A Layer-Independent Taxonomy for Evaluating Application Security
and its application to the Ethos OS

BY
FRANCESCO COSTA
B.S., Computer Engineering, Politecnico di Milano, Milan, Italy, 2009
M.S., Computer Engineering, Politecnico di Milano, Milan, Italy, 2012

THESIS
Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2012

Chicago, Illinois

Defense Committee:
Jon A. Solworth, Chair and Advisor
Jakob Eriksson
Stefano Zanero, Politecnico di Milano
To you.
ACKNOWLEDGMENTS

I would like to thank you my advisor professor Jon A. Solworth for supporting and guiding me during these months I spent working in the Ethos research group and to this thesis. I want to thank you all the people in the lab for helping me and making me feel like home, in particular thank you to Mike, Wenyuan, Xu, Yaohua, and Siming. Thanks also to my Italian advisor, professor Stefano Zanero, for the patience in supporting during this overseas graduation. A special thanks goes to Lynn Thomas, who has been our guardian angel for almost two years during this master.

FC
### LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>19</td>
</tr>
<tr>
<td>II</td>
<td>20</td>
</tr>
<tr>
<td>III</td>
<td>22</td>
</tr>
<tr>
<td>IV</td>
<td>25</td>
</tr>
<tr>
<td>V</td>
<td>29</td>
</tr>
<tr>
<td>VI</td>
<td>31</td>
</tr>
<tr>
<td>VII</td>
<td>32</td>
</tr>
<tr>
<td>VIII</td>
<td>34</td>
</tr>
<tr>
<td>IX</td>
<td>35</td>
</tr>
<tr>
<td>X</td>
<td>36</td>
</tr>
<tr>
<td>XI</td>
<td>38</td>
</tr>
<tr>
<td>XII</td>
<td>41</td>
</tr>
<tr>
<td>XIII</td>
<td>127</td>
</tr>
<tr>
<td>XIV</td>
<td>128</td>
</tr>
</tbody>
</table>

**THREATS AND SECURITY PROPERTIES**

**DREAD CHARACTERIZATION**

**CVSS DISTRIBUTION CHART**

**SANS TOP 25 MOST DANGEROUS SOFTWARE ERRORS**

**WEBER TAXONOMY**

**ASLAM TAXONOMY**

**SOLWORTH TAXONOMY**

**MAPPINGS ON WEBER TAXONOMY**

**MAPPINGS ON ASLAM TAXONOMY**

**MAPPINGS ON SOLWORTH TAXONOMY**

**MAPPING SUMMARY**

**PROPOSED TAXONOMY**

**COMPARISON SUMMARY: RISK, RELATED VECTOR AND DAMAGES**

**COMPARISON SUMMARY: RISK, AFFECTED MODULES AND AUDIT PLANS**


LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CVSS overview</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>Simplified schema of Postfix modules</td>
<td>125</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS

Top 25  CWE/SANS Top 25 Most Dangerous Software Error

SH  Security Holes

APL  Application Programming Language

EM  Ethos Markup

ES  Ethos Style

ACS  Access Control System

MAC  Mandatory Access Control

DAC  Discretionary Access Control

RBAC  Role Based Access Control

TE  Type Enforcement

ACL  Access Control List

ACE  Access Control Entry

VMs  Virtual Machines

JVM  Java Virtual Machines

TPM  Trusted Platform Module

MMU  Memory Management Unit

MAC  Message Authentication Code
LIST OF ABBREVIATIONS (Continued)

**AES** Advanced Encryption Standard

**ECC** Elliptic Curve Cryptography

**ECDH** Elliptic Curve Diffie-Hellmann

**RSA** Rivest, Shamir, Adleman

**OO** Object Oriented

**ETN** Ethos Type Notation

**RPC** Remote Procedure Call

**ASLR** Address Space Layout Randomization

**LLVM** Low Level Virtual Machine

**ARC** Automatic Reference Counting

**PKI** Public Key Infrastructure

**SSL** Secure Socket Layer

**TLS** Transport Secure Layer

**SSH** Secure Shell

**GSSAPI** Generic Security Services Application Program Interface

**PAM** Pluggable Authentication Modules

**DBMS** Database Management System

**TCB** Trusting Computing Base
LIST OF ABBREVIATIONS (Continued)

LSM  Trusting Computing Base Linux Security Modules

LEAP  Language for Expressing Authorization Properties

IPC  Interprocess Communication

2PL  2-Phase Locking

TTCTTOU  Time To Check To Time Of Use

DoS  Denial of Service

SASL  Simple Authentication and Security Layer

APS  Authorization Policy Specification

CVE  Common Vulnerabilities and Exposures

CWE  Common Weaknesses Enumeration

CVSS  Common Vulnerability Scoring System
SUMMARY

Providing a crisp definition of what is secure software, and how to establish whether or not some software is more secure than other is an extremely hard problem to solve. Many different attempts have been done, but none was so successful as to became a de facto standard.

Security breaches and vulnerability in software are topics that are rapidly gaining importance and fame. Today’s software is often poorly conceived and security is usually a marginal detail in the developing process.

It is really hard to compare, analyze and selectively address security in software. In this work, we try to build a conceptual framework to understand applications in relation to their execution environment and analyze them independently from the system layering. In this work, we aim for an approach which is layer-independent and capable of analyzing different systems regardless their architectural design or layering.

The thesis is organized as follows:

**Chapter1** in this chapter a brief high-level introduction is provided to the problem of security in software. In the midsection there is a short overview on the focus of the work and its context. In the last part is introduced some simple terminology which is intended to be used during the work.

**Chapter2** is devoted to provide a concise overview on the Ethos operating system, its major components, some information about its design and a brief summary of related security implications which will be detailed in following chapters.
Chapter 3 is about approaching the problem of understanding how to comparatively analyze security in applications and systems, despite an heterogeneous environment. An iterative refinement process across levels of abstraction is started in this chapter by surveying various common methodologies and gathering context information.

Chapter 4 the abstraction process is continued from the previous chapter and brought to the level of a taxonomization of security areas and components. In this context the major works in the field have been analyzed and a new layer-independent methodology has been proposed.

Chapter 5 is dedicated to provide, according to the proposed taxonomy, the set of tools that will be used to directly employ it in a practical case of interest, the Ethos case. While maintaining independence with systems and their design and layering, we produced a collection of prevention and mitigation mechanisms per area.

Chapter 6 uses the resulting methodology to compare in depth the Ethos case versus traditional operating systems. A comparative approach per area is developed in details.

Chapter 7 builds on the system understanding produced in the previous chapter, to apply the previously developed reasoning to evaluate a concrete application case. This final comparison involves an application running on Ethos and one for POSIX systems.

Chapter 8 contains a summary of the work done, results obtained and considerations about the future works and what role this work will cover in the future development of the Ethos project.
CHAPTER 1

INTRODUCTION

1.1 Stating the problem

Every year about 6000 vulnerabilities are officially classified in the NIST National Vulnerabilities Database. These are only a small fraction of the total number discovered, some researchers have estimated this number to be around 130.000/year. The purpose of reporting these numbers is to give a concrete sense of how serious the security situation is for today’s software.

Usually these vulnerabilities are not actually perceived by the final users, who are light years from the technical understanding of what happens in software and services that they daily use. This is due, in part, to the fact that we are in the era of Social Networks, where everything is shared and personal life become public, users seem to not actually care about leaking personal information to unknown subjects around the world. This is probably because there is a lot of disinformation about this topic among common people, but hopefully this is going to change.

Big companies security failures (for example, the Sony Playstation Network case (1; 2; 3) that for an entire month paralyzed the biggest gaming platform of the world) are increasing appreciation of this problem. It’s only a matter of bringing the technical aspect on a dimension near to the understanding of the average user: in the Sony case people actually experienced a
substantial service interruption in addition to a loss of personal privacy, in the Citigroup Bank case (4) thousands of checking accounts have been exposed.

Even if people seem not to care about disclosing personal life details, for sure they do care about their money and their favorite hobbies. Today’s software is plagued in a massive way by security holes and weaknesses which allow attackers to take advantage of poorly thought out software, that has always placed the security aspect in secondary place.

This can be explained in several different ways which have a common denominator: building secure software is difficult and requires experience and time, these both cost money to software companies, making sure that an application is secure cost time and in the end again money. Security flaws can always be justified as honest mistakes, things that are likely to happen considering the undecidable nature of software. This allows people to just not care about security or doing it in the cheapest way possible which unfortunately often corresponds to the worst.

1.2 Work overview

Taking as a reference the snapshot of today’s software situation, this work attempts a comparison of existing way of conceiving and thinking about software with Ethos OS philosophy which puts security first. The problem is extremely complex; security is a field which has in its nature a component of informality. There isn’t a universal accepted definition of what secure software is, what security is and how it can be unequivocally measured. This is an old problem to which several scientist and experts have tried to answer, proposing plenty of definitions, models, guidelines etc. The goal of this work is not to provide a final answer to a problem
that most likely doesn’t have a crisp answer, and is intrinsically well suited for having many interpretations depending on the perspective from which is observed.

The main part of this work is related to the development of a way to approach the problem, understanding the environment related to it and providing means of analyzing and comparing different systems and the applications that run on top of them. We have reasons to believe that the Ethos design and infrastructure, which rather than traditional systems, is security-oriented, has impact on the resulting security properties of applications which run on it. The proposed work has been developed in the context of the Ethos research group and inspired by the desire of comparing systems and entities which are very heterogeneous and differently designed and layered.

In the first part will be provided an overview of the Ethos OS, followed by a critical analysis of the problem and its difficulties. Later on, will be provided the set of steps and the reasoning which brought to the formulation of a layer-independent taxonomy associated to security prevention and mitigation mechanisms. In the final part of the work the taxonomy will be extensively applied to the Ethos case and ultimately in a concrete application case.

1.3 General Terminology: Bug, Flaws and Security Holes

Giving a precise definition of the terms Bug, Flaw and Security Hole is very difficult. Many similar but slightly different definitions can be found. The ones used for the purpose of this work are provided below:

- **Bug** and **Flaw** are synonyms and denote a failure in the understanding when implementing the program or of the underlying architecture and environment. This failure causes
the program to produce unexpected results or behavior. Bugs or Flaws are present in programs in an unknown number and, in general, there is no formal way of proving their absence given a fragment of code.

- **Security Holes (SH)** are vulnerabilities of a computer system that allow attackers to harm the system and all the infrastructures that depends from that system. In order for a SH to be defined so, three conditions have to be satisfied:

  **Bug:** it has to be a bug

  **Exploitability:** it has to be exploitable by an attacker

  **Violation:** its exploitation has to result in a security property violation

Correlation between Bugs and SH is not obvious, the only fact that can be precisely inferred is that there are at least as many bugs as SH. On the average—but it is not a formal statement—a reduction in the number of bugs directly translates in a reduction of the number of SH in a given program.
CHAPTER 2

ETHOS

Ethos is a high strength Operating System, specifically developed to address security related problems. The information that will be provided in this chapter have been extracted by studying internal project documentation (5; 6; 7; 8). The main goal in Ethos is providing a better environment in term of security properties. This is done by designing the OS with security as the primary focus, rather than an afterthought after all the other problems have been solved. One of the final purposes of Ethos is allowing the production of software which has better security properties together with reducing the amount of effort and responsibilities required to the programmer. This can be achieved by pursuing several subgoals:

Driving down the complexity: shrinking dramatically the code base of the OS, reducing the probability of incorporating flaws in the OS and increasing the quality of the verification methodologies that can be applied to improve its quality. This can be carried out by reducing accidental complexity first and then providing a well-balanced design in term of OS and Application Programming Language (APL).

Relieving the application programmer from critical tasks: this is achieved by moving the complexity related to security into the OS. Current OSs provide several different ways of addressing each security problem, with a variety of framework, services, libraries, etc.
Ethos takes care of all the aspects related to security, by default, provides one clear simple and effective interface to use its services.

**Security first:** considering security the main concern during design. This not only means better structuring, better abstractions, but taking the courageous path of not compromising with the past. If a currently available mechanism has been shown to be heavily wrong from the point of view of security, there is no reason to be compatible with that, even if it is in widespread use. To achieve higher quality in term of security properties, a drastic rethinking has to be done and there is very tiny room for compromises.

### 2.1 System Overview

In the following paragraphs a brief overview of Ethos features will be provided. The roles of its components will be described as well as the functionalities they provide. Each of the following components does not necessarily refer to a physical 1:1 implementation, but is a high level abstraction of some security properties that are managed by the OS.

#### 2.1.1 Processes

The concept of process itself is not very different from the usual semantic that this object has in traditional OSs. What is different is how Ethos processes are managed, how they communicate with each other, and how they deal with system resources.

- **Parallelism and Concurrency:** one of the main differences in Ethos is that, in order to provide more security, explicit parallelism is not allowed. There is no support for Threads
in the way we know it. This doesn’t mean that parallelism is not exploited; in fact Ethos uses an asynchronous model based on Events which will be further explained in 6.7.2.

- **Interprocess Communication (IPC):** In Ethos the whole structure of IPC has been simplified, this process went through unification with the networking system. Instead of a multitude of different services like, *sockets* and *pipes* in the named and unnamed version, *message queues*, *signals* and so on, Ethos provides a very simple structure which avoid all the security issues related to concurrency, which are instead present in the traditional shared memory and signals mechanism. Five networking system calls are provided and accomplish the whole set of needs related to internal and remote communication.

- **Resource Cap:** In Ethos it is possible to specify a limit on the amount of resources, amount of memory and CPU cycles, which a process can use. This prevent from having to impose a system-wide cap on resources whenever is necessary to control the evolution of some situations. Together with authorization settings the user has the possibility of managing the resources that will be available to a particular process.

### 2.1.2 Virtual Processes

In Ethos a Virtual Process is like a normal process from the point of view of the information carried, but other than a regular one it doesn’t allocate any resources if it’s not explicitly materialized. Virtual Processes are created by use of `fdSend` system call instead of the usual `exec` operation and they are used for authentication purposes.

A virtual process per user is created if it doesn’t exist, it represents the user, he explicitly owns the process. Through this process and some specific system calls like `authenticate` and
fdSend the user is authenticated. If positive the program will continue its regular execution flow, otherwise it will exit. The user is totally transparent to the authentication mechanism which resides in the kernel.

2.1.3 Events

As said before Events are the way in which Ethos deals with parallelism: no explicit concurrency is allowed and the duty of extracting throughput by parallel-oriented code writing is demanded to the OS and not to the programmer anymore. Ethos is an asynchronous OS, which means that system calls can have a completion order different than the issued one, as long as correctness of resulting output is granted. Usually what happens is that the syscall with the highest latency, the ones that has to deal with I/O for example, are queued until the resources are available and in the meanwhile other less time consuming syscalls are executed.

In Ethos an Event is represented by a pair of a descriptor and a label which identifies the Event Type. In order to enforce correctness the OS maintain a dynamic structure called Event Wait Tree in which Events are placed in leaves and internal nodes store the completion state of the subtree. In this way the OS is able to manage the scheduling of events.

2.1.4 File System and File Descriptors

Ethos file system is based on typed object, every directory in contains files of the same type. Each file can contain only object of the same type, of course it can be a complex object which contains many other components. Descriptors are used as long term handles to manage a variety of situations, so according to their type they can refer to different physical or abstract
objects: for example a communication channel between two hosts, a directory or group set (concept used for authorization 6.3.2)

2.1.5 Networking

Big role is played by networking: by design Ethos simplifies the interface of communication providing a unique clear set of primitives to perform and deal with every type of communication. Remote and local communication are then managed with the same set of functions by the OS. Ethos networking infrastructure incorporates both encryption and authorization, this is intrinsic of the OS not something to be provided and integrated by third party libraries. Networking in Ethos is resilient to Denial of Service (DoS) attacks by means of the resource bounds that can be imposed on specific processes. Communications in Ethos networking are again type safe, this aspect will be further explained in 6.6.2 sections. The set of confidentiality, integrity and authentication properties is supplied through the use of Salt (NaCl) cryptographic library. This state-of-the-art library provides high-speed, high security services for encryption, signing and verification by using Elliptic Curve Cryptography (ECC) public key cryptography, SALSA20 stream cipher for encryption and Poly1305 for authentication. Ethos does not require application programmer to deal with IP addresses, port numbers, authentication, authorization, encryption. It does not rely on network host identification (IP numbers or fully qualified domain names) for security. Network services are named by path names (as with UNIX sockets) but unlike in UNIX these names are usable outside the system.
2.1.6 Types

One of the most important facility provided by Ethos is Ethos Type Notation (ETN). Everything in the OS is managed as a typed object, this provides a large set of guarantees and a solid infrastructure to build tools for formal and automated verification of security properties. These kinds of facilities are usually provided at APL level and every runtime environment incorporates a set of libraries to allow typing of objects and offer an higher interface which is not related to low-level issues like byte encoding, architecture compatibility etc. Even in this case the choice was to provide all these features at OS level. This drastically decrease the complexity of having several different languages that have deal with typing in different ways, overloading the system with runtime implementations for each of them. ETN was designed to provide a language-independent type system, which describes data and is decoupled from the code that will manipulate these data. It is used to type IPC and all those kind of objects streamed through the network. Due to the fact that typing is directly managed at OS level, all the objects are under mediation of the OS and so all the previously described security facilities and infrastructures applies. Encoder and decoder abstractions provide services to virtually pack information and sending to the other side end of communication channel, end points can rely on different APL and Ethos will take care of translation.

2.1.7 Authentication

Authentication in Ethos is provided by using cryptographic public keys to identify users and hosts. Conceptually the same user can be represented in different ways allowing some
operations to be performed, for example, from a work computer, but not from a home one.

Ethos supports three way of dealing with authentication:

**Anonymous:** in this scheme public random keys are generated as needed, the key is then not traceable, but even if the communication is anonymous, confidentiality and integrity is still guaranteed.

**Fixed Key:** a higher level of authentication is provided by keeping track of keys, in such a way that a particular user or a particular host is associated with a key.

**Registered key:** this final level requires that keys related to users and host are linked to a physical identity verified through the usual scheme of public Certification Authorities or by a private one.

Basically all these levels of authentication require that the user has the appropriate degree of freedom in managing keys according to a specific situation. In some case in order for the scheme to be effective, the key has to be made not accessible to the user. By default keys will be stored and managed in a safe way and it won’t be possible to steal keys by means of a simple directory read (e.g. `.ssh` folder in traditional implementation of `ssh`).

2.1.8 **Authorization**

Authorization in Ethos is provided by **Language for Expressing Authorization Properties (LEAP)**. Every process has a **user** and a **executable** label, these information are used to determine whether or not a process is allowed to perform some actions. LEAP model allows to specify access control with a high level specification language, rather than dealing with Access Control Matrix which has low flexibility and makes easy to introduce mistakes during the
encoding phase. Moreover LEAP has decidable security properties which then allow to verify a set of requirements during the design phase. Because every sensible operations on objects in Ethos is directly managed and mediated by the OS, authorization is fully integrated in every part of the processing and in every different scenario: file management, Remote Procedure Call (RPC), IPC etc.

2.2 Security implications of Ethos infrastructure

Components described in the previous section have an impact in term of security, this section aims to give an overview about some important reflections on overall security, which are direct consequences of the design of such components. This is not intended to be a very detailed analysis, more specific consideration will be carried out in Chapter 6, the purpose of this is just giving general information about Ethos structure and management of some security aspects.

Processes: since interfaces for process communication are narrowed down and simplified and management of resources comes in a completely different fashion than the usual, all issues related to concurrency are avoided or extremely mitigated. The direct management of permissions and resources per process takes care of authorization and availability issues.

Virtual Processes: using virtual processes is not necessary anymore to perform setuid operation when authenticating a user, the ownership never changes, avoiding a large set of risky situations like privilege escalation.

Events: this component takes care of all the concurrency related security issues, relieving the programmer from the complexity of explicitly managing locks, synchronization, etc.
**File Descriptors**: they are important objects that take part in several security sensitive operations, in Ethos their role is even more important than in traditional systems, they are subject to an authorization policy.

**Networking and Types**: these two blocks are fundamental in dealing with all kinds of communication and user interaction related problems. The whole stream of data which comes from outside to the system and vice-versa is vetted and go through these two interfaces, this grants a powerful mediation action performed by the OS.

**Authentication and Authorization**: these two components has been designed specifically to provide better services with a much clearer understanding of what is the general assignment of permissions in the system. They are deeply incorporated in every system activity, so there is no way they can be bypassed or omitted.

