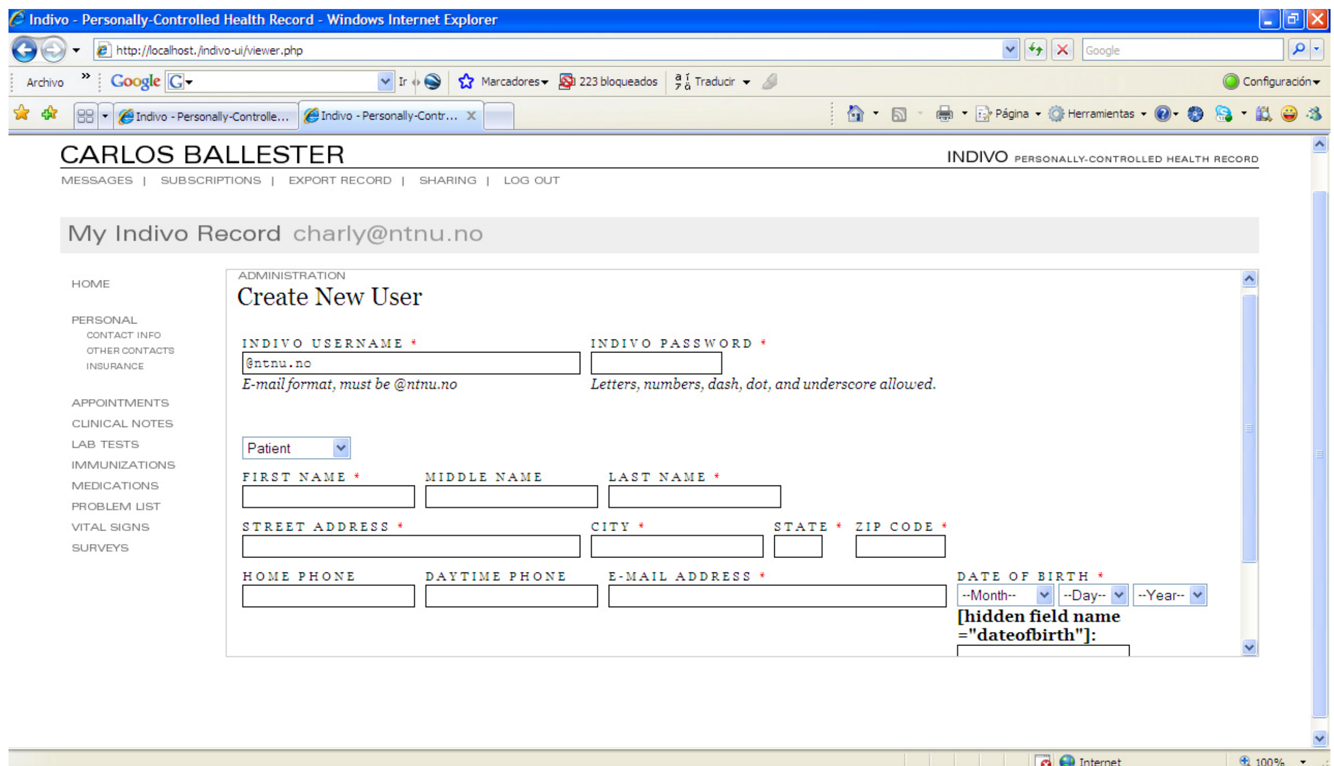

Figure 20: Accessing an administrative interface.

After doing this, I try to create a new user, but the access policy doesn't allow me to do this. Anyway, being able to access an administrative interface as a regular user is not safe in terms of application security, and administrative interfaces should be protected.

After this step, I will try using WebScarab to send malformed requests to the server in purpose to cause errors and to see if errors can disclose important information. The result of this action ends up in what can be seen in the next image.