### 2.3 Lifecycle

In software design and development, the actual code is not the only issue. Quality products and good results requires a careful design of all the phases of the lifecycle. If a development process is strongly and efficiently structured to be supported with intense and constant developing, testing and organizing activity, the quality of the final product will be increased. In term of security properties, stressing the design with accurate analysis and testing directly translates in a decrease in flaws.

The importance of accurately managing software development, is not only a matter of producing a higher quality product: the cost of fixing flawed code grows exponentially with the time of discovering of the bug (9). Enforcing specific set of assurance and testing procedures
on the developing process, can initially result in a greater effort in term of time and money, but is going to set higher standards for resulting products and smaller future maintenance costs.

Specifically in the field of computer security, higher quality in term of security can’t be achieved without going through severe testing and developing methodologies routine. The main point of reducing complexity and providing a better environment which is less error prone, is a huge improvement in the effort of reducing the final density of bugs. Unfortunately this alone is not sufficient to cope with the problem of providing secure code and must be joined with a careful and quality oriented design of the development process.

Ethos policy about lifecycle is very straightforward. It views the problem as two different lifecycles, one related to the system, in this case Ethos, and the other one related to application which will be build on top of this system. In general, system code is much more carefully designed and tested than generic application code. System code has to be more robust since from the security point of view it is usually part of the Trusting Computing Base (TCB). That’s why many types of exploits involving privilege escalation, break in through a weakly implemented application and then raise their privilege level in another way, rather than directly taking control of a system component which would be much more difficult. Moving back to the two lifecycles scenario, what usually happens is that systems provide a reasonable amount of functionality, but often lots of security relevant facilities and policies are at higher layers (this will be explained in detail during the rest of the thesis). The result of this is that often the code to properly secure an application has to be included in the application itself, or many security libraries have to be involved and in some way interconnected to the application and
in several occasions this operation is poorly structured and interfaces are not always easily under-standable. In addition to the underlying complexity of securing applications “case by case”, has to be added the fact that a common trend is that developers have strong background in programming, but poor skills in security, so sometimes due to lack of knowledge or distractions the resulting code is poorly secured.

Ethos philosophy can be seen as “aggressive”, meaning that, whenever is possible, all the complexity deriving from security has to be moved out from the application lifecycle into the system lifecycle. In case this is not feasible due to specific requirements or relevant technical incompatibilities, highly integrated facilities have to be provided to the developer with the cleanest and simplest possible interface, in such a way that they reduce complexity and consequently reduce errors in the application implementation. This goal is pursued during all the phases of design and implementation of the system. The choice of moving all the possible complexity related to security, out of the application lifecycle gives two main advantages: relieves the developer from the burden of producing every time an effective and robust implementation of security related code, and allow to system developers to maintain control over security sensitive strategies and robustness of the implementation.

2.4 Complexity

The Ethos system aims to reduce the complexity exposed to the user, together with the one internal to its infrastructure. The goal is providing an adequate abstraction which has a reasonable and manageable level of complexity, and place related components in a proper, according to a security perspective, collocation. Flaws are often referred to the design of
a system which could be improperly or ineffectively done. An improper layering can result in
dangerous placement of some components in the system which should not be able to share some
information. An ineffective design is an unjustified increasing of the complexity of the system,
which directly translate in a poor understanding and decreased usability to the programmer,
leading hence to an higher probability to introduce bugs in application implementation. Ethos
tries to embed all the security-related complexity, in the most simplified form, and provides to
the user a clear, easy to use interface which minimize the security-related effort.
CHAPTER 3

SCHEMATIZATION OF THE PROBLEM

3.1 Introduction: what is secure software?

In order to perform a meaningful analysis on secure applications, it is necessary to develop
criteria and spend some effort reasoning, aimed at defining what will be taken in account to
evaluate the security of an application. A reasonable approach is to isolate a set of dangerous
threats, analyze them, and study the different solutions, on the basis of the problems found
and the solutions proposed. It is then possible to address the problem more precisely, trying
to establish whether or not a platform behaves better than another in term of application
security. This kind of methodology can also help to analyze the impact that a specific platform
or framework has on applications running on top of them, if they are able to enforce some kind
of security properties. In this chapter will be covered the preliminary work done in order to
obtain a list of security issues which will be the starting point for the later analysis of application
security.

3.2 From Instances to Classes

The problem of obtaining the most important SHs, or stating if one is more important than
the other, is a tremendously hard and ambiguous task. Since there is not a common accepted
metric for the danger of a security hole, it is hardly achievable to obtain a ranking from which
is possible to extract the most important entries and analyze them.
Common Vulnerabilities and Exposures (CVE) is a dictionary of common names for publicly known information security vulnerabilities managed by MITRE, a not-for-profit organization which manages Federally Funded Research and Development Centers. The information provided is pretty specific: Vulnerabilities are in terms of specific software and release version. The details are low level. Thus, this database is not well suited for a high-level analysis which is abstracted from a particular occurrence. One approach might be to take the most important entries and abstract the concept behind the security hole and analyze that. There are two principal obstacles to this approach: the problem of ranking them and the size of the database, currently the CVE Database counts 47473 vulnerabilities.

3.2.1 Decomposing the problem: ranking and numerosity

There are several systems and indexes that could be used to give a score to a vulnerability, here are showed some of the most famous:

- STRIDE and DREAD and
- CVSS.

3.2.1.1 STRIDE and DREAD

STRIDE (10) is an intuitive system developed by Microsoft for classifying computer security threats. The approach consists of decomposing the system into relevant components, analyze each component against the threats in Table I, and if a threat is found, mitigate it. The process is iteratively repeated until the developer is reasonably confident that the analysis is complete.
TABLE I

THREATS AND SECURITY PROPERTIES

<table>
<thead>
<tr>
<th>Threat</th>
<th>Security Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spoofing</td>
<td>Authentication</td>
</tr>
<tr>
<td>Tampering</td>
<td>Integrity</td>
</tr>
<tr>
<td>Repudiation</td>
<td>Non-repudiation</td>
</tr>
<tr>
<td>Information disclosure</td>
<td>Confidentiality</td>
</tr>
<tr>
<td>Elevation of privilege</td>
<td>Authorization</td>
</tr>
</tbody>
</table>

Of course, this can’t prove in any way that the system is secure, but it is a fast and simple way of increasing the level of security of the software by getting rid of common trivial mistakes.

DREAD is a part of the system used to rate the threats on the basis of the formula

\[
\text{Risk} = \text{Probability} \times \text{Damage}.
\]

The calculation is incorporated in a scale which goes from 1 to 100, subdividing values in 3 bands Low, Med, High. The rating is performed against the following 5 components of the model:

- **Damage potential**: How great is the damage if the vulnerability is exploited?
- **Reproducibility**: How easy is it to reproduce the attack?
- **Exploitability**: How easy is it to launch an attack?
- **Affected users**: As a rough percentage, how many users are affected?
- **Discoverability**: How easy is it to find the vulnerability?
TABLE II

DREAD CHARACTERIZATION

<table>
<thead>
<tr>
<th>Rating</th>
<th>High (3)</th>
<th>Medium (2)</th>
<th>Low (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Damage potential</td>
<td>Leaking sensitive information</td>
<td>Leaking trivial information</td>
</tr>
<tr>
<td></td>
<td>The attacker can subvert the security system:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>get full trust authorization, run as admin,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>upload content</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>Reproducibility</td>
<td>The attack can be reproduced, but only with a</td>
<td>The attack is very difficult to reproduce, even</td>
</tr>
<tr>
<td></td>
<td>The attack can be reproduced every time and</td>
<td>timing window and a particular race situation.</td>
<td>with knowledge of the security hole.</td>
</tr>
<tr>
<td></td>
<td>does not require a timing window.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Exploitability</td>
<td>A skilled programmer could make the attack, then</td>
<td>The attack requires an extremely skilled person</td>
</tr>
<tr>
<td></td>
<td>A novice programmer could make the attack in</td>
<td>repeat the steps.</td>
<td>and in-depth knowledge every time to exploit.</td>
</tr>
<tr>
<td></td>
<td>a short time.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Affected users</td>
<td>All users, default configuration, key customers</td>
<td>Some users, non-default configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>All users, default configuration, key customers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Discoverability</td>
<td>Published information explains the attack. The</td>
<td>The vulnerability is in a seldom-used part of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vulnerability is found in the most commonly used</td>
<td>product and only a few users should come across</td>
</tr>
<tr>
<td></td>
<td></td>
<td>feature and is very noticeable.</td>
<td>it. It would take some thinking to see malicious</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using Table II a final weight from 1 to 15 is generated.

3.2.2 CVSS (Common Vulnerability Scoring System)

The Common Vulnerability Scoring System (CVSS) provides an open framework for communicating the characteristics and impacts of IT vulnerabilities. It consists of three groups:

**Base** the intrinsic qualities of a vulnerability

**Temporal** characteristics of a vulnerability that change over time

**Environmental** represents the characteristics of a vulnerability that are unique to any user’s environment
The computation is more complex than in the DREAD case and requires a spreadsheet.

### 3.2.3 Ineffectiveness of large scale range metrics

Even if both systems can be useful and effective for some purposes, they are not well suited for our goal of ranking SH in order to finally reduce the number to analyze. They both produce a range output instead of an absolute score. Investigating further for example considering CVSS it can be noticed in Table III that even considering only the most dangerous band of vulnerabilities we should consider more than 6000 detailed entries that should be later analyzed and abstracted to a general security problems.

For the reasons exposed above is necessary to move to a different kind of resource in order to have a more general approach. The Common Weaknesses Enumeration (CWE) is a
TABLE III

CVSS DISTRIBUTION CHART

Current CVSS Score Distribution For All Vulnerabilities

<table>
<thead>
<tr>
<th>CVSS Score</th>
<th>Number Of Vulnerabilities</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>29</td>
<td>0.10</td>
</tr>
<tr>
<td>1-2</td>
<td>289</td>
<td>0.60</td>
</tr>
<tr>
<td>2-3</td>
<td>2048</td>
<td>4.40</td>
</tr>
<tr>
<td>3-4</td>
<td>504</td>
<td>1.30</td>
</tr>
<tr>
<td>4-5</td>
<td>8177</td>
<td>17.40</td>
</tr>
<tr>
<td>5-6</td>
<td>9176</td>
<td>19.90</td>
</tr>
<tr>
<td>6-7</td>
<td>5104</td>
<td>10.80</td>
</tr>
<tr>
<td>7-8</td>
<td>14813</td>
<td>31.50</td>
</tr>
<tr>
<td>8-9</td>
<td>143</td>
<td>0.30</td>
</tr>
<tr>
<td>9-10</td>
<td>6491</td>
<td>13.80</td>
</tr>
<tr>
<td>Total</td>
<td>47074</td>
<td></td>
</tr>
</tbody>
</table>

Vulnerability Distribution By CVSS Scores

Diagram showing the distribution of vulnerabilities by CVSS score ranges.

Weighted Average CVSS Score: 6.9
project coordinated by MITRE, that aims at creating a catalog of software weaknesses and vulnerabilities. In this database every weakness has

- CWE Identifier Number/Name of the weakness type
- Description of the type
- Alternate terms for the weakness
- Description of the behavior of the weakness
- Description of the exploit of the weakness
- Likelihood of exploit for the weakness
- Description of the consequences of the exploit
- Potential mitigations

and many others. This useful resource provides a more abstract way of describing the security problems which is not related to the specific software and platform as CVE. Even if this is a big improvement regarding the purpose of isolating a list of the most important SH it doesn’t solve entirely the problem. Currently CWE comprehend 870 weaknesses and we still have the problem of ranking them in order to extract the top entries.

To narrow the set of weaknesses, we have chosen a list derived from the MITRE CWE, the 2011 CWE/SANS Top 25 Most Dangerous Software Error (Top 25). This CWE-compatible list contains the most widespread and critical errors which can result in serious vulnerabilities in software. They are often easy to find, and easy to exploit. They are dangerous in that they frequently allow attackers to completely take over the software, steal data, or prevent the
software from working at all. The list is the result of collaboration between the SANS Institute, MITRE, and many top software security experts in the US and Europe. It leverages experiences in the development of the SANS Top 20 attack vectors (11) and MITRE’s Common Weakness Enumeration (CWE) (12). MITRE maintains the CWE web site, with the support of the US Department of Homeland Security’s National Cyber Security Division. Top 25 entries are prioritized using inputs from over 20 different organizations, who evaluated each weakness based on prevalence, importance, and likelihood of exploit. It uses the Common Weakness Scoring System (CWSS) to score and rank the final results. The Top 25 list covers a small set of the most effective “Monster Mitigations”, which help developers reduce or eliminate entire groups of the Top 25 weaknesses, as well as many of the hundreds of weaknesses that are documented by CWE.

3.2.4 Abstracting from the list

Although the Top 25 is a big step ahead in moving from security reports to a more general and abstract security classification, some additional effort is needed in the abstraction process. Using a typical software engineering terminology, it is possible to say that the list gives us instances of SH. What is needed in order to perform a meaningful reasoning about security of application in Ethos, is moving from instances to classes of problems.

The motivations behind this further step is very simple: using the list as is, provides an array of specific problems in security, which are specifically related with some particular application used in a specific way. Even though the list is a level above the mere security report bulletin, this is not enough, because in order to carry out a significant reasoning about application
<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper neutralization of Special Elements used in SQL Command (SQL Injection)</td>
</tr>
<tr>
<td>2</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper neutralization of Special Elements used in OS Command (OS Injection)</td>
</tr>
<tr>
<td>3</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking the Size of Input (Classic Buffer Overflow)</td>
</tr>
<tr>
<td>4</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)</td>
</tr>
<tr>
<td>5</td>
<td>76.9</td>
<td>CWE-306</td>
<td>Missing Authentication for Critical Function</td>
</tr>
<tr>
<td>6</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>7</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>8</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>9</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>10</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>11</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>12</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>13</td>
<td>69.3</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)</td>
</tr>
<tr>
<td>14</td>
<td>68.5</td>
<td>CWE-494</td>
<td>Download of Code Without Integrity Check</td>
</tr>
<tr>
<td>15</td>
<td>67.8</td>
<td>CWE-863</td>
<td>Incorrect Authorization</td>
</tr>
<tr>
<td>16</td>
<td>66.0</td>
<td>CWE-829</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
</tr>
<tr>
<td>17</td>
<td>65.5</td>
<td>CWE-732</td>
<td>Incorrect Permission Assignment for Critical Resource</td>
</tr>
<tr>
<td>18</td>
<td>64.6</td>
<td>CWE-676</td>
<td>Use of Potentially Dangerous Function</td>
</tr>
<tr>
<td>19</td>
<td>64.1</td>
<td>CWE-327</td>
<td>Use of a Broken or Risky Cryptographic Algorithm</td>
</tr>
<tr>
<td>20</td>
<td>62.4</td>
<td>CWE-131</td>
<td>Incorrect Calculation of Buffer Size</td>
</tr>
<tr>
<td>21</td>
<td>61.5</td>
<td>CWE-307</td>
<td>Improper Restriction of Excessive Authentication Attempts</td>
</tr>
<tr>
<td>22</td>
<td>61.1</td>
<td>CWE-601</td>
<td>URL Redirection to Untrusted Site (Open Redirect)</td>
</tr>
<tr>
<td>23</td>
<td>61.0</td>
<td>CWE-134</td>
<td>Uncontrolled Format String</td>
</tr>
<tr>
<td>24</td>
<td>60.3</td>
<td>CWE-190</td>
<td>Integer Overflow or Wraparound</td>
</tr>
<tr>
<td>25</td>
<td>59.9</td>
<td>CWE-759</td>
<td>Use of a One-Way Hash without a Salt</td>
</tr>
</tbody>
</table>
security it is necessary to analyze the building blocks on which they will be founded and so it requires a better abstraction of security problems. In order to accomplish this further level of abstraction, in the next chapter the analysis will enlarge its focus from single flaws or security issues to objects that it’s possible to define classes of security problems, by the adoption of security taxonomies.
CHAPTER 4
SECURITY PROPERTIES IDENTIFICATION

This chapter describes the process used in identifying the security related aspects of a system, which will affect applications running on top of it. Then a comparison and inspection will be carried out for each component, evaluating whether or not Ethos design and mechanism are able to provide a higher level of security and thus resulting in applications which are more secure.

4.1 Taxonomies

In order to find a meaningful set of security aspects to later perform the analysis, a reasonable approach could start from looking at taxonomies of security flaws and selecting items from them. Obtaining a list of security properties to analyze from selected items is figuratively, like performing a negation on selected entries. Even in this case, producing taxonomies of SH is not an easy task. There aren’t so many SH taxonomies; sometimes they are poorly realized or out dated (as denoted in (13)).

For this purpose have been selected two taxonomies and a list of security properties:

- Weber05 (13): this is a more recent work, it is an improvement of the well known Landwehr work (14), which is one of the most important in this area. This paper revises and improves the so called **Genesis dimension** in the Landwehr work, which classifies SH on the basis of how they are introduced into the system. The two principal trunks of this taxonomy are
Intentional and Inadvertent flaws: the former are bugs introduced in the development; the latter are flaws or side effects of components which are deliberately inserted into the system.

- Alsm96 (15): this taxonomy is more coarse-grained than the previous, SH are divided by big classes which are again distributed in two principal families Coding Faults and Emergent Faults.

**Coding faults** Synchronization errors, Condition validation errors

**Emergent faults** Configuration errors, Environment faults

- Solworth Technical Proposal: the list of security properties extrapolated from this proposal is not properly a taxonomy, but points out a series of elements which are fundamental problems to address when talking about system and application security. The form of this list is very similar to what will be ultimately used to carry out the analysis.

Here are presented the three taxonomies, in order Weber, Aslam and Solworth.

4.1.1 Validation and Scoring

In order to demonstrate the validity of a taxonomy, it should be possible to map the Top 25 list onto it. As remarked in (13), this implies that the taxonomy includes the items contained in the Top 25, but this doesn’t mean that it is sufficient to cover a comprehensive set of other important security flaws. A good mapping can be seen as a minimum requirement, the list that we are going to use has to map the vast majority of the currently perceived most important security problems.
<table>
<thead>
<tr>
<th></th>
<th>Malicious</th>
<th>Trapdoor</th>
<th>Logic/Time Bomb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non-malicious</td>
<td>Covert Channel</td>
<td>Storage Timing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inconsistent access paths</td>
<td></td>
</tr>
<tr>
<td>Intentional</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inadvertent</td>
<td>Validation Error</td>
<td>Addressing error</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Poor parameter value check</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Incorrect check positioning</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Identification/Authentication Inadequate</td>
<td></td>
</tr>
<tr>
<td>Abstraction Error</td>
<td>Object Reuse</td>
<td>Exposed Internal Representation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Asynchronous flaws</td>
<td>Concurrency (including TOCTTOU)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Aliasing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subcomponent misuse/failure</td>
<td>Resource Leak</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Responsibility Misunderstanding</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Functionality Error</td>
<td>Error handling failure</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other security flaw</td>
<td></td>
</tr>
</tbody>
</table>
1. Condition Validation Error

**Boundary Condition Error:**
- Did the error occur when a process attempted to read or write beyond a valid address boundary?
- Did the error occur when a system resource was exhausted?
- Did the error result from an overflow of a static-sized data structure?

**Access Validation Error:**
- Did the error occur when a subject invoked an operation on an object outside its access domain?
- Did the error occur as a result of reading or writing to/from a file or a device outside a subject’s access domain?

**Origin Validation Errors:**
- Did the error result when the object accepted input from an unauthorized subject?
- Did the error occur because the system failed to properly or completely authenticate a subject?

**Input Validation Errors:**
- Did the error occur because a program failed to recognize syntactically incorrect input?
- Did the error result when a module accepted extraneous input fields?
- Did the error result when a module did not handle missing input fields?
- Did the error result because of a field-value correlation error?

**Failure to handle exceptional conditions:**
- Did the error manifest itself because the system failed to handle an exceptional condition, generated by a functional module, device, or user input?
2. Synchronization Error

Race Condition Errors:
- Is the error exploited during a timing window between two operations?

Serialization Errors:
- Did the error result from inadequate or improper serialization of operations?

Atomicity Errors:
- Did the error occur when partially-modified data structures were observed by another process?
- Did the error occur because the code terminated with data only partially modified as part of some operation that should have been atomic?