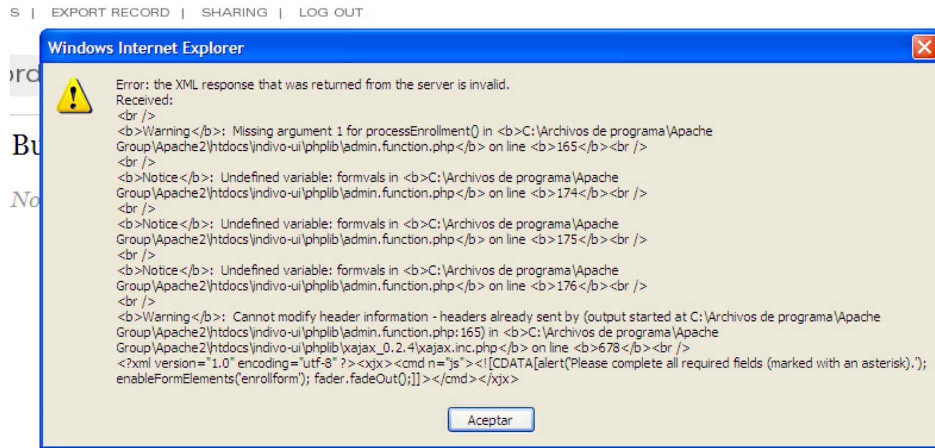

Figure 21: Indivo Error message.

As can be seen in the error message, Indivo reveals way too much information about the directory structure. Errors should be as brief as possible and never to show relevant information about the structure of the server. Revealing directory structures of the server can end up in a directory transversal exploitation to retrieve a file that a normal user is not intended to get access to.

Next two steps will be checking data input validation vulnerabilities such as SQL injections and Cross-site Scripting.

As previously said on the COTS section, Indivo uses BerkeleyDB as storage medium. BerkeleyDB doesn't use any querying language nor tables or rows or columns, and is not possible to inject it in anyway. It uses an API to perform the actions and stores data in a text encrypted file.
The other place that can be checked for injection is the log database that stores successful logins with their username and session ID, but as the query is not constructed until the username and password have been verified against the BerkeleyDB, it is not possible to inject, as whatever trial to inject in user or password fields will result on an unsuccessful login and the query against the MySQL database will not be constructed at all.

To check for Cross-site scripting vulnerabilities, I have tried to insert scripting language (like <script>alert(document.cookie)</script>) in fields that after being edited are displayed again to the same user or to others (like insurance notes).
Trying this has brought not any successful result as Indivo seems to filter scripting language in a way it is not displayed again to any user at all. For

example, in the case of the insurance notes instead of displaying an alert, it displays an empty note.

After this, I will try to browse through the server directory structure to see if relevant documents that are not intended to be accessed can be actually retrieved in some way.
By the error messages given by Indivo, one can more or less get a quite accurate directory structure. After browsing through it for a while, a lot of documents of the Indivo UI can be accessed, including the document were Indivo stores the access profiles and policies: "ping-security-profiles.xml", like can be seen on the following image.


```
http://localhost/indivo-ui/resources/xml/ping-security-profiles.xml - Windows Internet Explorer
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <ns2:ProfileCollection xmlns:ns2="http://ping.chip.org/xml/security">
 - <ProfileList>
    <Title>Primary Care Provider</Title>
    <Id>org:chip:ping:profile:primary-care-provider</Id>
    <Source>CHBoston</Source>
    <Description>This allows your primary care provider to read your record, it also allows your primary care provider to update and add documents.</Description>
    <Subject>Person</Subject>
    <RuleList>org:chip:ping:rule:send-message</RuleList>
    <RuleList>org:chip:ping:rule:query-record</RuleList>
    <RuleList>org:chip:ping:rule:read-labs</RuleList>
    <RuleList>org:chip:ping:rule:add-update-labs</RuleList>
    <RuleList>org:chip:ping:rule:read-immunizations</RuleList>
    <RuleList>org:chip:ping:rule:add-update-immunizations</RuleList>
    <RuleList>org:chip:ping:rule:read-clinical-notes</RuleList>
    <RuleList>org:chip:ping:rule:add-update-clinical-notes</RuleList>
    <RuleList>org:chip:ping:rule:read-demographics</RuleList>
    <RuleList>org:chip:ping:rule:read-contact-info</RuleList>
    <RuleList>org:chip:ping:rule:add-annotations</RuleList>
  </ProfileList>
 - <ProfileList>
    <Title>Spouse</Title>
    <Id>org:chip:ping:profile:spouse</Id>
    <Source>CHBoston</Source>
    <Description>This allows your spouse to read your record.</Description>
    <Subject>Person</Subject>
    <RuleList>org:chip:ping:rule:read-labs</RuleList>
    <RuleList>org:chip:ping:rule:read-immunizations</RuleList>
    <RuleList>org:chip:ping:rule:read-clinical-notes</RuleList>
    <RuleList>org:chip:ping:rule:read-demographics</RuleList>
    <RuleList>org:chip:ping:rule:read-contact-info</RuleList>
    <RuleList>org:chip:ping:rule:query-record</RuleList>
    <RuleList>org:chip:ping:rule:send-message</RuleList>
    <RuleList>org:chip:ping:rule:add-annotations</RuleList>
```