3. Environment Errors

- Does the error result from an interaction in a specific environment between functionally correct modules?
- Does the error occur only when a program is executed on a specific machine, under a particular configuration?
- Does the error occur because the operational environment is different from what the software was designed for?

4. Configuration Errors

- Did the error result because a system utility was installed with incorrect setup parameters?
- Did the error occur by exploiting a system utility that was installed in the wrong place?
- Did the error occur because access permissions were incorrectly set on a utility such that it violated the security policy?
<table>
<thead>
<tr>
<th>Security Property</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>type safety</td>
<td>ensures that an object’s bit are consistently interpreted</td>
</tr>
<tr>
<td>memory safety</td>
<td>ensures that objects are one-to-one with memory locations</td>
</tr>
<tr>
<td>provenance</td>
<td>track the source of data</td>
</tr>
<tr>
<td>isolation</td>
<td>separates various parts of the system</td>
</tr>
<tr>
<td>unbounded precision</td>
<td>prevents loss of information due to too few bits to contain a value</td>
</tr>
<tr>
<td>aliases prevention</td>
<td>prevents confusion of what a name maps to</td>
</tr>
<tr>
<td>defined behavior</td>
<td>prevents undefined of implementation defined behavior</td>
</tr>
<tr>
<td>race free</td>
<td>avoids synchronization (including TOCTTOU) errors</td>
</tr>
<tr>
<td>deadlock free</td>
<td>system does not lock up</td>
</tr>
<tr>
<td>starvation free</td>
<td>every application makes progress</td>
</tr>
<tr>
<td>input vetting</td>
<td>all input are checked to ensure they are as expected</td>
</tr>
<tr>
<td>error examined</td>
<td>ensures that errors are not ignored</td>
</tr>
<tr>
<td>bounded resources</td>
<td>ensures that long running applications do not grow arbitrarily large</td>
</tr>
<tr>
<td>restartability</td>
<td>after a failure, application can be started (its data are available)</td>
</tr>
<tr>
<td>input not lost</td>
<td>the system can keep up with its inputs (e.g. key strokes not lost)</td>
</tr>
<tr>
<td>randomness</td>
<td>needed for cryptographic key creation and other uses</td>
</tr>
<tr>
<td>error recovery</td>
<td>recover from transitory errors (and some permanent errors)</td>
</tr>
<tr>
<td>access authorized</td>
<td>prevents access control violation</td>
</tr>
<tr>
<td>configuration correctness</td>
<td>prevents flawed system configuration</td>
</tr>
<tr>
<td>durable storage</td>
<td>prevents the loss of data</td>
</tr>
<tr>
<td>narrow interfaces</td>
<td>reduce communication to the minimum necessary</td>
</tr>
<tr>
<td>rollback</td>
<td>enables return to an earlier, consistent and safe state</td>
</tr>
</tbody>
</table>
To support the thesis that the Top 25 is not sufficiently complete a little example is provided: concurrency and all related topics like locking and threading are commonly known to be some of the most important and difficult to deal with issues in programming. A large number of security problems are generated from this category. In the Top 25 there isn’t a single error related to concurrency, this doesn’t mean that it is unimportant to take in consideration when performing the analysis.

First step to produce a reasonable evaluation is providing an adequate scoring metric:

**Strong Mapping:** when a flaw in the list is strongly mapped on the taxonomy, and so the category addresses in a proper way that particular flaw, it produces a +1 in the scoring

**Weak Mapping:** when a flaw is weakly mapped, the category is related but does not exactly address the flaw, this produces a +0.5 in the scoring. Weak mapping will be denoted by using square brackets.

**Not Mapped:** if there is no suitable mapping it produces a 0 score.

Note: if an item is strongly mapped more than twice, every further mapping of the same item in different categories, will result in a -1, this because heavy multiple mapping could be interpreted as a sign that boundaries of the taxonomy are too blurry and categories are overlapping.

Table VIII, Table IX, and Table X contain the mapping for each of the 3 taxonomies:
<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Improper neutralization of Special Elements used in SQL Command (SQL Injection)</td>
<td>Poor parameter value check</td>
</tr>
<tr>
<td>2</td>
<td>Improper neutralization of Special Elements used in OS Command (OS Injection)</td>
<td>Poor parameter value check</td>
</tr>
<tr>
<td>3</td>
<td>Buffer Copy without Checking the Size of Input (Classic Buffer Overflow)</td>
<td>Addressing Error</td>
</tr>
<tr>
<td>4</td>
<td>Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)</td>
<td>Poor parameter value check</td>
</tr>
<tr>
<td>5</td>
<td>Missing Authentication for Critical Function</td>
<td>Identification/Authentication inadequate</td>
</tr>
<tr>
<td>6</td>
<td>Missing Authorization</td>
<td>Identification/Authentication inadequate</td>
</tr>
<tr>
<td>7</td>
<td>Use of Hard-coded Credentials</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Missing Encryption of Sensitive Data</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
<td>[Identification/Authentication inadequate]</td>
</tr>
<tr>
<td>11</td>
<td>Execution with Unnecessary Privileges</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Cross-Site Request Forgery (CSRF)</td>
<td>[Identification/Authentication inadequate]</td>
</tr>
<tr>
<td>13</td>
<td>Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)</td>
<td>Poor parameter value check</td>
</tr>
<tr>
<td>14</td>
<td>Download of Code Without Integrity Check</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Incorrect Authorization</td>
<td>Identification/Authentication inadequate</td>
</tr>
<tr>
<td>16</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Incorrect Permission Assignment for Critical Resource</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Use of Potentially Dangerous Function</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Use of a Broken or Risky Cryptographic Algorithm</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Incorrect Calculation of Buffer Size</td>
<td>Addressing Error</td>
</tr>
<tr>
<td>21</td>
<td>Improper Restriction of Excessive Authentication Attempts</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>URL Redirection to Untrusted Site (Open Redirect)</td>
<td>Identification/Authentication inadequate</td>
</tr>
<tr>
<td>23</td>
<td>Uncontrolled Format String</td>
<td>Identification/Authentication inadequate</td>
</tr>
<tr>
<td>24</td>
<td>Integer Overflow or Wraparound</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Use of a One-Way Hash without a Salt</td>
<td></td>
</tr>
</tbody>
</table>
TABLE IX
MAPPINGS ON ASLAM TAXONOMY

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Improper neutralization of Special Elements used in SQL Command (SQL Injection)</td>
<td>Input Validation Errors</td>
</tr>
<tr>
<td>2</td>
<td>Improper neutralization of Special Elements used in OS Command (OS Injection)</td>
<td>Input Validation Errors</td>
</tr>
<tr>
<td>3</td>
<td>Buffer Copy without Checking the Size of Input (Classic Buffer Overflow)</td>
<td>Boundary Coincidence Errors</td>
</tr>
<tr>
<td>4</td>
<td>Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)</td>
<td>Input Validation Errors</td>
</tr>
<tr>
<td>5</td>
<td>Missing Authentication for Critical Function</td>
<td>Origin Validation Errors</td>
</tr>
<tr>
<td>6</td>
<td>Missing Authorization</td>
<td>Origin Validation Errors</td>
</tr>
<tr>
<td>7</td>
<td>Use of Hard-coded Credentials</td>
<td>Configuration Errors</td>
</tr>
<tr>
<td>8</td>
<td>Missing Encryption of Sensitive Data</td>
<td>Origin Validation Errors</td>
</tr>
<tr>
<td>9</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
<td>Input Validation Errors</td>
</tr>
<tr>
<td>10</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
<td>Origin Validation Errors</td>
</tr>
<tr>
<td>11</td>
<td>Execution with Unnecessary Privileges</td>
<td>Environment Errors</td>
</tr>
<tr>
<td>12</td>
<td>Cross-Site Request Forgery (CSRF)</td>
<td>Access Validation Errors</td>
</tr>
<tr>
<td>13</td>
<td>Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)</td>
<td>Input Validation Errors</td>
</tr>
<tr>
<td>14</td>
<td>Download of Code Without Integrity Check</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Incorrect Authorization</td>
<td>Access Validation Errors</td>
</tr>
<tr>
<td>16</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Incorrect Permission Assignment for Critical Resource</td>
<td>Environment Errors</td>
</tr>
<tr>
<td>18</td>
<td>Use of Potentially Dangerous Function</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Use of a Broken or Risky Cryptographic Algorithm</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Incorrect Calculation of Buffer Size</td>
<td>Boundary Coincidence Errors</td>
</tr>
<tr>
<td>21</td>
<td>Improper Restriction of Excessive Authentication Attempts</td>
<td>Environment Errors</td>
</tr>
<tr>
<td>22</td>
<td>URL Redirection to Untrusted Site (Open Redirect)</td>
<td>Input Validation Errors</td>
</tr>
<tr>
<td>23</td>
<td>Uncontrolled Format String</td>
<td>Input Validation Errors</td>
</tr>
<tr>
<td>24</td>
<td>Integer Overflow or Wraparound</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Use of a One-Way Hash without a Salt</td>
<td></td>
</tr>
<tr>
<td>Rank</td>
<td>Name</td>
<td>Mapping</td>
</tr>
<tr>
<td>------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>Improper neutralization of Special Elements used in SQL Command (SQL Injection)</td>
<td>Input vetting</td>
</tr>
<tr>
<td>2</td>
<td>Improper neutralization of Special Elements used in OS Command (OS Injection)</td>
<td>Input vetting</td>
</tr>
<tr>
<td>3</td>
<td>Buffer Copy without Checking the Size of Input (Classic Buffer Overflow)</td>
<td>Memory Safety</td>
</tr>
<tr>
<td>4</td>
<td>Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)</td>
<td>Input vetting</td>
</tr>
<tr>
<td>5</td>
<td>Missing Authentication for Critical Function</td>
<td>Provenance, [Access Authorized]</td>
</tr>
<tr>
<td>6</td>
<td>Missing Authorization</td>
<td>Provenance, [Access Authorized]</td>
</tr>
<tr>
<td>7</td>
<td>Use of Hard-coded Credentials</td>
<td>Configuration Correctness</td>
</tr>
<tr>
<td>8</td>
<td>Missing Encryption of Sensitive Data</td>
<td>Type Safety</td>
</tr>
<tr>
<td>9</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
<td>[Type Safety]</td>
</tr>
<tr>
<td>10</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
<td>[Access Authorized]</td>
</tr>
<tr>
<td>11</td>
<td>Execution with Unnecessary Privileges</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Cross-Site Request Forgery (CSRF)</td>
<td>Isolation, Access Authorized</td>
</tr>
<tr>
<td>13</td>
<td>Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)</td>
<td>Input vetting</td>
</tr>
<tr>
<td>14</td>
<td>Download of Code Without Integrity Check</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Incorrect Authorization</td>
<td>Access Authorized</td>
</tr>
<tr>
<td>16</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Incorrect Permission Assignment for Critical Resource</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Use of Potentially Dangerous Function</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Use of a Broken or Risky Cryptographic Algorithm</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Incorrect Calculation of Buffer Size</td>
<td>Memory Safety</td>
</tr>
<tr>
<td>21</td>
<td>Improper Restriction of Excessive Authentication Attempts</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>URL REDIRECTION TO UNTRUSTED SITE (OPEN REDIRECT)</td>
<td>Input vetting</td>
</tr>
<tr>
<td>23</td>
<td>Uncontrolled Format String</td>
<td>Input vetting</td>
</tr>
<tr>
<td>24</td>
<td>Integer Overflow or Wraparound</td>
<td>Unbounded Precision</td>
</tr>
<tr>
<td>25</td>
<td>Use of a One-Way Hash without a Salt</td>
<td></td>
</tr>
</tbody>
</table>
According to Table VIII, Table IX, and Table X above, the reported score for each of the taxonomy were respectively 13, 18, 18. Considering a 25 score as an ideal mapping the obtained results were sufficiently good to prove the validity of this approach, every taxonomy mapped more than 50% of items in Top 25. A short summary of this result is shown in Table XI:
<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>W</th>
<th>A</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Improper neutralization of Special Elements used in SQL Command (SQL Injection)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Improper neutralization of Special Elements used in OS Command (OS Injection)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Buffer Copy without Checking the Size of Input (Classic Buffer Overflow)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Missing Authentication for Critical Function</td>
<td>1</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>6</td>
<td>Missing Authorization</td>
<td>1</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>7</td>
<td>Use of Hard-coded Credentials</td>
<td>.5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Missing Encryption of Sensitive Data</td>
<td>.5</td>
<td>.5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
<td>.5</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
<td>.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>11</td>
<td>Execution with Unnecessary Privileges</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>Cross-Site Request Forgery (CSRF)</td>
<td>.5</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Download of Code Without Integrity Check</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Incorrect Authorization</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Incorrect Permission Assignment for Critical Resource</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Use of Potentially Dangerous Function</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Use of a Broken or Risky Cryptographic Algorithm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Incorrect Calculation of Buffer Size</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>Improper Restriction of Excessive Authentication Attempts</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>URL Redirection to Untrusted Site (Open Redirect)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>Uncontrolled Format String</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>Integer Overflow or Wraparound</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Use of a One-Way Hash without a Salt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOT</td>
<td></td>
<td>13</td>
<td>18</td>
<td>18</td>
</tr>
</tbody>
</table>
It can be noticed that taxonomies Aslam and Solworth map uniquely respectively 17 and 24. It can be also calculated a percentage of covered categories which are respectively 6/10 (60%), 4/16 (25%), 8/22 (37%). All these information help us to understand how good are these lists as starting points (the score) and how good are they at covering various aspect other than the Top 25 (coverage), factor which is, like explained before, evenly important to produce a meaningful list.

One last significant data is the list of SH which weren’t mapped in neither of the 3 taxonomies, these are 14,16,18,19,25. These together with half mapping and some additional reasonings are useful information to decide whether or not other that the existing categories are needed to represent some class of security properties which have not been taken in consideration.

4.1.2 Proposed Taxonomy

Starting from these three documents and iterative brainstorming has resulted in the following taxonomy. There is no claim of completeness in the detailing of errors, but rather the focus has been on the coverage of security problems categories that can turn into SH. The perspective from which the problem is approached in that of the programmer: this list summarize several possible misunderstandings which lead to errors that increase the probability of having a security hole in the final program.

Determining how many bugs turn into real SH, is not an easy task. It cannot be analytically computed. In order to have a reasonable measure of this ratio it would be necessary to test large amount of buggy code and see if the bug lead or not to a security hole. Again this is
not something that can be algorithmically determined. Even if there was an algorithm for checking whether a bug leads to a security hole, this could not deal with bugs that haven’t been discovered, so the ratio would continue to be imprecise.

The idea is looking the problem from the origin, eradicating the bug. If successful, this will reduce the probability to zero of that bug of turning into a security hole. Eradicating the bug cannot be algorithmically done, but trying to mitigate them as much as possible can certainly bring benefits in term of resulting security properties.
### TABLE XII

**PROPOSED TAXONOMY**

<table>
<thead>
<tr>
<th>Flaw</th>
<th>Sub-category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input validation</td>
<td>code injection</td>
</tr>
<tr>
<td></td>
<td>validation failures</td>
</tr>
<tr>
<td>Authentication</td>
<td>provenance</td>
</tr>
<tr>
<td></td>
<td>authentication failures</td>
</tr>
<tr>
<td></td>
<td>weak authentication</td>
</tr>
<tr>
<td></td>
<td>weak passwords</td>
</tr>
<tr>
<td></td>
<td>too many retry attempts</td>
</tr>
<tr>
<td></td>
<td>failure to use salt</td>
</tr>
<tr>
<td>Authorization</td>
<td>least privilege failure</td>
</tr>
<tr>
<td></td>
<td>mediation failure</td>
</tr>
<tr>
<td>Improper isolation</td>
<td>missing encryption</td>
</tr>
<tr>
<td></td>
<td>broken cryptography</td>
</tr>
<tr>
<td></td>
<td>exposed keys or passwords</td>
</tr>
<tr>
<td></td>
<td>memory isolation</td>
</tr>
<tr>
<td></td>
<td>non-memory resource isolation</td>
</tr>
<tr>
<td></td>
<td>covert channels</td>
</tr>
<tr>
<td></td>
<td>cold boot</td>
</tr>
<tr>
<td>Initialization</td>
<td>Type errors</td>
</tr>
<tr>
<td>Memory safety</td>
<td>storage leak</td>
</tr>
<tr>
<td></td>
<td>double free</td>
</tr>
<tr>
<td></td>
<td>bad pointer</td>
</tr>
<tr>
<td>Concurrency errors</td>
<td>race condition</td>
</tr>
<tr>
<td></td>
<td>atomicity</td>
</tr>
<tr>
<td></td>
<td>serialization</td>
</tr>
<tr>
<td></td>
<td>TOCTTOU</td>
</tr>
<tr>
<td></td>
<td>synchronization</td>
</tr>
<tr>
<td></td>
<td>deadlock</td>
</tr>
<tr>
<td></td>
<td>starvation</td>
</tr>
<tr>
<td>Arithmetic errors</td>
<td>integer overflow</td>
</tr>
<tr>
<td></td>
<td>floating point rounding</td>
</tr>
<tr>
<td></td>
<td>division by zero</td>
</tr>
<tr>
<td>Aliases</td>
<td>Crash consistency</td>
</tr>
<tr>
<td>Error handling</td>
<td>missing</td>
</tr>
<tr>
<td></td>
<td>incorrect</td>
</tr>
<tr>
<td>Logic and Syntax</td>
<td>infinite loop or recursion</td>
</tr>
<tr>
<td></td>
<td>wrong counting</td>
</tr>
<tr>
<td></td>
<td>assignment in place of equality test</td>
</tr>
</tbody>
</table>
4.1.3 Definitions

In order to clarify and describe the context of the flaws that will be later analyzed, this section provides definitions for taxonomy terms reported in Table XII. Previous taxonomies were produced by thinking from the point of view of a programmer, so some definitions can be related to this particular perspective.

Input Validation issues include the set of missing or improper validation of any kind of data coming from and external to the program environment. These include storage, networking, and user input. Input validation is a tremendously difficult task because is difficult and in the end not completely possible to find a formal way to analyze user input, but missing or weak validation is one of the major source of several of the most dangerous attacks.

Authentication flaws involve a series of problem related to missing, improper or flawed management of the mechanisms used to establish the identity of a user. Authentication is necessary to enable later checks against authorization policies, to determine if access to specific functions and data is allowed. In the worst case, authentication can be missing, leaving the user free to declare the identity that he best like, or improperly designed giving the opportunity to bypass it. Another dangerous flaw is when the implementation does not behave as intended.

Authorization flaws encompass all problems related to defining whether or not some entities inside and outside the system is granted access to functions, information or resources. This is strictly correlated with authentication and depend in part on a correct identification of the user. Typical flaws are missing or incorrect authorization mechanism which
are respectively not designed or poorly designed and so allow access to information and services which are not intended to be used to some entities. We could say that there is a mediation failure between the user and system resources which should have been properly guarded. Flawed implementation of the authentication mechanism can result in dangerous behaviors too. One particular class of authorization problems is violating the least privilege principle, which states that the minimum amount of privileges to complete a task should be provided.

**Improper Isolation** flaws comprehend inadequate handling of sensitive data, in which there are inadequate mechanisms to prevent attackers from accessing this sensitive information. This category has a very broad spectrum that include issues that space from exposed keys and use of broken cryptography to missing sandboxing and covert channels.

**Initialization** flaws refer to that set of problems related to variables set up before executing a program for the first time or for the nth iteration. Incorrect setting of values during the startup can produce unexpected behavior in program flow, preventing specific part of the program to work correctly or lead to other flaws in the taxonomy. Some software is written to restart execution and providing services without actually terminating the process. Missing an appropriate cleaning of data between one cycle of execution and the next one, can cause serious problems of confidentiality and integrity of data, private information can be disclosed to the next client.

**Type Errors** flaws are related to all kind of data misuse and violation of type safety. Explicit violation of type coherency like type casting, direct memory access without suitable
functions (for example in strings), can result in major security problems if not carefully handled. This flaw class includes also the cases in which there is no type enforcement and not even checking. Dealing with data without being sure that are the expected type cannot be exploited by attackers to cause the program execute arbitrary malicious operations.