Figure 22: Accessing unintended files.

After this step, next thing to do is to check the integrity of the communications between the client and the server. Usually, for this a tool called packet sniffer is used, but as the testing environment has the client and the server on the same machine, I will be using a proxy that will intercept communications between client and server, as usually sniffers cannot intercept communications going through the loopback interface inside the computer.

The first thing to try is to login and to intercept the packet sent to the server with the login information. The result of this action can be seen in the next image.

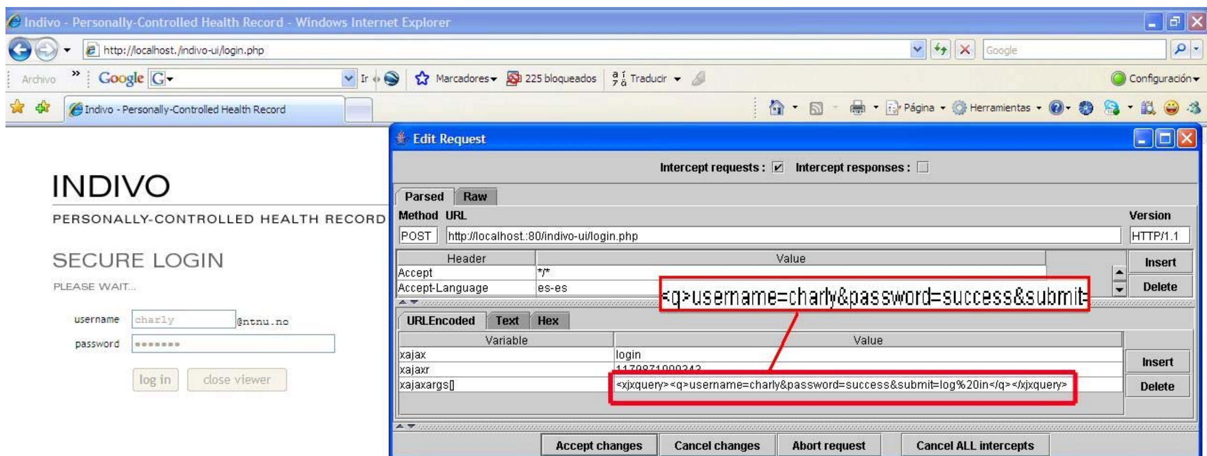

Figure 23: Capturing login information.

As can be seen, Indivo works by default over http, making the transference of the login data to the server over plain text, thus being possible to capture that information to impersonate a legitimate user.

The next thing I am going to try is to intercept a session ID and to try to use it to hijack the session of a legitimate user.

First step is to use the sniffer, or in this case the proxy to intercept a login packet. Obviously, it would be easier just using the login information on the packet, but the aim of this is to check whether Indivo permits to use a session ID more than once at the same time or not, and if it is enough to provide a valid session ID to access the session of a user or login information is also needed.