**Memory Safety** these kind of flaws embody all those problems related to incorrect or dangerous memory management. They can result in violation of security property like integrity and confidentiality, in the case of direct access to memory location whose access was not allowed, but also in availability problems in the case of storage leaking. Incorrect or missing releasing of memory area to the system, results in a continuous increasing of the amount of resources used by the program. This problem often affect non-garbage collected memory, in which memory management is the responsibility of the programmer, leakage in a program can lead to availability problems because no more resources are available or the program has to be restarted in order to return in a proper functional status.

**Arithmetic Errors** are all the flaws caused by a failure to understand and use correctly the arithmetic properties of the computer. The misunderstanding can be based on very low levels details of the architecture or language, or in dealing with compatibility on different system and representation issues. Errors related to overflows and approximation can lead to unintended path of execution.
**Aliases** flaws are all that implementation and design behavior that allow two different names to identify the same object, location, service, etc. Aliases decrease the level of clarity in the system allowing multiple ways to perform the same task or to name the same resource.

**Concurrency Errors** this class involve all those kind of issues related to managing explicit parallelism in programs. A very large variety of problems have the common root of being related with concurrency, but they are characterized from misunderstanding of different practical aspects of the problem.

One source of failures is composed by all problems related to wrong or not careful managing of concurrency, for example shared memories issues, missing synchronization, missing locking of critical section and key variables, improper serialization of some operations, TOCTTOU etc. Another source could be related to availability issues due to ineffective management of resources by the system that eventually turns out in situations like deadlocks and starvation.

**Crash Consistency** is the set of errors resulting from leaving the environment in an unclean state after an execution terminates, as in a crash. The status of the program is not independent from the termination status of the previous execution.

**Error Handling** flaws derive from an improper or missing handling of errors during a program execution. Unexpected condition can cause error in the program, producing an unusual behavior or path of execution. Programmers often don’t really focus on these paths because they are less likely to happen, sometimes they even ignore the existence of a possible path that can be triggered by an error. Flaws could result from a not handling
of the error or an incorrect handling, like for example not proper signaling to the user or propagating the error to other components.

**Logic and Syntax** are an improper use of program semantic mainly due to lack of attention. Logics misuses of control flow structures often translate in unreasonable paths of execution which where not intended by the developer. Syntax errors derive from an incorrect understanding of the language notation or its improper accidental use, for example performing an assignment instead of an equality test or the classic off-by-one error.
CHAPTER 5

PREVENTION AND MITIGATIONS

This chapter builds on the results obtained in the previous chapter. Starting from the taxonomy in Table XII, every main category will be dissected and possible prevention or mitigation techniques will be enumerated. This step is necessary in order to give a summary of the possible ways of reducing bugs and consequently security hole density. In Chapter 6, the enumeration of possible prevention and mitigation techniques will be used to bring concrete examples of how the computer science community has tried to cope with these security problems and what kind of solutions they have effectuated. Ultimately in Chapter 6 will be shown what kind of techniques and trade-offs have been chosen for the Ethos operating system.

5.1 Input Validation

Input validation is a very broad area, a useful classification of prevention and mitigation according to (16) accounts the following two classes:

- **Validation Checks**: ensures that the data is consistent with type, correct syntax, within length boundaries, contains only permitted characters, or that numbers are correctly signed and within range boundaries. This category of input validation is very difficult to handle properly due to the broadness of its broad range. It is almost impossible to write tools which are in general effective on every kind of input, so basically people usually
write ad-hoc tools. However is still possible to design mechanisms with some degree of
generality.

• **Business Rules:** Ensure that data is not only validated, but business rule correct.

  This involves applying a set of specific policies or behavior in response or in prevention
  of certain actions associated with some events. For example, interest rates fall within
  permitted boundaries.

In the original classification from OWASP there were three categories, we believe that
**Integrity Checks** even though is related to ensuring properties on input data, like for example
that they’ve not been tampered, is a problem that belongs to the Authentication domain and
will be discussed in detail in section 6.2.

### 5.2 Authentication

Authentication is a key factor in computer security. It’s the foundation of every secure
architecture, in fact identifying the subject who is operating on the system defines what is
allowed or not to do, according to the specified configuration. Without identification it is not
possible to assign privileges and resources in a meaningful way. Because this operation is so
important and on its outcome will rely several other security services, authentication is one of
the most stressed categories in term of attacks.

Authentication is performed on the basis of three possible domains: **what you know,**
**what you have,** **who you are.** What we list as preventions or mitigations for attacks are
in the end authentication techniques and their relatives improvements to stand against known
vulnerabilities.

They can be subdivided in four broad families:

**Physical Based** This type of authentication mechanism comprehends all kinds of biometric techniques like fingerprints, voice matching, retina scanner, handwriting or keystroking. All these methods can be attacked in several ways and they are not equally strong: for example, retina image is much more accurate than using handwriting recognition. Attacks of physical-based authentication are less related to the computer science sphere and more connected to the personal and human related one.

- **Coercion and substitution** in order to mitigate the risk of physical abduction of the privileged subject aimed at taking over the system, adequate countermeasures have to be used like arranging proper (physical) protection of the person of interest and of those which have leverage over the person such as his family, close friends or relatives.

**Crypto Based** Several authentication advanced techniques make use of cryptography in order to guarantee high level of security. Like other sensitive information, keys can be exposed, there are various possible techniques which will be described from the strongest to the weakest:

- **Distributed Key Scheme** the key is “broken” in pieces and distributed among different people and location, in order to reconstruct the secret a minimum amount of people over the total is necessary, this is a so called threshold scheme. Strength
of this technique is that it doesn’t rely only on a single source, so it is both stronger against attacks (stealing, insubordination) and flexible in term of recovery in case one or more key holder are missing. Drawback is the difficulty of properly design the threshold and decide the key owners in such a way that is difficult that a subset of them can secretly agree to betray the others and that unifying the key is a task sufficiently easy to handle.

- **Multi-Key repository with Master key** using the same key for different services can exposed it to attackers: in order to mitigate this problem one way is having a set of keys properly locked up with a master strong key which has to be carefully kept in a safe place. This technique offer great flexibility in using a wide variety of different keys (at most one per service), drawback is that this scheme is as strong as the master key and the security level of its management.

- **Single Key** this is the most basic approach that involves cryptography, using a single key for authenticating gives much more strength than passwords or passphrase in proportion to the key length.

**Device Based** This subcategory is not mutually exclusive with the previous one: some authentication devices make use of cryptography. This technique covers every type of physical device which is used for the specific purpose of authentication. Often such devices are used in combination with other techniques in order to strength the authentication mechanism. Example of these devices are Keys, Cards (Smart or Dumb), Tokens etc.

Possible attacks and relative mitigations are:
• **Stealing** this represent acquiring the device from its owner, countermeasures range from restricting access to the device to limited number of people and to proper training of owners.

• **Copying** in this case the device is cloned in order to steal the secret while avoiding the owner to realize that it has been stolen and replacing the security mechanism. Special cases are non-destructive attacks, used in cases where the device is built in such a way that is very difficult to physically read and clone the secret, so several techniques can be used to reveal the information without actually destroying the device, for example power, timing and radiation attacks. Possible mitigation are using tamper resistant technologies and anti-counterfeiting techniques in order to detect whether the device it is a copy or if an original one has been copied.

**Password Based** This is probably the most common and cheapest way of performing authentication. Due to its simplicity, it’s also the most attack prone. Since passwords are usually typed by humans into systems, their security is affected by their tendency to commit errors (password reusing, choosing simple or personal related passwords etc.), which greatly weaken the effectiveness of this mechanism. Principal attacks and relative mitigations are:

• **Guessing** especially when passwords are not carefully chosen. Short, easy, and obvious passwords, make it very easy for an attacker to guess on the basis of personal knowledge of information about the subject. Mitigations include mandatory
requirements on the password creation, like character set, minimum length, password
duration, and black lists.

- **Over the shoulder** one of the most common way of obtaining a password is reading
it without being noticed. This is a completely human factor based on the lack of
attention of the subject who is typing the password or has written it down on a note.
Mitigations actively involve the subject who has to be trained to perform a series
of adequate actions (like scheduled password replacement) and follow certain best
practices when dealing with his password.

- **Automated online attacks** this attack is based on repetitively attempts of guessing
the password using automated techniques that exploit softwares and dictionaries in
order try a large amount of password on the system. This attack can be mitigated,
since it is performed on an online system, by limiting number of attempts per user
and carefully detecting and controlling suspicious retries rate.

- **Reading password files** usually systems store user password in a file. In order to
mitigate this threat it is important not to store passwords in plain text, but saving
them securely using adequate cryptographic techniques.

- **Automated offline attacks** it consists in downloading the password file and trying
to crack it offline, in order to bypass possible limitation on the number of attempts
and also to not arise suspicion. Possible mitigations include setting up properly
mechanisms to prevent the file to be copied and using strong cryptographic tech-
niques to sufficiently protect it against offline attacks.
• **Spoofing attacks** these attacks make use of dummy login windows or screen in order to trick the user convincing him that it’s actually the real login form. Mitigation is the arrangement of *trusted path* between the system and the user interface in order to ensure that the user is actually communicating with the system and not with other entities.

• **Eavesdropping attacks** make use of collecting data on physical properties (acoustical properties, key logging, fingerprints etc.) of the devices used to insert the password in order to later reconstruct the sequence of characters. Mitigations includes properly restricting physical access to facilities and machines and inspections for signs of intrusion or alteration of devices in use.

### 5.3 Authorization

Authorization is the very next step after Authentication, it encompasses all the decisions regarding what a subject allowed to do on the system. As in the previous category, it is clear that this step is crucial in order to ensure adequate security properties. This is a particular category, the only strong mitigation is providing an Access Control System (ACS). Access Control is involved at different levels in the system, as described in (17). It can be implemented at *Application level* like in a modern business site, or as *Middleware service* like Database Management System (DBMS)s in order to regulate transactions. There are also lower level implementations of ACS traditionally built at *System level* with the support of *Hardware level* security oriented services (for example, address access protection).
Even though there are a lot of implementations at higher levels, usually the focus is on facilities provided at system level, in 6.3 will be provided further explanations for this point. “Going all the way back to early time-sharing systems we systems people regarded the users, and any code they wrote, as the mortal enemies of us and each other. We were like the police force in a violent slum.” - Roger Needham, this quote summarize quite effectively the important fact that moving up from lower to higher level access control mechanisms become more complex and less reliable.

There are several families of ACS and will be briefly described in the following paragraphs. Like in every categorization it is always possible to create new structures by modifying or combining existing ones, listed here are only the most important and well known.

- In the **Discretionary Access Control (DAC)** model, access to resources is based on user’s identity. A user is granted permissions to a resource by being placed on an Access Control List (ACL) associated with resource. An entry on a resource’s ACL is known as an Access Control Entry (ACE). When a user (or group) is the owner of an object in the ACL model, the user can grant permission to other users and groups. The ACL model is based on resource ownership. A possible categorization of DAC systems is the one proposed in (18):

  **Strict DAC** the owner can grant and revoke ordinary privileges such as **read**, **write** and **execute**.

  **Liberal DAC** the owner can also delegate and revoke grant authority to other users.
**DAC with Change of Ownership** the owner can also transfer the ownership of an object to other users.

**DAC with Revocation** the owner can revoke permissions, there are 2 variations: dependent, only the granter can revoke, or independent, whoever has the authority can revoke access from users.

- In the Mandatory Access Control (MAC) model, users are given permissions to resources by an administrator. Only an administrator can grant permissions or right to objects and resources. Access to resources is based on an object’s security level, while users are granted security clearance. Only administrators can modify an object’s security label or a user’s security clearance.

**Lattice-Based** access control systems relies on the algebraic structure of a *Lattice* in order to define levels of security of various objects. Every user log in at a certain clearance level and can access and modify objects which have equal or lower security level.

**Type Enforcement (TE)** implements access control through an access matrix where *domains* are classes of trusted programs (or at least one) and *type labels* are associated to group of objects (at least one). Privileges are then recorded in the matrix according to the combination (row,column) and mandatory define access policies on those resources.
• In the **Role Based Access Control (RBAC)** model, access to resources is based on the role assigned to a user. This is technically a form of non-Discretionary access control, but it’s commonly described as a primary access control policy together with DAC and MAC (19). In this model, an administrator assigns a user to a role that has certain predetermined right and privileges. Because of the user’s association with the role, the user can access certain resources and perform specific tasks. RBAC is also known as Non-Discretionary Access Control. The roles assigned to users are centrally administered.

DAC and MAC can be used in combination in order to provide a better suited security configuration for the system environment. It is up to the designer defining the roles that each ACS has to play on the totality of the security decisions and when and how one should be priority with respect to the other.

5.4 Improper Isolation

• **Sandboxing:** is a security mechanism intended to separate execution of running programs. By providing a strictly controlled set of resources and inhibiting or severely restricting privileges such as permanent memory access and network access, a sandbox isolates the program from surrounding environment. Interaction with other programs or system resources is tightly mediated by the sandbox.

• **System-wide Encryption:** this mitigation prevents unintended users from reading sensitive data. It consists of applying encryption techniques to every physical stored data or flow of information over the network. This protection is as strong as the algorithm used,
proportional (ideally) to the key space. The drawback is decreased data throughput due to additional load introduced by encryption.

- **Sensitive Information Encryption**: it differs from the previous mitigation because encryption is only applied to selected sensitive file and secret keys. It prevents secret data from being read by an attacker, has lower performance impact than system-wide encryption because the smaller amount of data encrypted. The drawback is that since the encryption is not mandatory, the programmer must define what to encrypt; errors in so doing can expose sensitive data.

- **Covert Channel Analysis**: this consists of series of code (or network) analysis techniques in order to detect covert channels. For example, some of these use approaches based on information theory concepts such as entropy, while some other query the network infrastructure in a particular way in order to evidence the possible presence of a covert channel. Analysis and careful design can massively reduce covert channels, but not completely eliminate the possibility of having one in the system.

- **Process Isolation**: one of the most powerful mechanism to enforce a reasonable level of isolation is the one adopted in memory management by modern operating systems. Through a set of hardware and software techniques, like for example address space virtualization, the system is able to prevent one process to write or see the address space of another process. The concept is very similar to sandboxing, it’s been treated in a separate section, because usually process isolation refers to the deeply system-level integrated
mechanism adopted by OSs in order to define a clear separation between processes that are currently running.

- **Information Hiding**: correct application of information hiding principles in software design, is a powerful technique of isolation. It allows boundaries to be defined and strongly specify access methodologies and paths to components and data. Downside is that its application and correct implementation completely rely on programmers.

- **Hardware Isolation**: this category depends more on physical mechanisms than the other, it includes all sets of mechanism to physically prevent or restrict access to a specific hardware component. For example secure facilities, ciphered memory, combination locks, etc. It is clearly differentiated from the other categories in that it can’t be implemented by OS or software and has to rely on external additional external devices or facilities.

### 5.5 Initialization

- **Path-based initialization**: to ensure for each variable $v$, for any possible execution path in the program, the program initializes $v$ before $v$’s value is used. This is performed with compile-time checks, based on use-definition pairs (20). The drawback of this technique is that this is a quite strict requirement because in some case it could be a mere explicit re-assignment of a default value. Moreover this put an additional computational load on the compiler.

- **Explicit initialization**: ensure that at creation time, each variable is given a fully defined value. This can be enforced by the compiler. Drawbacks of this are similar to the
previous: The explicit initialization might be meaningless. The additional computational load on the compile should be lighter than the previous case, since an exhaustive coverage of all execution paths is not required.

- **Default value initialization** this is the simplest and weakest solution. It requires the system or the language to explicitly initialize value of declared variables to a pre-defined default. Drawback is that this value could sometimes be meaningless for the logic of the program, but in any case is better than an unpredictable random value.

Common limitation of all the previous cited techniques is that this kind of reasoning applies to a local scope, so is not possible to track previous history of a variable that comes from outside. Tracking possible flows of data inside the program could not be feasible at compile time and in any case too expensive in term of computational resources to be performed at runtime. However even if is not possible to ensure a meaningful initialization of an external variable with regard to the current procedure, at least we could predicate that if one of the three mechanisms described before is applied on all the procedures of the program, the variable should come initialized at some sort of default value meaningful for the context in which it was declared.

5.6 **Type Safety**

- **System-Level Type Enforcement**: a type system which is implemented at the system level provides a rigorous type enforcement to all the data that flows through the system. It has to decouple typing of data from specific APL, and yet be consistent with that APL. The drawback is the intrinsic difficulty of the problem of finding a suitable mechanism to
provide a general and the same time customizable typing interface to different APL that could be installed in the system.

- **APL Level Type Enforcement**: this technique consist of adopting APL that implement a type system. Preferably one that provides strong typing mechanisms which are able to check and inform the user about forbidden type assignments or manipulation. The drawback is that the type system is strictly related to the language used and so to the code that has to deal with the type. Moreover some languages provide type checking but only at runtime, due to limitation imposed by the language design.

Both these techniques have to be able to guarantee data assignment and manipulation consistency at least from the point of view of the type associated with them.

5.7 **Memory Safety**

- **APL Variable Range Bound**: this consists in adopting a language which gets rid of the possibility of out-of-range access to variables by design. The chief type of out-of-range is buffer overflow, but pointer arithmetic is also a problem. This kind of mitigation is usually quite effective, however there are some particular case in which requires particular attention:

  - In almost every high-level language is possible to call native code for performance purposes, native code is usually written in a language which is not variable range bound.
– The code of the runtime—whether virtual machine (if the language requires a VM to operate) or not, it is usually written in a language that is not variable range bound—this can result in indirect violations.

– The Interpreter or JIT compiler may not working properly and generates intermediate level code which doesn’t check appropriately boundaries of variables.

• **Incremental Storage Allocation**: providing mechanisms which are able to dynamically allocate extra-storage to detect variable range errors. For example, canaries. (21) Major drawback of this technique is that, if not explicitly checked, writing huge amount of data out of the original boundaries can arise serious availability problems due to resource saturation.

• **Randomized Allocation**: this mitigation randomizes data layout, to greatly increase attacker effort needed to construct a suitable chain of events that allow the execution of malicious code through memory exploits. Alternatively, the technique can be used to decrease the effectiveness of an attack.

• **Disable Execution on Memory Areas**: this consist of explicitly marking some memory areas as not containing executable machine instructions. This mitigation partially address execution of malicious code loaded through buffer overflow exploits.

• **Inhibiting Direct Memory Access**: enforcing at system and APL level memory access restrictions which cannot be bypassed. Having the possibility of directly accessing memory in various way (for example pointer arithmetic) has been proven to be extremely
dangerous. Preventing the programmer to perform accesses which are not mediated by PL abstraction and OS supervision by removing/ modifying features with these vulnerabilities, translates in a strong mitigation of memory safety issues.

- **Garbage Collection:** this consist of providing facilities that are able to safely deallocate resources no longer in use. This kind of mitigation removes the problems related to access to memory locations which no longer holds the supposed data anymore (the famous *dangling pointer problem*). In the best case, this kind of violations can result in simple access to random or outdated information. But sometimes this access enables secret data to be read from previously allocated variables. One known drawback of garbage collection is execution performance overhead.

5.8 **Concurrency**

Developing and debugging in a concurrent environment is without any doubt one of the hardest task for a programmer. Dealing with highly concurrent applications (which are often distributed) is difficult, and often programmers have shown an inadequate level of experience in approaching such a problem. A large number of problems are concurrency related, like using outdated information, system deadlocks, serialization error on some operations, non-convergence of distributed data to a consistent configuration, synchronization with a precise timing, etc.

System are more concurrent than in the past at every level: machine, Virtual Machines (VMs), and multi-core. Programming on concurrent systems is hard, interactions between various code fragments running in a non-deterministic way are not obvious and influenced by
a lot of different factors. Concurrency problems, for the purpose of this analysis, can be seen as grouped in two big clusters, the ones related to Race Condition and the ones related to Deadlock.

The first group contains all that kind of issues which deals with data integrity, and how it can be compromised by the non-deterministic execution of multiple logical programs without a proper management of shared resources. In the first category fit all those flaws like wrong serialization, Time To Check To Time Of Use (TTCTTOU), atomicity problems etc.

The second group deals with some issues which are derived from the fact that there is a concurrent model of execution and different programs are competing for shared resources. This usually brings the system in a state in which activities slow down or at worse completely stop. In this case the prominent aspects are availability concerns. Reaching a state of deadlock, or starvation (both examples of this second category) is usually a consequence of a incorrect management of shared resources, but can be triggered also by one of the elements seen in the first group. These, in fact, can generate a state of high instability because the programmer hasn’t considered the possibility of a given sequence of events to occur. From this unstable state is very easy to move towards a lock condition.

Preventions and mitigations will be provided according to the previous described grouping criterion.

Race Condition

- System synchronization mechanisms these kind of primitives are usually provided by the operating system in order to accommodate and manage various forms of shared
memory. This memory is necessary in order to make possible communication between concurrent executing tasks. Finally a set of additional mechanisms is provided to secure this shared information typically according to the concept of resource locking.

- **APL synchronization mechanisms** generally this kind of mitigation consists in higher abstraction offered by modern programming languages. APLs offer advanced mechanism for managing concurrent execution, integrated in the semantic of the chosen language. This offers more sophisticated tools and relieve the programmer from dealing with the actual low level system management of concurrency. Clearly libraries provided by languages rely on the implementation of these high level mechanisms through the previous cited system synchronization mechanism. If the language is supported in different OSs the developer has to familiarize only with one set of functions and one semantic in order to deal with concurrency: the one offered by the language, regardless which OS is using.

- **Distributed synchronization mechanisms** this kind of mitigation is about operations so called transactions, which are using and modifying shared data while running od different systems (which can also be running in physically separated machines). These mitigations guarantee consistency of the shared data and verification of correctness over performed operations (read and write) through the enforcement of specific polices on resource management. Such mechanisms are generally more complex than the others because have to face additional issues regarding possible communication problems between remote systems, lack of timing synchronization etc.
• **Atomic operation ensemble** this mitigation provide set of ad-hoc atomic primitives composed by multiple operations like check+use, in order to cope with problems related to context switching like for example TTCTTOU issues.

**Deadlock** According to Coffman in order for a deadlock to happen, four necessary condition have to subsist:

- **Mutual Exclusion** a resource can be exclusive to a process
- **Hold and Wait** a process that holds resource can ask for other held by other process
- **No Preemption** resources can’t be removed from a process until it has finished using them
- **Circular Wait** two or more processes mutually are involved in waiting for resources held by other processes in the chain

• **Prevention** these prevention techniques enforce system policies that completely remove the possibility for a deadlock to occur, usually by removing one or more of the previously described necessary conditions.

• **Avoidance** this mitigation doesn’t remove the possibility for a deadlock to occur but, employees algorithms often based on information about the resource allocation for processes, in order to technically avoid that deadlock situations arise.

• **Detection** this mitigation detects a deadlock while the system is running and actively recoveries it from that state by applying some mechanisms to disengage the lock, usually by aborting one element in the waiting chain.
5.9 Arithmetic errors

- **Variable Size Integers** introducing a mechanism that extend the size of the integer in case of overflow, allowing the system to proper handle bigger numbers. This mitigation basically eradicates the possibility of having integer overflow as long as the number fits in the memory of the computer, (actually the hypothesis of exceeding the total memory for a single number is quite unrealistic), but introduces another problem. If this extended number are used to index array location or other indexes which are physically related to the machine architecture, values that are greater than the word size are very likely to result in errors or unexpected behaviors.

- **Formal Methods** analyzing the program source code using formal methodologies can address only a subset of the possible overflow situations: the one in which the value of the variables involved is independent from external input. In this way the execution path can be traced back from the condition to be evaluated and, if present, the overflow can be reported to the programmer.

- **Automatic Check Insertion** automatically inserting the standard conditional code on every possible overflow situation during code compilation, translates in basically addressing each possible overflow in the program by preventing unexpected behavior and so flows of execution. Drawback is that since this is a standardized automatic checking, that is not meaningful and coherent with the program semantic, management of the overflow situation is just a simple passive reaction to signal the problem or in some way drive the routine or the program to a controlled termination status.
• **Exception on Overflow** this is a very basic solution, whenever an overflow situation occurs it is signaled in an appropriate way to the programmer who will modify or correct the code in order to properly take care of it.

Automatic Check Insertion and Exception on Overflow have the common drawback that dangerous overflow situations are discovered only if they actually happen during a test. If due to an unlucky coincidence, data used for testing does not create an overflow situation, the problem, which may be present, won’t be nor signaled or discovered.

5.10 **Aliases**

• **Remove Symbolic Links:** eliminate symbolic links. Attackers often use symbolic links to set up familiar environment in order to mislead the user to perform or authorize malicious actions.

• **Restrict use of Symbolic Links:** this mitigation is a soft version of the previous one, instead of completely eradicating symbolic links from the system, it aims to better control their presence. Various criteria could be limiting in number, limiting in type, providing a clear spot where is possible to see every wiring associated to a symbolic link and others.

• **Canonical Form:** providing a well defined canonical form for symbolic links that helps to perform analysis and comparison in order to establish whether or not some of access or operation is malicious. For example prefix comparison in order to see if some provided link match up with intended environment of operation.
5.11 Crash Consistency

- **Clean Up Mechanism:** performs automatic cleaning of corrupted resources in case of unexpected termination, in order to restore a previous stable state. The drawback is that in order to correctly establish which resources has to be reset or cleaned up, additional bookmarking has to be kept by the system and the user has to explicitly signal such resources.

5.12 Error Handling

- **Mandatory Handling:** use an APL that implements mandatory error handling in every situation which can cause of significant errors. The problem with this approach is that discerning between error that has mandatory to be solved or not could be a difficult process not often straightforward for language designers. Moreover part of the burden remains with the programmer who must write code to handle the error and also create higher level errors when appropriate.

- **Error Signaling:** providing a system accepted way of signaling errors to support system, resulting in an automatic response such as termination and restarting and forced resource deallocation.

5.13 Logic and Syntax Errors

- **Cleaner Semantic:** this mitigation consists of providing cleaner and less error prone languages, with a more understandable semantic and less ambiguity in the symbology used (for example in the case of `=` vs `==`).
• **Bounded Properties**: providing mechanism to enforce upper bound on certain properties like for example the number of iterations in a certain loop.
CHAPTER 6

APPLICATION OF THE TAXONOMY TO THE ETHOS CASE

In this chapter, for each category of the proposed taxonomy in Table XII some practical examples will be presented of preventions and mitigations developed and used by the computer science community. Along with this will be presented the approach followed by Ethos operating system, describing its design and the resulting impact on security.

Every kind of design decision which has in the end defined the behavior and the philosophy adopted by Ethos regarding security properties, has to be evaluated considering the intrinsic difficulties of the problem. For each type of critical decision in every design process there are several aspects, playing a concurrent role in the solution of the problem, to be taken in consideration at the same time. This happens also in the field of computer security. Addressing these problems requires a continuous trade-off in order to satisfy every demand in the best possible way, even if it is less effective than an ad-hoc solution which doesn’t consider any other aspect.

6.1 Input Validation

Concrete approaches to address input validation, range from languages with tighter semantics and rules to framework specifically developed to serve for this purpose. This class comprehends a very broad set of possible attacks and flaws, so it’s difficult to point out a generic approach which is flexible enough to be widely applicable to solve all the possible prob-
lems. Here will be presented some examples of these approaches, with focus to some specific issues related to the categories defined in section 5.1.

6.1.1 Community approaches

- Ensuring that integrity checks are effectively performed should be one of the fundamental activity when securing a software, whether is web or local based. Relying on corrupted data or parameters can completely compromise the security of the whole application. In this direction, plenty of libraries and framework provide a rich set of cryptographic primitives to support common operations of encryption, hashing etc.

Programming languages give built-in support for the most common cryptographic operations, for example in the case of JAVA (22) and .NET (23) which can be used locally or remotely (like in Active Server Pages) depending on the implemented architecture. In addition to official support or in case of lacking there is a wide variety of community implementations of cryptographic libraries written or wrapped for the majority of the programming languages, examples are JASYPT, PyCrypto, Crypto++, Bouncy Castle etc.

Even in the case of completely web based languages there is a strong support for input validation, for example in the case of JQuery, the most famous Javascript library, there are several plugins to address integrity validation operations (24). Despite the wide support, there still are some problems: the effort required to correctly handle and use crypto facilities is completely on the programmer, who can be lazy or unexperienced. Moreover such a wide variety of libraries to accomplish the same purpose, sometimes for the same
language/runtime (like for example Microsoft cryptographic technologies which include CryptoAPI, Cryptographic Service Providers (CSP), CryptoAPI Tools, CAPICOM, WinTrust etc. (25)), can easily generates confusion and also misunderstanding between groups of developers which are not carefully coordinated on the use of same tools.

- In order to prevent SQL Injection some features have been arranged in programming languages to provide facilities that help developers to mitigate the problem. The following Validation techniques are offered from languages such as PHP, JAVA,.NET, Perl etc.

**Parameterized Queries with Bound Parameters** and **Parameterized Stored Procedures** (26) rely on careful building of the query to avoid SQL Injection: the idea behind this is to use placeholders for part of the query which are specified by variable input. These markers will be replaced by the actual query parameters which are passed together with the query as argument. The structure of the query is analyzed by the DBMS and an execution plan is created, later the placeholders are replaced with parameters. Even if these contains malicious code the structure of the query should not be altered. This is true in the majority of cases but by exploiting some specific DBMS features, it is still possible to find a workaround. Difference between the two is that in the second case the code of the query to be parametrized is stored directly in the database. Choosing between these solutions has to be done according to specific requirements and benefits offered by one implementation or the other.

Another degree of prevention is provided by validating the input provided by the user before actually generating the query: **Escaping All User Supplied Input** according
to a particular character scheme. This process has to find the right balance between expressiveness and security. Parameters can be checked for structure correctness against a **White List** of allowed well defined patterns, the strongest way of carrying out this process is comparing provided parameters with a list that enumerates all the allowed input parameters.

The effectiveness of these techniques can be very high if they’re used properly. The downsides are that the burden of all the validating process is completely on the programmer which can accidentally skip or improperly implement the checks. For example, the programmer might include dynamical query generation inside a stored procedure, which most likely will compromise the final security level. In addition to this the second major issue is that, even if the DBMS or the programming language provides the facility to secure query execution, their usage is not mandatory so programmers can again accidentally or voluntarily undermine the robustness of the whole application by simply skipping the proper validation.

- Several approaches in the field of *data validation* are currently under development, see (27). Supporting validation with tools helps programmer because they can concentrate more on security policies and less on the actual implementation. In their work, Munwar et al. propose a framework to apply what they call “Security-Oriented Program Transformations” to specific sensitive part of the code in order to implement security checks and validation. The work is described as part manual and part automatic, but it is also mentioned that there exists a variety of similar tools for automatically detecting validation
issues in the code, the problem is that these tools don’t provide a solution so programmer intervention is still required to check implementation.

A good number of tools don’t actively solve the security issue, but instead help developers to detect them for later fixing for example in Brinhosa et al. (28) is provided a framework based on XML property specification to analyze and detect possible input manipulation attacks weaknesses. This kind of tools aim at providing an higher level interface for specifying desired properties and automatically checking for their fulfillment, rather than the activity of manually inspecting the code looking for patterns, which can easily miss important issues.

Part of the problem in computer security, and in specific in the input validation area, is that some sort of particular problems are not known from the developers or they are not taken in account due to distractions. The work of El Khoury et al. (29) aims to better organize the knowledge about security property and the correspondent implementation of security patterns by using ontologies and semantic queries. The purpose of this and other similar works is to provide a more efficient and easy to use method of accessing information about validation methodologies in order to remove bugs introduced due to ignorance or inaccurate knowledge of a particular security problem.

- **Business Rules** management is traditionally related to the system used to store and handle data. All these set of policies can be implemented at database level, via use of **Triggers** (30). DBMSs provide support to write procedural actions and the definition of their triggering condition, in such a way to allow the system to be reactive and capable of
handling or preventing certain actions to occur. Triggers are supported by all commercial and non-commercial systems (Oracle, SQL Server, MySql etc.). Additional tools for a high-level handling of business rules, like Oracle Business Rules(31), are often provided as well as packages or service integrated in enterprise platforms, for example in the case of J2EE and .NET platforms there are respectively JBoss Enterprise BRMS (32) and Business Rule Framework (33). These tools can be compared with triggers in terms of their functionality, but their logic is written at the application code level instead that directly in the database. They offer better support and integration with the development environment to the programmers, who are more likely to be already involved in the development of the enterprise application for which they are writing business rules, and so can get higher benefits from this deep integration.

6.1.2 Ethos Approach

Input is the largest part of the attack surface of a system. For input validation, it is very difficult to allow a certain degree of expressiveness and at the same time ensure security properties. Sometimes it is even difficult having a clear understanding of what kind of restrictions and verifications processes have to be applied to certain fields or input streams in order for them to be vetted. Due to the nature of the problem, it is very difficult and most likely impossible to develop a specific tool or protection that addresses all possible problems. Even if such a tool existed, it would not be possible to cover, and be prepared to handle, all the possible (countless) input variations. It is not possible to create this kind of universal protection system. What is instead possible, is try to put in a foundation for a solid, and possibly fairly general, mechanism
to handle input validation. Rather than Input Integrity (which is carefully handled by default from the system cryptography) and Business Rules, the focus in Ethos is on the Validation. Two principal abstractions in this process are Ethos Markup (EM) and System Typing.

EM is a markup type intended to be used in Ethos applications. Differently from other markup languages, EM is not text based, and is instead type based. Aside with Ethos Style (ES), which is out of scope for the purpose of input validation, it defines how object should be displayed in Ethos application and gives information to the system about what they are supposed to be. The second aspect is actually the interesting one, because it allows a mediation between input data and the underlying system. Describing the possible sources of interaction in an application using EM elements, gives the opportunity of implementing directly in the system the handling of specific problems. The code written in the system is first, much more tested than a possible generic application code and second deeply integrated with every other facilities provided by the OS. For example in the case of file involvement as a container for elements or in the elements definition, permissions associated with the element are automatically managed and delimited by the file through the usual authorization system, which will be explained in details in 6.3

This structure provides the basis for the possibility of input which is in some form automatically constrained or at least subject to a verification. The entity of this verification process is embedded in the underlying system, relieving the programmer from the burden of actually implementing these kind of mechanisms and the possibility of coding them incorrectly. Assume a simple example in which in a given user interface of an Ethos application an hypothetical
\textit{Bounded Integer} field is used for storing a quantity of an ordered item. System will provide facilities to easily specify the bounding range and it will automatically validate the field content accordingly to the policy associated with the markup element.

Ethos is inherently typed (will be explained in detail in section 6.6.2), by means of this there is a strong wiring between what the system is given in input and types. This allow in addition to what described in previous paragraph to perform type checking at every level of the OS, constantly being able to verify the correctness of operations on data, at least from syntactical point of view.

Obviously both markup and types have reasonable limitations, analyzing syntactical structure of an operation or an object doesn’t ensure that its content is semantically meaningful and as long as policies can be attached to markup this doesn’t guarantee that every possible “dangerous case” will be covered by those. Furthermore being able to provide a sufficient rich set of markup elements to give a fairly expressive semantic and at the same time tightening the attack surface is not a trivial task to accomplish.

Together with this reasonable limitations, Ethos approach is however capable of providing mechanisms which give programmers the availability of a well tested and deeply integrated mediation system, which is easy to use and compatible by design with the underlying infrastructure. This removes or greatly reduces the programming effort required to set up a proper and reasonable security checks on input. It is a great achievement considering that, in the major part of the cases, bugs comes from laziness and not sufficient care in implementing proper
checks or from poor knowledge of particular security problems deriving from an inadequate handling of some input data.

Plus in comparison to previous presented approaches the basic security level offered by the above described Ethos mechanisms, produces a certain degree of enforcement: even if is still possible to step out the intended path and explicitly force work-arounds in the provided abstractions, this is less like to happen than in other cases. The reasons are very simple: in this approach, the default way of implementing things is the “secure” way. This means that it is studied and designed to be easy to use and straightforward in its application. Contrast this to the usually situation, a developer who wants to secure a component must rely on external libraries or implementation their own and add significant complexity to the program. In Ethos, writing applications in an insecure way and explicitly forcing a bypass of certain facilities is much more complex and requires much more effort, than using what is provided by the system, this has a huge impact on developers, who are often tempted to write less secure code to save time and effort.

In addition with this major features there is a minor benefit provided by the adoption of GO as APL. GO language doesn’t accept arguments passed to main function. This force the usage of syscalls for providing external arguments to the program, which are then mediated by the OS, since system calls are involved in the process.

6.2 **Authentication**

As stated in 5.2, Authentication is one fundamental piece of the whole security chain. Every security layer relies on authentication in order to correctly perform its tasks. Two main
categories of authentication can be performed: the one that involves users and one that involves operations. The first is the classical authentication that logically connects a piece of software to a person or entity. The second one, in the end, is again a way to check that an operation is performed on behalf of a well identified subject, but represents a more fine granularity of the aforementioned process. Clearly it is not always possible to authenticate every single operation, this is done only when it’s critical according to the relevance of the operation performed. For example ensuring that the data passed as parameter have not been tampered. This can be done by using signatures and cryptographic hashes. This is an important activity, which is necessary in order for the following operations on those data to be meaningful, and for the programmers to be able to rely on information which they are sure that are not be altered by an external, possibly malicious, entity.

The second important perspective about authentication is the one regarding localization: people need to be identified locally on their machine, but also remotely on servers or other machines. These two kinds of authentication have to comply to a variety of different standards and problems. Moreover due to the physical impossibility of application, some techniques can’t be used in both cases.

The key of an effective authentication is to find an optimal ratio between the fraud rate and the insult rate. These two measures represent respectively when a user who shouldn’t be successfully authenticated is able to log in and the opposite. The magnitude of both rates, which are in some way complementary, is related to the level of security ensured by the applied mechanisms, which is often proportional to the level of complexity (exposed to the user) in
using them correctly. Is almost always possible to strengthen a mechanism, but the price is usually a loss of performance and ease of use. It then becomes critical to carefully craft the mechanism according to the specific service, in order to meet security requirements and allow the user to be in the condition of correctly using the system.

The importance of having a strong and effective authentication system is well explained by the concept of *Repudiation*. A user can repudiate an action every time can’t be proven that he committed that action. The term proven, refers to the minimum legal requirements that give a commonly accepted degree of confidence that an action has actually been performed by a subject. Authentication systems that are broken obviously doesn’t meet such requirements, therefore a failure in the authentication is not limited to the technological impact, but can have serious repercussions in different domains like legal, social, etc.

### 6.2.1 Community Approach

Also Authentication is very broad area, in the following paragraphs will be presented a series of examples of improvements made by the community on current systems in order to provide more secure infrastructures and effective authentication.

- Communication over the network is always been traditionally unencrypted. With the evolution of the networks in the Internet era, a lot of situations requires an higher level of protection and so plain text communication is not anymore acceptable. In order to cope with these kind of problems, many *protocols* have been developed to provide secure infrastructures for network communication.
The principal protocol over the Internet is HTTP. In order to adapt it to security requirements for operation such as e-banking, sensitive data submission, etc. it was necessary to add a security layer: Transport Secure Layer (TLS) formerly known as Secure Socket Layer (SSL). The resulting composition, HTTPS is then used to secure communication over the Internet. Even though is a de facto standard, due to problems related to TLS, it has been proven to be attackable (34). Drawbacks include the lack of mandatory mutual authentication which leave space for man-in-the-middle attacks (35), and the fact that resulting security level depends highly on the correctness of the browser implementation and from the set of cryptographic algorithms supported by the web server.

- Another important function in the field of authentication is played by remote shells. Having the possibility of securely authenticate on and manage a remote system, has always been a real priority. Even in this case the evolution of software used by the community has severely improved the level of security, former remote terminal systems like Telnet, used to send passwords over the network in plain text.

Modern systems like Secure Shell (SSH) use encryption to protect data transmission, this protocol has become so popular that lots of other services have been implemented “over SSH” in order to be secured, examples are SCP and SFTP, but in general everything can be encapsulated via tunneling. Unfortunately some aspects still lack of the necessary protection, for example password authentication in SSH is still allowed, and even when using cryptography, keys can be held in plain text in a simple folder without any type of guarding.
Networked services and programs always rely on frameworks in order to secure their communications and data exchanges. Such frameworks like Generic Security Services Application Program Interface (GSSAPI) (36) are used together with Pluggable Authentication Modules (PAM) and setuid in order to manage authentication. This scheme is very complex and contains a lot of security issues.