After intercepting the packet, I get a valid session ID (PHPSESSID= 896e578bffa76c6ed1e5490c2caea342) that is just being used by a user that logged in right now.

After this, I open a new browser and I directly enter the address of the Indivo viewer (http://localhost/indivo-ui/viewer.php). I intercept the request with the proxy, and I change it so the session ID that I stole will be sent with the request, as you can see in the following image.

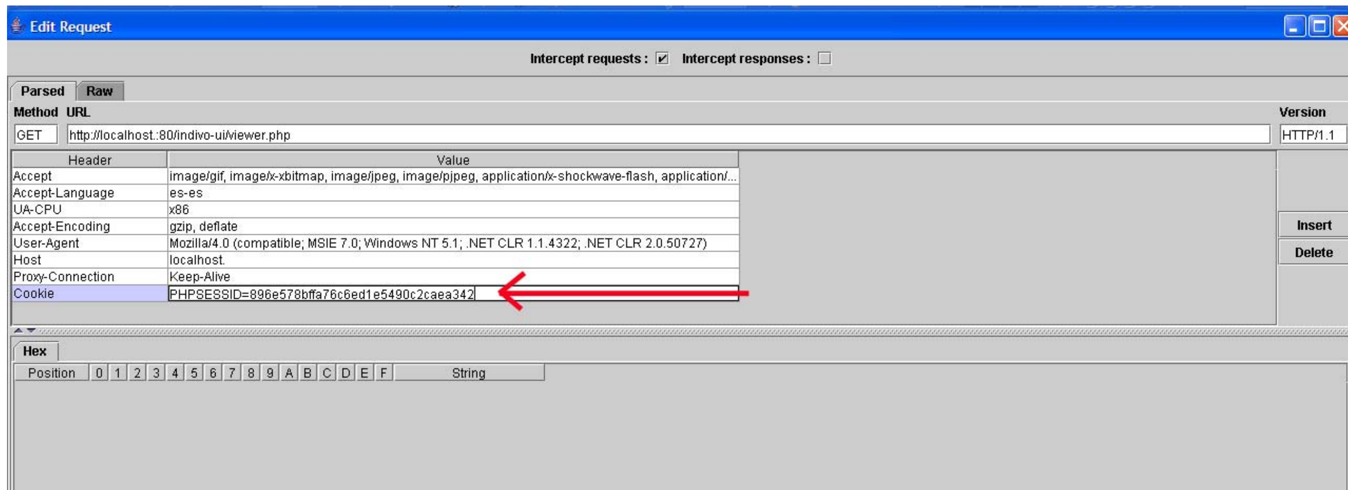

Figure 24: Editing request to forge a stolen SessionID.

As the result of all this process, I end up logged as administrator (it was the user's session ID I did the experiment on and that I stole), and I am able even to create a new user.
This proves that stolen Session IDs can be used as long as the legitimate user is still logged in and the session doesn't time-out.

Last set of tests will try to discover whether is possible or not to use automated password cracking tools to conduct a dictionary attack or a brute force attack against the login system.
As far as I have tried, not any remote password cracker works with Indivo, as Indivo login form doesn't use a regular method like post or get, but uses an own developed method called processForm(), thus making impossible to send automatically passwords through regular methods like post/get.
Anyways, I tried by hand to make incorrect logins by hand to check if it is possible to repeatedly login with the same user without blocking the access after several tries. The result is that you can login in an incorrect way as many times as you would like with the same user.
In consequence, if someone would write a remote tool that would be able to automate the login process, it would be totally possible to attempt to perform a dictionary attack or a brute force cracking on the Indivo login system.

5.5.1.2 Testing Indivo's Surrounding Technologies

In this section, I will address the vulnerabilities that exist for each of the technologies Indivo uses to run. For this, pages like Security Focus or Bugtraq will be used.
The main problem here relies in that Indivo needs specific versions of those technologies, making impossible to upgrade to newer and safer versions.

Apache Tomcat 5.5

Apache Tomcat presents two vulnerabilities, one that permits cross-site scripting and another that permits directory transversal attacks.

As explained on Security Focus webpage [24],

> *Apache Tomcat's documentation web application includes a sample application that is prone to multiple cross-site scripting vulnerabilities because it fails to properly sanitize user-supplied input.*
> *An attacker may leverage these issues to execute arbitrary script code in the browser of an unsuspecting user in the context of the affected site. This may allow the attacker to steal cookie-based authentication credentials and to launch other attacks. The only solution is to upgrade to last version of the product (v. 6.0.11).*
> *The other vulnerability consists on Apache HTTP servers running with the Tomcat servlet container are prone to a directory-traversal vulnerability because it fails to sufficiently sanitize user-supplied input data. Exploiting this issue allows attackers to access arbitrary files in the Tomcat webroot. This can expose sensitive information that could help the attacker launch further attacks.*

Apache Server 2.0.59

Only one vulnerability can be found in Apache Server 2.0.59, Apache "mod_alias" URL Validation Canonicalization Vulnerability.
In words of Secunia [25],

this vulnerability can be exploited by malicious people to bypass certain security restrictions and disclose potentially sensitive information. The vulnerability is caused due to a canonicalization error in the "mod_alias" module in the handling of case-sensitive alias directive arguments on file systems supporting case-insensitive directory names. This can e.g. be exploited to disclose the source code of applications placed in the "cgi-bin" directory on certain non-default configurations where the ScriptAlias directive references a directory inside the document root by accessing an URL with a capital directory name (e.g. "CGI-BIN"). The solution consists in editing the configuration to ensure that alias directives (e.g. ScriptAlias) references directories outside of the document root.

Java 1.5.0_08

Java version 1.5.0_08 doesn't present any known vulnerability according to Bugtraq, Secunia or SecurityFocus.

PHP 5.1.2

PHP 5.1.2 presents multiple vulnerabilities (44), and it is not worth to comment one by one, as they are too many, and information can be easily found on SecurityFocus webpage [24].

Java Bridge 3.0.7

Neither SecurityFocus nor Secunia can find any security issue on Java Bridge 3.0.7

MySQL 5.0.37

The following security vulnerabilities for MySQL 5.0 can be found in Secunia webpage [25]:

MySQL Two Privilege Escalation Security Issues, that consists on two security issues that have been reported in MySQL, which can be exploited by malicious users to gain escalated privileges. It is possible for a user to rename a table without having DROP privileges and it is possible that

stored routines defined with SQL SECURITY INVOKER do not change back privileges when returning and can be invoked by users to gain escalated privileges.

MySQL IF Query Denial of Service Vulnerability, that consists on an error when handling specially crafted IF queries, which can be exploited to crash the server (Denial of Service attack).

6. Discussion

Security is an important topic, not only to Open Source Consumer Applications, but to all kinds of computer system or software. The reason why consumer applications need to focus on security is because they tend to contain sensitive information that should not be compromised.

This is even truer when talking about web applications like Indivo, because being accessible in a remote way, puts them in a special position. The fact that they are open to attacks potentially launched by anyone connected to the Internet implies that a heavy load of threat countermeasures needs to be employed in order to protect confidential data.

It is important to understand that security is not something that may be added to an application after a complete development process. This is not only a risky idea, but also a great increase on the development effort, as modifying an already implemented application requires much more work than to integrate security through all the application's lifecycle, from the earlier stages of development until the last one.

Some people argue that integrating security all the way through the development process is costly as well, and it does require some extra time and effort, but the truth is that it saves you from unwanted results at the end of the development cycle.

Applications do need security, and the best way to make sure that they are developed in a way that confidential data is as safe as it can be, is making sure you pay attention to the potential security issues that might arise from the very beginning.

The main chapter of this thesis is the development of a Security Guideline. The developed Security Guideline is not only useful to avoid well-known threats and vulnerabilities, but it also encourages integration of security practice from the beginning of the application's lifecycle, that will result in a more secure application, so it may in fact help prevent future threats as well.