PAM code runs in application space, a failure in the module can seriously create security issues like leaking user sensitive information or even bypassing the whole authentication. setuid mechanism has been very risky in term of security since it allows the same user to posses multiple credentials at the same time. Speaking of GSSAPI the most famous implementation is Kerberos. Major limitations of Kerberos are the requirements of a central authentication server (which also implies that if the server is compromised the attacker can impersonate whoever he wants) and the fact that the administration protocol is not standardized and differs among implementations, leaving space to misunderstanding and errors.

Password file in traditional OSs it has evolved from being stored in plain text to a much more secure version that make use of Hash and Salt to protect sensitive information. For example the etc/shadow file on Unix system stores hashes of user passwords. Every time a password is inserted the hash is computed and compared with the one on the file. Using a strong one way hash function guarantees that disclosing the hash value won’t compromise the system, because is extremely hard to obtain the former password from the hash value, due to cryptographic properties of the used function. In addition to that,
salt (random bits used in the hashing and stored along with the hashed password) is used in order to make more difficult to build a reverse dictionary of encrypted password and makes impossible to tell whether or not the same user is using the same password on different systems.

- Among of physical devices, tokens have become really common these days, especially with the growth in popularity of e-banking services. These little devices (built using anti-tamper methodologies) are used as a support to strengthen traditional password authentication. Usually they provide in an algorithmic way a one-time temporary number that is used in combination with a user ID and a PIN. This number is checked by the server using the same algorithm in order to verify if is valid or not in that frame of time. The drawback is that if the algorithm is identified and there are not further protections, like for example the use of cryptographic keys in the generation process, an attacker could himself generates valid one-time numbers.

- In order to securely manage several passwords in a secure way many solutions are available integrated in the systems and separately on the market. One of the most famous is KeyPass (37). This portable program provide extensive of functionality related to passwords like secure memorization through master password and key encryption, arbitrary length and char set password generation, auto-typing, importing from browsers passwords, etc. Using this kind of software helps to follow best practices like using different and complex passwords on different services, avoid writing down passwords in order to remember them,
avoid forgetting the password copied in system clipboard on insertion (the program auto-
clean the clipboard after 10 seconds when a password is copied).

6.2.2 Ethos Approach

Ethos primary focus is providing strong and secure authentication infrastructures, while
driving down complexity (38). The task of making authentication the more simple and straight-
forward as possible has a really precise goal: simplifying (so reducing errors caused by high
complexity) the infrastructure in order to make it easier to be correctly used by programmers
and to be configured by system administrators. Requirements for achieving such an ambitious
goal can be summarized in the following list:

**Strong Authentication** one of the key factors in the design, is adopting techniques which
have been proven to be sound, hard to break, and at the same time adequately flexible
to handle both remote and local authentication.

**Effectiveness in user-process association** system has to effectively associate the user with
a process. Therefore authentication has to be able to avoid attacker intercession during
this operation and privilege escalation. The code aimed at authenticating the user has to
be clearly separated from the application code.

**Support Internet-scaling** the adopted mechanism has to be sufficiently flexible to scale even
over large context as the Internet, while maintaining efficiency and security at a reasonable
level.
Confidentiality usually authentication techniques deal with secrets: passwords, algorithms, keys etc. The system has to carefully manage these secrets during the process, in such a way that they are not disclosed to third party or attackers.

Compulsory if the authentication mechanism can’t be bypassed because is in some way “imposed” by the system, several flaws resulting from missing authentication, can be avoided.

Ease of programming a strong simplification in the authentication mechanism provides abstractions which are much easier to use and helps programmers to avoid the introduction of inadvertent vulnerabilities.

Lightweight of configuration administrators usually have lots of responsibility, sometimes systems are so complex that small institutions can’t afford a proper verification process and so tend to outsource the service to larger organizations. By providing a lightweight configuration infrastructure, administrators can effectively handle themselves these kind of operations.

Starting form the above requirements the design of Ethos authentication has been crafted in order to provide better security and lower complexity. Ethos supports distributed authentication through a Public Key Infrastructure (PKI), the system administrator is able to select himself which keys are used as roots of trust. Every computer is identified by an Host Key (a couple of private and public key) which is associated to the physical machine, in addition to each user on the system is provided a personal User Key. Using keys is possible to securely identifies users over the network, moreover Ethos deals with anonymous authentication by using the host key. In this way is always possible to authenticate a subject. Authentication and keys
are managed at system level, keys are never provided to the applications, all related actions take place in the system. The key store is protected by a Trusted Platform Module (TPM) and mitigate cold boot attacks by erasing from RAM keys which are not needed.

The authentication mechanism in Ethos is privilege escalation free because the process never changes its ownership (so it’s not possible to perform privilege escalation exploiting setuid features (39)). This task is accomplished using virtual processes in the way described in 2.1.2. Network authentication guarantees that authenticated access is automatically performed by simply receiving the incoming connection, and confidentiality is preserved since the distributor can’t modify the descriptor content. In addition to the mutual authentication, all network connections are encrypted so even if the content is in some way wiretapped by an attacker, is not possible for him to modify it or read without being noticed.

User authentication on the system, in its concrete version where the user is physically able to type into the machine, is handled through passwords. Usual facilities such as password file encryption and use of hash and salt are adopted to adequately protect such information. In addition authentication it is implemented using the same virtual process scheme. The user’s virtual process which has received the terminal descriptor (and has no need to change ownership and so prevent privilege escalation) is in charge to call the system function authenticate, which will verify user inserted credentials, without ever exposing them outside. This process implements a trusted path with the operating system, and provides visual confirmation that the user is actually typing his password into the system and not into some other process, preventing spoofing.
Summarizing Ethos offers an authentication service highly integrated with the system and decoupled from the application code, easier to use and able to relieve programmers from the burden of writing secure authentication, while enforcing its utilization. The compulsory aspect of this infrastructure provide many benefits, because these set of protections cannot be bypassed by the programmer:

- Process can’t change owners
- Applications don’t have access to secrets
- Network connections are naturally authenticated and encrypted
- Authentication relies on solid standards and techniques defined at system level
- Unauthorized communications can’t happen (will be explained in detail in the Authorization section)

6.3 Authorization

The biggest challenge in Authorization is building a system which is at the same time **Strong** and **Flexible**. It has to be strong enough to enforce required protections and to be resilient to malicious attempts to bypass it. It also has to be sufficiently flexible in order to accommodate possible necessity of complex policies or implementation of high level abstractions. Usually the increasing in flexibility causes a consequent increasing in the complexity of management and design of the authorization policies. Other than a failure in the ACS the most dangerous factor of vulnerability is a flawed design in the authorization policies, that can be caused by accidental
mistakes, but more likely is driven by the incapacity of fully understanding the ACS and the implications among rules.

Another important issue is scalability: an ACS can be very good in a limited environment, because allows to carefully design and manage permissions, this could be the case of a small company with less than 50 employees. In order to be an effective system it has to be capable of scaling in larger contexts such as a big company, a simple example can better clarify this point. Considering the simple case of using an of an Access Control Matrix, that takes in account (user, program) the total number of entries will be the employee number times the number of applications; in a case with 25000 employees and 100 apps is 2,500,000. Such a large number of entries to manage becomes extremely complex to handle and the filling procedure is highly error prone. Together with the scaling to a larger environment, it becomes sensitive also the performance aspect, the ACS needs to provide reasonable overhead to the normal unfiltered access in order to not bottleneck the whole OS. A desirable property when talking about ACS is being able to perform some sort of automatic verification in order to test that the system is sound and some permission leakage can’t happen. An important challenge in this sense is the so called Safety Problem (40). “Given a specific protection system and a specific right r , a subject s and object o such that initially r ∉ M[s, o], can it ever be that after a sequence of commands the right r is leaked, that is r ∈ M[s, o]?” Being able to answer the previous question in an automatic way, gives the opportunity to verify that policies on the ACS have been designed according to a list of mandatory requirements like: this subject should never have this kind of permission on this object etc. Due to previous considerations having an implementation of an
ACS which is decidable can severely decrease flaws related to accidental mistakes and design complexity for large environment.

Like anticipated in the previous chapter in 5.3, implementing access control at system level is preferable and more reliable than doing it at higher level. Here will be discussed several reasons to support this choice.

**Extensive verification** since on authorization code will rely several applications it will be much better tested than some custom code written for one or a few programs.

**Definitional** since system code evolves separately from applications interfaces must be defined in a rigorous way and gives a certain stability over time.

**Better abstraction** again due to the generality of the applications that will run on the system, defined protection mechanisms have to be suitable to interact with a broad range programs and so their abstractions have to be more carefully designed than in a specific ad-hoc case.

**Unification** A single mechanism is used to protect different applications, so there is more compatibility and developers can be familiar with the protection system without going through the burden of learning several specific systems.

**Failure Isolation** a failure in the application can’t compromise the authorization system since the code doesn’t not reside in the application itself, otherwise compromising the application could result in bypassing the authorization system.

**Avoids Compositional Problem** given two applications with two secure protection mechanisms, their composition is not secure by default. If the protection mechanism is unique
and has only one implementation, if it is assessed to be sufficiently secure, composition problems do not arise.

**Analyzable protection** if the protection is implemented using a general purpose programming language it will be intrinsically undecidable. Using a properly designed language gives the chance to obtain results in term of decidability.

**TCB shrinking** if the protection is distributed among applications, all the applications code for this purpose should be reasonable accounted in the TCB and this is going to be much more than a single “system implementation”.

**Simplified applications** developers are relieved from the burden of actually implementing themselves authorization code.

**Safer applications** if they don’t contain authorization code, untrusted applications can be run granting them a tighter set of privileges.

Desirable properties in a good ACS are *expressiveness, robustness* and *analyzability*. Expressiveness means being able to effectively express required protections in a different variety of situations. Robustness refers to the fact that has to be tolerant to system changing while still providing the same core properties. Finally Analyzability deals with the fact that has to be sufficiently easy to understand in order to be analyzed and verified.

### 6.3.1 Community Approach

- **Application Level** ACS, like explained before, are not the best choice, but the community offers also this kind of abstractions in modern APL. One example is the *Sys-
\textit{System.Security.AccessControl} namespace in the .NET framework (41). Here are provided at a high level, a set of suitable building blocks to implement several types of well known or even customized ACS. Disadvantages of doing so have been heavily covered in the previous introduction. The advantages are high customizability and broad support for features. (crypto-base access, auditing etc.).

- In the case of \textbf{Unix-style} systems (including Linux) the standard ACS implementation provides a DAC based on some simple privileges like \textit{read}, \textit{write} and \textit{execute}. Basically it relies on a “per column” storing of the Access Matrix (assuming that we have Users on rows and objects on columns). The structure used for storing permissions in this way is called ACL. Along with the r,w,e information there are other additional privileges that can be granted, like for example the possibility of using \texttt{setuid}. Privileges information are stored for the \textit{Owner}, for the \textit{Group} and for \textit{All} the others. Owner and Group names are stored together with the information about permissions.

Another possibility derived from the Access Matrix is storing permissions by rows. In this way each subject will have associated a list of so called \textit{capabilities}. These capabilities can be arbitrarily precise, depending on the level of granularity required from the activity of specifying some sort of constraints on the system. This and the previous implementations has both advantages and disadvantages on various perspective, but a detailed analysis of these aspect is out of the scope of this thesis, the important point is the overall robustness and expressiveness provided by this family of ACS implementation, which is in the end quite basic.
The most important exception in the aforementioned mechanisms is the introduction of the concept of superuser. A superuser is special user which has been provided more privileges: ordinary and system ones. So this root user can basically perform every action on the system. This figure was introduced for purposes such as implementing system functionalities outside of the kernel, or administrating the system. Due to its privileges, gaining the superuser status is highly desirable for attackers, too often this task is easily achievable, and the access to that status is not properly protected and controlled.

These mechanisms are a typical examples of DAC model, it is highly centered on the user and it’s not sufficiently expressive to implement more sophisticated types of protections. Therefore using this mechanism on large systems, in which decisions has to be centrally taken by an administrator, can become really problematic and almost infeasible. DAC can be considered as a very basic form of access control.

• Aside from the actual implementation the Windows NT standard ACS is similarly structured as the one described before, regulating assignment of permissions (r,w,e), and the way in which such permissions are managed (ownership, granting etc.).

• In the direction of hardening and improving security in ACS, many steps have been made by the community. One very famous, and often cited in other categories, project is SELinux. In SELinux are provided 3 different ACS (42) a modern TE, RBAC and Multi Level Security. Basically to processes and resources are assigned one or more suitable types, and on the basis of these access policies are specified, a further level of restriction is achieved using roles. By default all access are denied, and policies specification is done
through centralized configuration files using an high level pseudo-language. This helps in term of understandability by a human reader decreasing the number of accidental mistakes.

The structure of SELinux access control is designed to provide a flexible, fine-grained ACS, able to mandatory enforce access policies of resources. Everything is driven by the principle of least privilege, providing the necessary freedom to accurately tighten policies in order to avoid privilege escalations. The result heavily addresses the limitation of standard Linux distributions which relies only on two privilege levels: user and root.

- An important contribution in providing a lot more flexibility and options in choosing among ACS (43) has been given by Trusting Computing BaseLinux Security Modules (LSM) (44). LSM is a general purpose access control framework for the Linux kernel that allows various access control modules to be implemented as loadable kernel modules. Some existing access control implementations have been ported to this framework:
  - POSIX.1e capabilities
  - SELinux
  - Domain and Type Enforcement
  - SubDomain (least-privilege confinement)
  - OpenWall

A wide variety of implementations and models is offered from the community to provide effective ACS to system administrators and users. Every system has is pros and cons which
have to be carefully weighted in relation to the specific situation. The purpose of the Ethos approach, which will be discussed in the next paragraph, is trying to provide an ACS which is sufficiently adaptable and versatile to be effective in a fair number of environment and situations while keeping bounded the complexity of the system itself.

6.3.2 Ethos Approach

Ethos authorization system has been designed to supply adequate facilities under very different situations in term of requirements. Policies are specified on the basis of the pair (object label, subject label). This pair can correspond to different things according to the chosen labels, one of the simplest and most similar to previous discussed ACS, is the case in which an executable and a user are represented by the respective object and subject labels. Several other elements can be tagged with object labels like for example directories, files, communication streams etc. This directly reflects on the fact that, the number of properties that can be enforced increases, the number of objects on which the OS can enforce policies increases and there is a reduction in the number of types required to express properties and so a consequent simplification.

The whole authorization system relies on LEAP (45) in order to express authorization policies. As anticipated earlier, the LEAP language is very versatile. Building blocks of LEAP are object and subject labels. On the object side is accounted every possible element which can be labeled. In the policy specification objects can plays two roles. One is the target of the rule which is the class of objects which are affected in the rule application (like all the financial logs) and the other is the receiver which is the class of object to which the rule applies (like for
example all the spreadsheet application to manipulate financial logs). Regarding the subject part, LEAP deals with groups of users. This gives enormous flexibility, other than a generic group case, if needed group can be reduced to single user or be associated to a single object.

The mechanics of the language are mostly based on groups, there are three principal building blocks:

**Group Set Template** The rules for creating and maintaining the group.

**Group Set** collection of related groups, governed by its group set template

**Group** collection of users

So in LEAP other than the rule definition section, a whole subset of the infrastructure is dedicated to define how authorization elements are supposed to behave and evolve into the system. The most interesting element is the first: Group Set Template, here are defined serveral information:

1. It contains the listing of groups and means for determining group membership (via the group tag map).

2. It contains group relabel rules (distinct from object relabel rules) which regulate group membership.

3. It describes how users are added to the group set (via new user rule and group initialization rule)

The group set template can be seen as the abstract specification of a type which is in the end instantiated in a concrete Group Set which follows the rules defined in its template.
Once the architectural group specification is completed there are many ways of specifying the actual policies, the mechanism like said before relies on targets, receivers and groups. Usually a rule is specified in the form

\[
\text{Permission}(\text{Target}) = (\text{receiver}, \text{group})
\]

meaning that a particular permission on object of class target is given to object of class receiver which belongs to a particular group. Classical unary permission are Read, Write, Execute and Create. LEAP also provide 2 binary permissions, mayFlow and Relabel. Relabel gives the permission to change an object’s label while mayFlow gives permission to write objects with tag \(T_2\) after having read an object with tag \(T_1\). The mayFlow is a necessary but not sufficient condition: read permission was previously necessary to allow the read operation on the first label; write permission is necessary at the time of the mayFlow.

The framework described above allows to address a variety of problematics providing support to:

- group creation and management
- access rights confidentiality
- information flow integrity
- approvability and separation of duties
- fine grained permissions
- DAC and MAC integration
With LEAP is possible to implement many standard models such as DAC, MAC or RBAC. For example a set of fundamental rules which are going to deal with the organizational managed data (independent from the user identity) can be defined using a MAC, while the unmanaged data which are user specific with a DAC; this according to the rule that the MAC is priority and defines what is let over to the discretionary system.

Another fundamental aspect of LEAP are its decidability features. Like showed in (46) the implementation of a discretionary model using LEAP has decidable safety property in polynomial time (47). Being able to “interrogate the system” gives the possibility of mathematically verifying a set of requirements, detecting flaws in the policies configuration before the system actually goes live.

So far it has been showed that Ethos authorization system has been designed in such a way to provide good security properties. It is expressive enough to allow the implementation of standard authorization models and handling additional features which are usually omitted by other systems (separation of duty, information flow etc.) It’s robust because it doesn’t rely on sophisticated system concepts other than user and resource, and this gives high degree of freedom to extend or tighten these categories while maintaining authorization restrictions. It’s analyzable because policies are expressed in a high-level language, specifically built for this purpose, so it provides high scalability and, rather than a low-level syntax like a permission matrix, is more clear and understandable by a human reader. Moreover it’s analyzability is improved by its decidability properties.
6.4 Improper Isolation

Isolation mechanisms are a careful blending of privileges management, resources management, and monitoring. The real challenge is trying to provide suitable isolation while allowing the application to properly execute its tasks. Often, one of the main obstacles for the final user, is the difficulty of specifying a configuration in such a way that applications are still able to run, but are also correctly isolated.

6.4.1 Community Approach

- An early work in terms of isolation, but in general in providing a system which is more secure independently from the security additions later installed, is Compartmented Mode Workstation (48). This system included several security features such as MAC, root split up, two man rule, system administration split up etc. and was rated B1 (including features from B2 and A) according to the Orange Book assurance levels. Together with rigorous access control based on the concept of compartmenting resources, also at network level the system was only allowed to communicate within a certain list of authorized hosts, that was contained in a database. This system hadn’t much luck in future deployment because of economic reasons.

- On the market are available a lot of commercial software whose claim is to take applications and executing them in a Sandbox. Often this solution provide a separation in term of files visibility, in order to protect leaking of personal information. Some example of these are Sandboxie (49), GeSWall (50), Bufferzone (51). Limitations of these solutions are often related to their effectiveness, due to the fact that the integration of the sandbox
is built on top of the system and so has to deal with mechanisms which weren’t initially
designed to support these modes of execution. In addition a big problem is represented
by compatibility with other software running on the system, often the majority of the
issues arise from antivirus and firewalls.

Several antivirus like Kaspersky, Avast, Comodo etc . . . , are now providing “sandboxed”
execution mode as an additional feature in order to test or constrain some programs
installed on the system. This is quite similar to the concept exposed in the previous
paragraph.

Even if the way of implementing the sandbox by the above described software could be
different (most of them are proprietary, details are not disclosed), the common idea is to
provide the ability to ultimately isolate an application, building these mechanisms over
the existing system.

- Regarding Linux environment, Sandboxing has been sometimes implemented using oper-
ations like seccomp or chroot. These two techniques have both limitations, for example
in the chroot case: if a program has sufficient privileges can perform a second chroot and
breaking the “jail” condition in which was put.

Modifications introduced by SELinux to the kernel allow more effective implementations
of the sandbox. sandbox (52) provides isolation by dynamically generating SELinux
policies which are automatically loaded and applied. It exploits MAC isolation policies
to actively enhance sandboxing. Basically the operations performed are: creating a tem-
porary sandbox directory, copying the files, setting up the labeling, execute applications and clean up on exit.

Sandbox is a good tool with respect to the simple use of seccomp or chroot, but it requires a specific set of services provided by SELinux, which are not available on all standard Linux distributions.

- **VMs** were first implementation by IBM is about 40 years ago, but was abandoned around the ’90. VMWare in 1999 managed to turn virtualization into a technology actually usable on the PC. Since that time, improvement has been enormous, this field has gained more and more relevance. Indeed, processor manufacturers started introducing dedicated set of instructions to guarantee better support for virtualization. Progress in this area is the reason why virtualization is widely used to perform isolation.