The Security Guideline includes general guidelines that should be present during all stages of the application's development, and also specific guidelines for every specific stage of the application's lifecycle.

About the guideline applicability, personally I didn't find any place after the first iteration that should need improvement, but I do believe that if applied to a full

project, or applied by a more experienced expert, then more areas of improvement should be found.

After developing the guideline, the next step taken has been to try the guideline over a real application. There is no better way to check if something is working than to try it on a real environment.

To try the guideline, an Open Source Consumer Application for the healthcare field called Indivo was chosen. The high confidentiality that the data this application manages requires made it a good target to test the guideline.
Of course, it would have been much better to apply the guideline from the beginning on an application that was in its earliest stage of development, but to perform this, great amount of time would have been required, as I would have had to go through all the development of an application, and it was not possible to spend that much time.
That's why some of the sections of the Security Guideline were not applied as intended, but just used to check if decisions already taken in the development of Indivo made sense and were good choices.
The sections that could be applied as intended were the design of the test plan, the risk analysis, the code review and the penetration testing part both in Indivo and in the surrounding technologies.

The test plan was developed trying not to leave anything important that should be checked out of the scope, but also trying not to make it too big as the amount of time to develop the thesis was limited.
After having applied the Test Plan to Indivo, and because of the results obtained, I think that it was complete enough to achieve a reasonable security evaluation, but that maybe with more time I could have gone in more deep for example in the code review, as making a good code review implies having a full understanding of the application's code, so one can check the real severity of the issues discovered by the automated code reviewing tool, and to discard in a more confident way the false positives.

About the risk analysis, after seeing the results of the penetration testing, I think I assessed quite in an accurate way which could be the risks that Indivo might be exposed to.

The scope of the thesis only involved one iteration on the testing of the guideline, and that's why I classified the risks by severity and not by risk, that requires to know also the probability of a given vulnerability, as risk is severity*probability. Usually to know the probability of a vulnerability being exploited more than one iteration is needed, as the first iteration is used to evaluate if the vulnerability is easy to detect and to exploit.
A good example of how the first iteration can evaluate the probability of a certain vulnerability being exploited is for example the case of session ID's being guessable. After the first iteration, it has been proved that session ID's on Indivo are random and don't follow any particular distribution, thus being difficult to guess. This fact would set the probability for that particular vulnerability happening really low. In the other hand, after first iteration probability for the vulnerability of the login information being intercepted and exploited should be set really high as it is something easy to perform.

The code review was carried out using an automated code reviewing tool called FindBugs. From the results obtained, I only commented the bugs that affected security of the application directly, as the ones that were only a matter of code correctness and efficiency fall out of the scope of this thesis.
As said before, this section should need improvement as the time limited me on getting a deep understanding of the source code of Indivo, thus making not possible to determine totally whether the found bugs were truly a security menace or not.

Last but not least, I will comment the results obtained in the penetration test section. This section was the most satisfactory in my point of view, as it not only revealed several security issues, but let me to practice a real penetration test, which gave me a better understanding on how several vulnerabilities work and made me to improve my knowledge and creativity a lot.

The first step of the testing involved getting the characterization of the system to evaluate. This was not a real necessary task as all the characteristics of the system were already known, but it is a good way to check which information an outsider attacker can get of the system.

The second step was to use a proxy between the client and the server to check several issues like strength of session IDs, changing hidden fields to reveal if it

is possible to take advantage of it to perform some unwanted action, changing methods on the requests sent to Indivo and sending malformed requests to make Indivo display error messages in purpose to check if relevant information was disclosed.

After analyzing several session IDs, it has been proved that they are random enough and they are not distributed in a fixed pattern that could make them guessable; Indivo's session IDs present a high degree of strength.

Changing hidden fields didn't allow performing any unwanted action, as they didn't modify relevant information, and when they did, Indivo access policy denied the action.

Changing the methods inside the requests sent to Indivo resulted in the possibility of accessing administrative interfaces that should be not accessible to regular users. This cannot be considered a high risk degree as any action made with that interfaces was denied by Indivo's access policy. This fact doesn't mean that it is not in fact a vulnerability that should be fixed, as administrative interfaces should have restricted access to regular users.

By sending malformed requests to Indivo, I made Indivo to display error messages that contained way too much information about the server's directory structure. This should be fixed as it can be used to carry directory transversal attacks with the result of accessing unintended files.

Next step was to perform various checks to find out if Indivo was vulnerable to injection, to cross-site scripting attacks, and if by browsing through server directories files that shouldn't be accessed could be accessed.

Because of the nature of the data storage that Indivo uses (BerkeleyDB), it was not possible to perform any injection attack on Indivo. Trying to embed scripting languages in parts of Indivo that could be displayed back to other users or to the same, didn't worked for the cross-site scripting checks, as Indivo seems to filter this by not displaying scripts.

Because of the errors displayed by Indivo, it is not difficult to get in mind a quite accurate directory structure of the server. By exploiting this, I was able to access the access policy file on the server. This should be fixed, as accessing this kind of files is totally unacceptable and should be restricted.

After this checks, the next step is to check the integrity of the communications between the client and the server. In a normal testing scenario, this would be done with a packet sniffer or a packet capture tool, but in this case, as both

client and server are in the same machine and communications use the loopback interface, regular sniffers cannot capture this kind of traffic and using a proxy between client and server was decided for evaluation purpose.
By intercepting the packets that were going from the client to the server, login information could be fully retrieved. This is a direct consequence of Indivo working over http by default, and not by https. This is a really important security issue and should be fixed.
By intercepting random packets, I got a session ID from a session that was currently active. I used this ID to try to perform a session hijack attack. By addressing my browser directly to the Indivo viewer, intercepting the packet requesting this action and modifying it inserting the session ID previously stolen, I managed to get inside the session of another user and to perform whatever action I wanted. This is a severe security issue that should be fixed, maybe not allowing two sessions in the application with the same session ID. I also checked if this attack worked once the legitimate user logged off, but it didn't, which means that Indivo does a good work eliminating used session IDs.

Last step of the penetration testing on Indivo involved using brute force and dictionary attacks to go through Indivo's login system. This couldn't be performed with the existing automated tools as Indivo uses not regular methods like GET or POST to send the login information form, but an own developed method called processForm().
What I was able to try by hand is to introduce repeatedly incorrect login information for the same user, to check if after several incorrect logins Indivo blocks the user account. The result was that Indivo allows to introduce unlimited times incorrect login information. This could end up in someone developing a self-made automated tool to try to crack Indivo's login system. It could be solved by implementing a two stage login or a delay on login.

Test of the surrounding technologies that Indivo needs to run was made simply querying databases of known vulnerabilities as SecurityFocus, Secunia or Bugtraq. This investigation revealed that several of the technologies that Indivo needs to run were affected by well known bugs.
Indivo requires specific versions of several products, making their update totally impossible, and this represents a security issue, as long as some of those vulnerabilities are only solved in newer versions of the same product, and this makes the system running Indivo exposed to several security issues that cannot

be avoided. The only solution is that Indivo should be made more flexible in the versions of the technologies it needs to run.

7. Conclusion and Further Work

The aim of this Master Thesis was to provide a better understand on how to develop secure Open Source Consumer Applications through the development of a Security Guideline and the test of this Guideline on a real Open Source Consumer Application.

One of the most important things that should be always present while developing an application from the first stage is to integrate security in every stage of development, from the first to the last. The philosophy of patch-after-development has been proved to be more expensive and inefficient.

In my point of view, both the development of the Security Guideline and the posterior testing of the guideline have been successful, as a working methodology has been established and several security issues have been revealed in Indivo.

When working on this thesis, I have experienced the difficulty of finding information on how to effectively assess security in applications, in addition to descriptions of vulnerabilities, threats, and countermeasures. I therefore hope that my thesis will provide information that will be useful for those who are to develop a consumer application, both open source, like the one evaluated in this thesis, and closed source, as the fundamental principles for security are the same. As the Security Guideline is platform independent, it can be applied to every application.