Due to its characteristics of completely separating the virtualized system from the host OS (of which the former is not even aware) and other possible virtualized installations, this technique is often used to provide a dedicated environment when there aren’t enough physical resources to perform separated installations. Common case is using VMs in hosting applications, where several web servers are run separate VMs on the same host.

The drawbacks are that even though VMs have become more and more efficient, they still require a significant amount of resources (compared to a simple application running alone) which can be too expensive for the purpose of just isolating one or several applications. Plus, the virtualized system running the application is isolated from the outside environment, but the application is not isolated from the virtualized system on which
is running. Therefore if the application is attacked the system could be corrupted and arbitrary altered to harm other applications.

Another example of isolation by virtualization provided at application level rather that system one, are Java Applets which are self-contained programs running in a virtual machine (Java Virtual Machines (JVM)) which performs the isolation.

- sometimes isolation is achieved using Encryption services. Encrypting data used by the application or exchanged with other trusted processes or hosts, prevents malicious programs from wiretapping information. One simple level of encryption can be achieved working on a ciphered partition or virtual disk. This kind of service is provided for example by TrueCrypt a powerful tool that allows to encrypt whole partitions or alternatively creating ciphered volumes residing on a single file. Even in this case encryption comes always as an additional feature and the programmer has to explicitly take care of it in the implementation of the application.

- in the case of Web Browsers isolation has become a particular relevant topic. The majority of the time spent by an average user on a PC is surfing the web, it has strongly increased over the years (53). Web browsers have become more and more rich in features and in capabilities, adding a tremendous amount of complexity to their design and implementation. Nowadays browser implementations count millions of lines of code (54). This together with the fact that is the software most used by users and it’s deeply “connected” to the Internet, made browsers the preferred medium for cyber attacks. Large amount of
code and complexity: therefore high bugs probability, and consequently more chances of finding security holes to exploit.

For the above described reasons isolation is a very important issue in web browsers, more than in several other softwares. For example in the case of Google Chrome, every tab executes as a single process which, is deprived of several privileges such as writing files or reading from sensitive areas etc. Every process is then terminated when the tab is closed. Similar solution have been adopted also for other important browsers like Internet Explorer (Protected Mode), Firefox, Opera, Safari.

Sandboxing sensitive components as pages, scripts or plugins prevents malicious code to easily take over the process and harm the user’s computer through the browser, the increasing importance of this topic is confirmed by the inclusion of the sandbox attribute for the iframe tag in the HTML5 standard.

6.4.2 Ethos Approach

Ethos operating system deals with Isolation on several aspects. At process level, like all other operating systems it performs isolation through virtual memory management, so that address spaces between processes are not shared. In addition to that, programs are not allowed to see the process table, in this way information regarding resource consumption, timing, ownership of processes etc. are protected. Visibility is restricted among related processes: parent processes has access to information of their child processes.
An important piece of a correct isolation includes carefully managing sensitive information. Generally sensitive information like network keys are managed by a so called “Key Ring” this facility basically stores on disk and encrypted version of the information which can be extracted for being used only by providing a valid password, usually associated to the user. This mechanism has two issues: one is integration and the other one is availability.

The first problem is represented from the fact that usually information stored in the keyring are a subset of all the sensitive information of the system. For example document folder encryption is available on request, but not mandatory enforced. Another famous case is the SSH keys one, by default keys are stored in the form of plain text in a regular folder where everyone can easily access and steal them by simply having access to the file system. In Ethos the concept of keyring is planned to be implemented widely at system level on every sensible data. Sensitive information, like keys are stored and managed by the system, which is the only one interacting with users applications, allowing adequate protection even in case of malicious access.

The second issue of the previously described mechanism is availability. In some cases the physical presence of the user typing the keyring password could be problematic. Increasing the availability of protected information while maintaining same level of security is not an easy task, one of the possible solution proposed is using a TPM to manage keyring protection, in such a way that only the system is able to access the keyring and can do it also without the user typing a password. Additional considerations have been done in the direction of allowing key recovery in case of damage, results are deeply related to trust issues.
Covert channels are one of the most subtle problems to detect and address. They can be found or voluntarily be inserted basically everywhere (55). Their presence can be reduced by carefully design and testing. However, a certain removal would come at a too high cost: implementing code which is in some way completely not related with any form of tracking or pattern is indeed possible, but for example will involve avoiding using caches, Memory Management Unit (MMU), dynamic scheduling techniques etc. This is an unacceptable compromise, that’s why Ethos pursues the final goal of decreasing as much as possible covert channel presence by accurate design of its component and implementation of system libraries.

Isolation of a system from the external environment is an important part of the design. The way in which the OS manages connections with the external world is in the end a map of all possible entry points that are exploitable by an attacker. Therefore, it is very relevant providing a certain degree of separation between what relates to the outside environment and what has to be let inside the system.

In order to secure incoming and outgoing connections, Ethos implements system-wide cryptography. Encryption is broadly applied through all system communications, internal and external ones. Ethos uses NaCl cryptography library (56), a state-of-the-art high-security high-speed software library for managing cryptographic operations such as encryption, hashing, signature etc. The library is specifically designed to be fast and easy to use.

Even if a cryptographic algorithm is sound and it’s not been broken, his design can easily lead to implementation which are vulnerable. This happened for example with Advanced Encryption Standard (AES) (57), where researchers were able to obtain the full key in 65 milliseconds using
cache attacks. One of the main focus of algorithms used in NaCl is that their straightforward implementation should be as strong as possible, this means lowering the additional amount of code and complexity to be added in order to produce a secure implementation, compared to other famous algorithms.

Building blocks of the NaCl library were all designed by the implementer of the library prof. Daniel J. Bernstein. They will be presented in correlation with the equivalent standard commonly adopted by the community to better clarify their role in the library. **Elliptic Curve Diffie-Hellmann (ECDH)** belonging to the family of ECC, built on the particular curve25519 (58) much faster and secure than the standard NIST P-256 curve. This block copes with all the public cryptography services, advantage compared to Rivest, Shamir, Adleman (RSA) is lower size of the key at the same security level. **Salsa20** (59) is a stream cipher intended to manage all operations related to symmetric cryptography, it is consistently faster than AES in its 20 rounds version. **Poly1305** (60) is an high-speed Message Authentication Code (MAC) used to efficiently handle signature and verification operations, its security level is as higher as the underlying encryption algorithm.

Implementation of Isolation is in the end an appropriate blending of privileges and resource management, the facilities described above provide a reasonable amount of isolation regarding system components and in general of the system with respect to outside environment. Additional tools like the one seen in the community approach section can be developed exploiting facilities as Ethos access control which have been detailed described in the Authorization section 6.3.2.
6.5 Initialization

General level of protection against Initialization flaws is the one demanded to APLs and it deals with default values assignment.

6.5.1 Community Approach

- In many modern languages initialization of variables is managed according to types. For example in JAVA, every variable which has not an explicit initialization is set by the JVM to a reasonable default value for its type. In the case of an int would be zero, for a string would be empty string, etc. When it comes to complex objects like lists the null value is assigned. In the case of classes the JVM ensure through Initializers that every initialization code (constructor or direct assignments) is executed before the new object is used. This ensures that variables are initialized before being used, it doesn’t provide guarantees on the meaningfulness of the initialization.

- Another interesting example is the case of Python, which is instead a dynamic typed language. Due to this characteristic type is not known a priori and so the compiler can’t initialize variables to a default value. Is therefore forbidden to declare variable without initializing. There is the possibility to assign a none value, which is a generic default null value, but consequence of the language design and philosophy is to declare a variable when is used and so initialization is always performed. Even in this case there is no guarantee on the fact that the initialization assigns a correct (meaningful) value.

- There exist some tools developed to give an higher level of enforcement, one example is this modular system (61), which via policies specification and a type system is able to
verify if a particular initialization pattern is fulfilled. Unfortunately it requires additional effort to the programmer. Use of such tools seems not to be very diffuse among developers, who just passively use APL facilities.

6.5.2 Ethos Approach

In the Ethos OS the APL is GO. Similarly to JAVA, GO automatically assigns values to uninitialized variables according to their type. GO is a procedural language, but provide necessary facilities to write Object Oriented (OO) code. There is a separation in the code between the data definition and the operation (what is generally called method in a normal class). In the case of complex objects the same rule explained before is applied so that even in that case variables comes initialized to a standard value. As previously described, there is no particular guarantee on the fact that the value is reasonable and meaningful with regard to the program semantic.

6.6 Type Safety

Even for this category usually a great portion of the impact on security is determined by APLs.

6.6.1 Community Approach

In traditional OSs the type management is implemented as and additional abstraction at APL. Almost all modern languages are built around the concept of type. Every language has rules and way to describe types and define how these are going to behave and interact with each other. The implementation of the infrastructures used to manage types are then defined
and provided by the language, so for every different language, types may have different im-
plementation and layering through the system. Little differences in the conventions or in the
intended ways of performing standard actions, are often cause of several bugs when different
languages interact with each other. In some cases even using the same language due to imple-
mentation which are too permissive, unpleasant issues can arise, for example when loading or
saving instances of types (classic objects serialization of JAVA) on disk.

Another example is the case of SELinux types. In this case the approach is more similar
to the Ethos one (which will be described in the following section), in the sense that type
implementation and enforcement is managed directly by the OS, but totally different from the
point of view of the meaning associated to types. In SELinux types are conceptually labels
associated to processes and resources used by the authorization system to enforce security
policies according to a specification contained in configuration files. In Ethos, types are intended
as the straightforward concept traditionally associated with programming languages types.

6.6.2 Ethos Approach

As anticipated in the previous section, the concept of type in Ethos is the same that everyone
expects. The big improvement is that Ethos types are implemented in a language independent
way at system level, moving the burden of all handling and verification at a lower, more trusted,
more tested and inherently more secure level.

ETN is a programming language independent type system, it offers to every supported APL
facilities to interface itself to the OS, like for example encoding and decoding of types, definition
of new types etc. In Ethos, differently than in common cases, there is a clear separation between
definition of data and the code used for their manipulation. Being language independent allows different languages to refer to types in the same way without compatibility problems, reducing probability of flaws in the implementation.

Like in the case of SELinux, Ethos types have an impact on the authorization system. Types together with other information are actively used by the authorization system to enforce security policies and checks. For example all the files contained in a directory have to be of the same type. Automatically, at the creation, proper authorization rules are associated and later enforced according to file type which is always determined. Using the type system, Ethos can actively check this requirement.

Another important aspect is represented by communication. Every kind of communication in Ethos is typed. This happens for the case of network communication (like in send and receive operations), of RPC, in local communication among processes etc. In this way is always possible to verify the formal correctness of a received data type, what is not possible is ensuring that its content is meaningful and appropriate.

In the end, Ethos offers a flexible infrastructure to deal with typing, unifying and simplifying communication between entities and different languages. Implementing all the related mechanism in the OS gives the unique opportunity to exploit type benefits widely in all the activities of the system. In addition, incorporating at kernel level, is a further guarantee in term of quality and care in the implementation and gives the necessary facilities to build automatic verification tools.
6.7 Concurrency

As anticipated in 5.8, Concurrency is a very hard matter to deal with. Interactions between concurrent fragments of code which are running on the system are non-deterministic and heavily related to low-level and architectural factors, which are difficult to keep in mind and to track, especially while the developer is focusing on the problem to solve, rather than how to correctly manage concurrency. Starting from the base point that concurrency is not something that can manage itself alone, intervention is required to bring some form of order and securely deal with possible dangerous flaws that can arise from a concurrent execution model.

In order to give developers the possibility to secure the interactions between concurrent programs, several tools are generally provided at different levels. Among the categories of flaws, Concurrency is for sure one of the most difficult to “make it right” when it comes to secure programs using appropriate tools and abstractions. Therefore even if several facilities are provided the effort required to the developer in order to effectively manage concurrency is higher than usual, and requires a certain amount of experience.

6.7.1 Community Approach

- Commonly the independent units of execution directly involved in Concurrency are processes and threads. From the point of view of security the former are more reliable than the latter because there is a clear address space separation like explained in 5.4, and information must be explicitly shared using IPC mechanisms provided by the system, while with threads almost all the data is shared by default.
Process and threads can be managed by the system kernel in different ways (62). Usually a thread is seen like a lightweight process because it is conceptually a unit of execution, with the variation that carries less overhead, and so results in lower isolation, but higher performance in switching. Many systems implement threads as a different schedulable object than a process, in this case the system is aware that it is dealing with threads. In the Linux case (63) threads are implemented as normal processes which share address space. Regardless of their technical implementation, the important point is the final abstraction provided. Referring to threads there is an automatic sharing of data, so they do not have to explicitly arrange communication like as do processes.

Concurrent processes that want to work on shared data are supposed to arrange this by using system facilities. In the Linux OS this is usually done using primitives such as `shmget` for Shared Memory, `mmap` for Mapped Memory, `pipe` for Pipes and `socket`, `listen`, `accept`, `bind`, `connect` for Socket. Concurrent access to shared data is regulated through locking mechanisms which in the Linux case are `fcntl` for file locking, `mlock` for memory locking and `semget`, `semctl` for System Semaphores (System V semaphores). In the case of threads, memory is already shared so the only necessary action to perform is to properly managing sensitive variables in order to avoid interferences and violations of data integrity. In Linux environment these functionalities are provided by the `pthread` library and usually mechanisms used include `mutex`, `semaphores` and `condition variables`. 
The facilities described so far can be considered system provided. Even if only a subset is directly implemented a system call, their semantic is deeply related to the system and low level. This is a big disadvantage in term of complexity. A correct use of previous infrastructures requires the programmer to deal with parameters of not immediate understanding, highly related to architectural information, like page size, memory management, signals etc. Due to these reasons, a programmer who decides to secure concurrent applications at this level has to be more skilled in order to produce a reasonably secure implementation.

- Mechanisms very similar to the one exposed in the previous point can be found in APL provided with richer and usually more understandable semantics. Modern languages have taken the burden of implementing these higher abstractions on all the systems which support the language. For example the `System.Threading` and `java.lang.Thread` namespaces, respectively in C# and JAVA, provide all sorts of abstractions to properly deal with threads synchronization, and even more complex structures like thread pools and advanced scheduling facilities. Assuming a solid implementation in the language runtime, this solution is far less error prone than the previous one and can give several advantages in term of clean design and usage of appropriate software engineering design patterns.

- Distributed systems have to add a relevant amount of additional rules in order to properly coordinating concurrent transactions operating on shared data (usually contained in a database). The upside is that the programmer has only to deal with the protocol by checking that operations have been confirmed, in the worst case it has to re-send it.
The complexity of validating, ordering and recording transactions is moved to the DBMS. This model differs from the classical concurrent thread model because concurrency is generated by the fact that many users are querying a shared resource, instead of a single program performing multiple operations at a given time. However the two situation can mix together without modifying too much the problem from the perspective of the DBMS, because it is not aware that the application is or not multithreaded, what it sees are just multiple requests to manage. Given a sequence of operations, it is computationally too expensive to determine if the sequence is actually serializable, (equivalent to a sequence of operations that produce a consistent result), or actually leads to a state which is not allowed by integrity constraints of data (30). What is typically done is enforcing a set of restrictions in resource assignment to ensure that the resulting sequence of operations is equivalent to a valid sequence. This is the case of 2-Phase Locking (2PL). 2PL is a concurrency control mechanism that guarantees serializability. It is adopted by all commercial systems, because of is effectiveness together with performance, which are in general better than the one of Timestamps method. Another technique is multiversisoning the shared resources by providing to every transaction access to the appropriate version of the information in order to produce a consistent final state. This solution has the same effectiveness, but requires additional storage resources.

6.7.2 Ethos Approach

The Ethos approach with concurrency is very radical. As has been stated many times during this thesis, one guiding principle is to avoid security compromises. This is the case of Threads,
years of failures give us reason to believe that this model is not actually well suited for achieving
good security standard.

The concurrent model of threads has become along the years more and more powerful, and at
the same time complex with the introduction of advanced hardware and software facilities. This
tremendous amount of complexity make very difficult for the programmer to reason linearly and
produce a correct implementation. This complexity arises from the fact that threads break the
abstraction of the program. Specifically they impact on the program semantic. The programmer
has to move his reasoning scope from a sequential flow of high level instructions belonging to a
single program to a randomized interleaving of low-level machine dependent instructions from
different threads.

In Ethos, threads don’t exist, explicit parallelisms is not supported, but the exploitation of
concurrency is ensured by the asynchronicity of the kernel. Syscalls are by default non-blocking,
so as soon as they are invoked they return control to the program. The concurrent execution of
these, is automatically managed by the system. Syscalls and their generated events give enough
information to the system, to establish whether or not two or more calls are independent, and
can be safely executed “in parallel”, or if additional actions has to be performed. In case of
conflict the system applies holds to calls, and lock to involved resources, in order to ensure that
a consistent result is achieved at the end of the execution.

Even though the system is by default asynchronous, is possible to explicitly block on one
ore more event to occur. This kind of interaction is provided by Event Wait Tree, where leaves
represent events, and the structure and internal nodes store the completion status.
Possible deadlock situations are managed in a reactive way. Detection mechanisms are applied, and in case a situation arise it is taken care of by the system terminating and restarting the specific transaction or system call involved, in such a way that the circular waiting condition is removed and the deadlock can be unlocked.

Summarizing, Ethos offers an approach quite different from the facilities of traditional systems. One can argue that in term of exploited parallelism and performance this is not as effective as the thread model, but the main goal of Ethos is security. As long as the infrastructure is able to ensure reasonable performance, and it is the case, is more important achieving cleaner and simpler abstractions to reason on (and so better resulting security and code quality), than higher performance.

6.8 Memory Safety

Memory safety is one fundamental issue when dealing with unsafe languages such as C/C++. Due to low-level management operations, intended to achieve higher performance, these languages are easily exposed to flaws such as buffer overflows, dangling pointers etc. Big steps ahead over the years have been done in the direction of reducing the occurrence of memory safety issues in programs, and will be discussed in the following sections.

6.8.1 Community Approach

- One of the problems to which developers are more sensitive is for sure buffer overflow.
  
  Several tools and techniques exist in order to try to mitigate this flaw, which has been proven to be one of the favorite medium to break into a system and then perform an attack. Stack Guard (64) and its enhancement Pro Police (65), are extensions for the
GCC compiler. Their function is to generate Canaries and the relative code, in order to detect a buffer overflow attack.

Another technique, which is implemented in all modern operating systems is **Address Space Layout Randomization (ASLR)**. Randomization techniques differ from system to system, but the underlying general concept is the same for every engine.

Although described techniques are really useful to reduce an attacker’s probability of succeeding, they are not sufficiently strong to completely remove the problem (66). These preventions make the attack harder to accomplish because different, less conventional, methods has to be used in order to bypass the protections.

- Memory leaks are a relevant problem especially in the case of long running applications such as servers. Accumulation of leaks can definitely undermine the stability of a software, causing availability problems like memory saturation and necessity of restarting the process periodically. In this direction two important paths have been taken—**analysis tools** and **garbage collected languages**. An example of the former is **Valgrind** (67), an instrumentation framework for building dynamic analysis tools. Using **Memcheck** is possible to detect memory leaks in unsafe languages like C/C++. The latter is the case of many modern languages like JAVA, Python or C# which provides a garbage collector service in their runtime. Clearly it has an additional cost in term of performance, but removes from the developer the burden of actively manage the memory. An intermediate solution is the one adopted by Apple, the case of **Automatic Reference Counting (ARC)** (68) on the Clang compiler front-end, which relies on the Low Level
Virtual Machine (LLVM) back-end to provide assisted memory management. A set of primitives are provided to developers to signal to the run-time system what data has to be referenced from the management system, so specific code is generated and compiled to accommodate these requests. Since this whole mechanism is compiled it set a trade-off between the burden of manually managing memory allocation and the lost of performance of having a garbage collector running.

6.8.2 Ethos Approach

Ethos approach works in two directions. From the point of view of APL, Ethos uses safe languages such as GO. The GO language is garbage collected, then memory leaking problems are avoided. It also doesn’t give support for using explicit pointers (C style pointers) or pointer arithmetic. Pointers are actually supported in the unsafe package of GO, but its installation and supporting is not intended to be provided or encouraged in Ethos.