7.1 Further Work

Further work that could be done after this thesis is to test the guideline on a starting project that is in its first stage of development, so the guideline can be totally tested and applied to more than one iteration. This way I believe better results could be achieved, and also the guideline could be updated and redesigned in the steps that will be find out that don’t work as intended.

Appendix A: References

[1] Eugene H. Spafford, director of the Purdue Center for Education and Research in Information Assurance and Security.

[2] Apache Website,
http://httpd.apache.org/docs/1.3/howto/auth.html.

[3] Microsoft TechNet Security,
http://www.microsoft.com/technet/archive/community/columns/security/essays/vulnrbl.mspx.

[4] OWASP Website
http://www.owasp.org/

[5] Smashing The Stack For Fun And Profit, Aleph One
http://insecure.org/stf/smashstack.html.

[6] Andrews and Whittaker, How to Break Web Software, Addison-Wesley (2006).

[7] World Wide Web Consortium (W3C)
http://www.w3.org/

[8] Wikipedia, The free Encyclopedia
http://en.wikipedia.org

[9] European Parliament, Temporary Committee on the ECHELON System
http://fas.org/irp/program/process/europarl_draft.pdf p.69.

[10] Viega and McGraw, Building Secure Software, Addison-Wesley (2002).

[11] Carnegie Mellon, Software Engineering Institute
http://www.sei.cmu.edu/str/descriptions/clientserver.html

[12] Gary C.Kessler, An Overview on Cryptography (1998)
http://www.garykessler.net/library/crypto.html

[13] Stoneburner, Goguen and Feringa, A Risk Management Guide for Information Technology Systems, NIST (National Institute of Standards and Technology) (2002)

[14] NTNU Indivo Wiki
http://hiwiki.idi.ntnu.no/hiwiki/index.php/PING_Discovery:_Pilot_implementasjon_og_utforskning_av_PING

[15] The Open Web Application Security Project - The Guide project,
http://www.owasp.org/documentation/guide/guide_downloads.html

[16] Gary McGraw, Software Security – Building Security in, Addison-Wesley.

[17] Steven Splaine, Testing Web Security, Wiley Publishing (2002).

[18] Jason Voegele webpage, http://www.jvoegele.com/software/langcomp.html

[19] Microsoft Support Centre, http://support.microsoft.com/kb/176799/en-us

[20] Krishnan Viswanath, The New RMI (2005),
http://today.java.net/pub/a/today/2005/10/06/the-new-rmi.html

[21] CERT® Coordination Center, Choosing an Operating System
http://www.cert.org/tech_tips/choose_operating_sys.html

[22] Jason Miller, Open Source versus Closed Source Security (2004),
http://www.securityfocus.com/columnists/269

[23] OWASP Testing Guide v2. © 2002-2007 OWASP Foundation,
http://www.owasp.org/images/e/e0/OWASP_Testing_Guide_v2_pdf.zip

[24] SecurityFocus, http://www.securityfocus.com

[25] Secunia, http://www.secunia.com

[26] http://www.cs.ucla.edu/~kohler/class/05f-osp/notes/fig19-01.jpg

[27] http://www.awprofessional.com/content/images/chap4_0321369440
/elementLinks/04fig17.jpg

[28] http://en.wikipedia.org/wiki/Image:Webservices.png

[29] http://content.answers.com/main/content/wp/en/thumb/8/8f/400px-
Corba_Server.png

[30] http://en.wikipedia.org/wiki/Image:JavaRMI.JPG

[31] http://scitec.uwichill.edu.bb/cmp/online/CS22K/images/twotier.gif

[32] http://scitec.uwichill.edu.bb/cmp/online/CS22K/images/threetier.gif

[33] http://content.answers.com/main/content/wp/en-
commons/thumb/e/e9/280px-Public_key_encryption.svg.png