From the point of view of the system code, dynamic size buffers are intended to be supported. Whenever a buffer overflow of the previously allocated space is detected the system automatically allocates other storage to accommodate the overflowing content of the variable. This mechanism is very useful for cases of accidental overflows due to mistakes or lack of attention by the developers, but has to be carefully manage in such away that the dynamic allocation doesn’t generate availability problems due to memory saturation.

6.9 Arithmetic

One of the most greatest plagues among arithmetic flaws is integer overflow. Several tools like IntScope (69) or RICH (70) work in the direction of analyzing the code in order to detect
overflows. They usually work on a runtime basis, starting from an intermediate code representation such as the one of the compiler or the output of a disassembler, they instrument the code in such a way that unsafe statements are detected. Some tools perform a rewriting in order to handle the problem, like for example accommodating bigger types for overflowing results. Drawback of these approaches is that the detection is not complete since not all paths can be tested exhaustively and some higher level semantics of the language are not recognized at the lower level where the analysis is performed. Another problem is represented by the high number of false positives and the possible code breaking due to automatic rewriting. Some APL implement arbitrary precision arithmetic in order to accommodate growing values, or intermediate solution like in the case of Python in which an overflowing integer is upcasted in a larger type such as a long. These on-demand patching has the advantage on previous techniques that it doesn’t add overhead to the execution.

6.10 Aliases

Symbolic Links are one of the biggest source of confusion, voluntary or involuntary, with which a programmer has to deal. They are often exploited by attackers to show fake views of the system, misleading programmers that think to deal with a certain file and/or service and instead are just going along with the attacker scheme. In traditional systems symbolic links can be disabled on request. This is good, but at the same time gives no guarantee that it will be done and that a possible attacker won’t turn the switch on in order to prepare for the real attack. Ethos approach is very strict on this point, clarity has been privileged over comfort of creating suitable shortcuts and abbreviations.
Another type of alias is the famous double dot notation \(/\). Providing carefully crafted path strings to interpreters is the basis on which relies the \emph{path traversal} attack. Programs that are often victims of these kind of attacks, like for example web servers, incorporate protections to reject a set of “malformed” paths. In this case the word malformed is used to refer to all those paths whose structure resembles the one of a string specifically built to attempt a path traversal attack, according to some heuristic measure. This mitigation is of course partial since can both, reject harmless paths strings and accept malicious paths that have been coded in a way that is not checked by the protection module.

\section*{6.11 Crash Consistency}

In this phase crash consistency mechanisms are not the scope of the project. A simple set of facilities to manage termination, is offered by special virtual processes, called \emph{terminate portal} that can be delegated to terminate and clean up processes, according rules on execution information.

\section*{6.12 Error Handling}

Controlling the correct functioning of a program goes through the management of “unexpected situations”. Unexpected situation are not really not expected events, but more behaviors that differ from the normal intended one. This situations have to be manage in order to recover the program execution or in the worst cases drive the application to a safe, stable termination state. The most common facility to deal with error is offered at APL level and it’s an \emph{Error Handling Mechanism}. Modern languages offers basically two main philosophies: \textbf{error codes} or \textbf{exceptions}. The first one doesn’t really need support from languages since is simply asso-
ating a conventional return value to a function indicating the status of the operation. The second one has to be explicitly supported by the language, all modern languages like JAVA, C#, Python, GO etc. support exceptions. Establishing a priori which one of the two is the best way to handle errors is not an easy task. With the advent of newer paradigms the answer seemed to be clearly exceptions, since they provide a cleaner way of handle errors situations. This is supported also from the trend of all major languages moving from the old way of returning error codes of C/C++ to the new exception mechanism. The real answer is that is not really possible to assert than one is better or worse than the other. This is subject to different factors like the context in which they are used, the way in which are used and the correctness of the design of the mechanism. Many of the drawbacks of error codes that exception supporters often point out are due to poor design, for example lack of a unique agreement on return values and their unclear meaning, and not to the mechanism itself. It turns out that also the exception model has its weaknesses (71). In the end every mechanism can be valid if properly used.

Regarding error handling Ethos refers to its privileged APL, the Go language supports both mechanisms. Like said before error codes don’t need a specific support, but many improvements can be done in order to make these easily usable in the code. Go has chosen has its privileged approach the error codes one. This approach is adopted by all the libraries provided with the language and highly encouraged from the language designers, in the documentation and tutorials, moreover multiple return values in functions have been introduced to make syntactically easier to deal with error codes. Exceptions are supported and the actual name used in GO envi-
The environment is *Panic and Recover*, their use is suggested for rare situations that highly compromise the stability of the program.

### 6.13 Logic and Syntax Error

This area is not currently in the scope of the project, future interest in this direction will be addressed by the introduction of a system scripting language **EL**.
CHAPTER 7

APPLICATION CASE EVALUATION

7.1 Evaluation of Software security

Assessing the security of a program or of a whole system is a tremendously difficult task. Although many approaches have been proposed, none of them seem to be sufficiently effective and accurate to be adopted as standards over the others. In the first analysis, it is difficult to produce some sort of absolute score, instead many metrics are used in an attempt to provide a value included in a bounded range. This limits the ability to perform a meaningful comparison among evaluated software since differences often collapse in the same intervals (e.g. Table III). Another great deficiency is that metrics are not usually comparable because they are based on very different approaches, and so it is difficult to combine contrasting metrics in order to produce a final evaluation.

There is a multitude of metrics (72). They represent different approaches: some are based on scoring of vulnerabilities, some analyze the source code, some are based on system architecture security measurement, some based on the measurement of security in object oriented class diagrams, or based on risk. For each of category it is easy to find faults that greatly decrease the effectiveness of such metrics. For example the one based on consultation of vulnerabilities databases (e.g. CVSS, CVE etc…) do not address possible future flaws, are completely ineffective on software which is not been deployed to the community and so no vulnerabilities
has been reported. Other metrics work at the code level (73), in this case it is really difficult to
find a correlation between some patterns in the code and corresponding vulnerabilities, many
factor can bias the outcome and is not difficult to argue with the assumption behind the scoring
rules because they are highly subjective, and finally rules and patterns that may work well for
some categories of codes may be completely wrong with others. These simple examples give
a rough sense that security metrics today are far from being trusted as oracles. This doesn’t
mean they are useless, but they should be taken more as a “feeling” about the security level of
a software and less as an unquestionable truth.

When analyzing entire systems, it is even more complex to determine how to proceed:
since the code contains the TCB, it has to be verified and assessed in detail. But in order
to understand its resulting security, the line-by-line analysis must be abstracted to enable
comprehension of security infrastructures, services, mechanisms and models. The standard for
analyzing system is the Trusted Computer Evaluation Criteria (74), known as the Orange Book.
Although there is an established and widely adopted standard, it doesn’t mean it is either
perfect or effective: since it is a government standard, higher levels are reserved to systems
that typically deal with government classified data, reducing its effectiveness for evaluating
commercial systems (75).

Limitation in security metric reliability is dictated by the high problem complexity. Several
factors coincide in hardening the generation of an unbiased, effective score. The cause of
metric deficiency are often related to not considering all the factors. For example the human
factor (quality of code highly depend on skills of developers), code density (comparison of
programs in different languages should be carefully approached), \textit{development methodologies} (the rigorousness of the development process, highly impacts on the quality of final product) etc.

7.2 \textbf{Motivation of a Case Study}

We’ll evaluate the taxonomy through a case study for different reasons: some of them are constraints independent from the purpose of the thesis, others are advantages that the use of this methodology can give in term of visualizing the topic discussed in the Analysis Chapter (6). Careful evaluation software is a task that requires substantial financial resources and time. Moreover, as previously shown, current metrics lacks of general applicability and effectiveness. In particular, none of the metrics that relies on public databases could be used, since the Ethos is currently a work-in-progress and is not publicly released yet. The few metrics that work at code level are too subjective and the results not meaningful.

A good compromise in term of significance of the results and practical applicability considering time, money and availability constraints is a case study. Using a case study allows the differences to be compared and highlighted of a typical application in another environment versus one in the Ethos environment. Since the chosen applications use a variety of security sensitive services considerations and results can easily be extended from the specific case to the generally underlying security issue. The case study also allows us to map results and evaluation on reasoning and evidence provided throughout the thesis and the previous analysis.
7.3 Case Study

In this section will be presented a comparative security analysis between two applications: **Postfix** (v 2.8.7) and **Xmsg**. The first one runs on a typical OS environment, while the second runs on top of the Ethos OS.

Postfix mail server was written mainly by the security expert Wietse Zweitze Venema (76), researcher at IBM labs. It is a POSIX compliant application and, in common with Ethos, it is built with security as first concern. Xmsg is an essential mail server written by W. Michael Petullo (77), a Ph.D student working in the Ethos project as kernel developer, further details can be found in (78) currently in submission.
7.3.1 Analysis

In Table XIII and Table XIV summarize the results of the comparative analysis between Postfix and Xmsg. In order to read the results with a clear perspective on the problem, the reader should pay particular attention to the following two statements.

The estimates made during the comparison are very conservative for the Postfix side, in order to provide a meaningful information according to the complexity of the software. This policy biases the comparison favorably towards the Postfix side.

The purpose of this analysis is not proving that Xmsg is a more secure application than Postfix in absolute terms, or that Postfix is a flawed software from the security point of view. This analysis aims at estimating the effort necessary to obtain a reasonable secure application in the case of Postfix (regular OS environment) or in the case of Xmsg (Ethos ecosystem).

Some information that will help to better interpret the comparison results:

- **Total Postfix source code**: 98,467 LoC
- **Total Xmsg source code**: <1K LoC
- **Total NaCl code base**: 51,733 LoC
- **Total OpenSSL code base**: 252,000 LoC
- **Total Cyrus SASL code base**: 44,343 LoC

The analysis relies on all the above and related operating systems to be verified and trusted. We therefore only consider the evaluation of Postfix and Xmsg.
## TABLE XIII

### COMPARISON SUMMARY: RISK, RELATED VECTOR AND DAMAGES

<table>
<thead>
<tr>
<th>Risk</th>
<th>Vector</th>
<th>Affected Areas</th>
<th>Monetary Damage</th>
</tr>
</thead>
</table>
| Delivering a message to a different recipient | Altering the process delegated to fetch data from output spoolers and deliver messages in users’ input spoolers. | **Taxonomy:** Authorization.mediationFailure  
**Top 25:**  
6. Missing Authorization  
15. Incorrect Authorization | $ - $$ |
| Redirecting messages to other users | Altering the process delegated to receive data from users’ mail clients and deliver them in the users’ input spoolers. | **Taxonomy:** Authentication. (provenance | failure | weak)  
**Top 25:**  
5. Missing Authentication for Critical Function | $ |
| Receiving mails from potentially dangerous senders | Exploiting flaws in a broken, weak, or misconfigured authentication system | **Taxonomy:** Improper isolation. (missing | broken) cryptography  
**Top 25:**  
8. Missing Encryption of Sensitive Data  
19. Use of a Broken or Risky Cryptographic Algorithm | $$-$$$$ |
| Exposing or altering the content of message | Exploiting flaws in a broken, weak, or misconfigured encryption system | **Taxonomy:** Memory Safety.  
**Top 25:**  
3. Buffer Overflow  
11. Execution with Unnecessary Privileges | $$-$$$$ |
| Executing malicious program | Exploiting flaws in privileged code in order to execute arbitrary injected programs. | **Taxonomy:** Authorization.leastPrivilegeFailure  
**Top 25:**  
3. Buffer Overflow  
11. Execution with Unnecessary Privileges | $$-$$$$ |
| Stealing files on the machine and mail them as attachment | Combining above described vectors in order to place in the out spooler a well formed message containing sensitive files as attachment. | See other vectors | $$-$$$$ |
### TABLE XIV

**COMPARISON SUMMARY: RISK, AFFECTED MODULES AND AUDIT PLANS**

<table>
<thead>
<tr>
<th>Risk</th>
<th>Postfix</th>
<th>Xmsg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivering a message to a different recipient</td>
<td><em>local, qmgr, authorization</em> (relies on SELinux)</td>
<td><em>msgDistributor, msgReceive, authorization</em> (relies on Ethos)</td>
</tr>
<tr>
<td>Redirecting messages to other users</td>
<td><em>smtpd, authorization</em> (relies on SELinux)</td>
<td><em>msgSend authorization</em> (relies on Ethos)</td>
</tr>
<tr>
<td>Receiving mails from potentially dangerous senders</td>
<td><em>authentication</em> (relies on Cyress SASL)</td>
<td><em>authentication</em> (relies on Ethos.NaCl + usrlList)</td>
</tr>
<tr>
<td>Exposing or altering the content of message</td>
<td><em>encryption</em> (relies on OpenSSL + AuthDB)</td>
<td><em>encryption</em> (relies on Ethos.NaCl usrlList)</td>
</tr>
<tr>
<td>Executing malicious program</td>
<td><em>master, local</em></td>
<td><em>no privileged modules</em></td>
</tr>
<tr>
<td>Stealing files on the machine and mail them as attachment</td>
<td><em>combination of above</em></td>
<td><em>combination of above</em></td>
</tr>
</tbody>
</table>
7.3.1.1 Delivering a message to a different recipient

While analyzing this risk it is considered that the attacker exploit flaws in the authorization configuration and eventually also subvert the processes that will be cited later, it’s not considered the possibility of the attacker directly modifying the authorization configuration.

The Postfix documentation explains that several modules are in charge of delivering received messages. For the purpose of this case study we want to focus on the local module which is the agent for standard UNIX-like mailboxes (79). Potentially, every misplaced or corrupted write operation on an arbitrary input spooler folder can cause a mail to be delivered to the wrong user, this operation can occur in every part of the program that has been subverted. However it can reasonably be assumed that such operation is more likely to be performed by subverting the agent delegated to the mail delivery, the previous cited local. In this case the code to be sanitized for being robust again external tampering of parameters should be local and the chain that handles the message after its reception: in a simplified view we only rely on qmgr (queue manager).

In order for the attack to be possible, since the local module run with the userid of the recipient it’s also needed that the authorization policy is compromised. One can argue that sanitizing the previous cited modules is enough and there is no need to put particular effort in verifying the authorization system configuration, this is not correct because like it was explained before, every possible unrestricted write operation in the Spool-In can be dangerous, so the authorization system should be verified too anyway. In Postfix authorization is demanded to OS enforcement, in order to proper restrict Postfix processes, one has to define rules at OS
level. This will involve setting up a proper authorization mechanism like explained in 6.3.1, and extensively verifying several configuration files.

In Xmsg case the module which is in charge of delivering received message is `msgReceive`. Like in the Postfix case we analyze the chain after that deals with the mail reception: the module on which the analysis relies is `msgDistributor`. Since `msgReceive` runs as a virtual process on behalf of the user who will receive the message (2.1.2). Like every other Ethos application Xmsg exploits the integrated authentication infrastructures, and no additional code is required to bridge the application with the system. Moreover the portion of code to audit is much more limited to a tiny section.

7.3.1.2 Redirecting messages to other users

This risk is similar to the previous one, with the difference that the attack (placing a message in a spooler different that the one indicated by the sender) is performed at sending time rather than receiving. Potentially an attacker can write the targeted message in his own input spooler for further manipulations and eventually redirection. In Postfix the process delegated to fetch messages from clients and dispatching in the corresponding user input spoolers (incoming queue) is `smtpd`, so it can reasonably (and optimistically) be assumed as in the previous case that an attack will be directed towards this process rather than on an arbitrary module. In this case the code to be audited and carefully sanitized (since the process is directly connected to the network) is the `smtpd` module. Same considerations as the previous case are still valid for authorization auditing.
In the Xmsg case the correspondent target module is `msgSend`, the code to be verified and sanitized against malicious attempts to force writing in the wrong input spooler is the one related to the module itself. It is by construction not possible that the program writes in the wrong input spooler through parameters tampering, because the virtual process mechanism (2.1.2) handles authorization transparently at OS level and nothing has to be coded or bridged at APL. The only occurrence of a write operation in the wrong input spooler, could derive from a misconfiguration of the authorization policy (wrong assignment of input spooler folders to users), but in this case the effort required to audit it is significantly lower with respect to a traditional system.

7.3.1.3 Receiving mails from potentially dangerous senders

In its standard fashion emails are neither authenticated or encrypted, if someone wants to elevate the security level this is demanded to the server and client implementation and it’s not compulsory.

In order to avoid to receive mails from blacklisted dangerous senders, one strong countermeasure is using authenticated messages. Like explained before, this is quite uncommon for common users, and usually what it is done is simply applying Spam filters to the sender field in the email header. In a context where security is relevant this is not sufficient, it’s known that in SMTP the sender can be arbitrarily set, and anyway messages that are not signed can be easily tampered.

The Postfix Authentication system is based on Cyrus Simple Authentication and Security Layer (SASL) (80). The authentication system can work in different modes depending
on several configuration parameters. Misconfiguration of SASL results in broken, weak or missing authentication. It is possible that the system is working using cryptographic utilities which are broken or weak, because the client is not able to provide higher facilities or because the configuration settings are wrong and lower level utilities have been prioritized over better crypto standards. In the worst case, it is possible that the system exchange confidential information over unprotected channels and/or accepts connection without requiring authentication. In order to make sure that the application is robust against above described security issues, always providing the most reasonable and restricted estimate, it is necessary to verify that the software is correctly bridged with the library by analyzing \texttt{xsasl} calls across the modules and by verifying the code in the \texttt{xasl}. In order for Postfix to be aware of the authentication, it has to be configured in five entry points and about 30 SASL related parameter (81; 82) have to be set in the \texttt{main.cf} file, these have to be revised too. Finally, SASL relies on an authentication database to store users identification information, this can be managed in six different ways using different back-ends (83) whatever choice is made (typically PAM 6.2.1) this has to be separately verified in its own configuration about correctness and robustness.

In Xmsg case authentication is not a choice, meaning that it is compulsory and transparently forced by Ethos OS and it can’t be bypassed. This exclude a priori occurrences of missing authentication. Instead of having different variously secure mode of operation, Ethos authentication operates only using strong and fast cryptographic facilities (NaCl library 6.4.2), this avoid problem of weak of broken authentication. Verification of correctness of the bridging code is not necessary, because there is no bridging code, since the process is running on the behalf
of the sender the verification is transparently handled through the virtual process mechanism (2.1.2). The only thing that has to be verified is the Authorization Policy Specification (APS), specifically about the correctness of the users assignment in groups. No external authentication databases is required and users are simply user of the OS installation, all their sensitive files (keys and/or other secrets) are managed and accessible only to the OS kernel and are never provided to the application, Xmsg in this case.

7.3.1.4 Exposing or altering the content of message

Like remarked in the previous section by default e-mails are not encrypted, this means that the message is exposed and can be easily tampered by a skilled attacker. Currently it is very rare that messages are encrypted between clients and servers, usually only the exchange between SMTP relays is protected.

In Postfix encryption of the communication relies on the OpenSSL library, in order to have properly encryption the programmer has to put particular care in validating the code necessary to bridge the program to the services of the library. Across Postfix modules there are several calls to TLS services, and the code has to be audited and verified again misuse and parameter tampering. Even in this case in order to properly interface itself with OpenSSL, Postfix needs to be properly configured: like explained in 6.2.1 a lot of problems can arise from misconfiguration of TLS services. The problems range between a poorly performed encryption (due to compatibility issues across clients/server) to the worst case of complete absence of encryption of parts or even the whole message. In order to configure TLS in Postfix there are several parameters in the main.cf file (hundreds of occurrences of the *tls_* string).
Since Xmsg exchanges information through network streams, it benefits of the mandatory high-speed encryption introduced in Ethos by design. The encryption comes with no additional effort dedicated to setting up and using the cryptographic infrastructures. Every network stream included IPC and RPC are compulsory secured by the operating system 2.1.5. No explicit invocation to cryptographic utilities is necessary, so there is no bridging code to verify and moreover the stream is fully encrypted in all the communication not only on server side.

7.3.1.5 Executing malicious program

Execution of arbitrary code is always a very broad statement. In our case we want to limit to cases deriving from privilege escalation or injection of specifically crafted buffer (buffer overflow attacks). The second attack can be more or less effective if it attacks portion of the code which are running in privileged mode, in this way and attacker will be able to execute its own malicious code having access to all system facilities and data.

Of course as said many times during this comparison, these attacks can potentially occur everywhere in the code, but studying the Postfix architecture (84) we can reasonably identify two modules (among the ones of interest in this comparison, which are not the totality) that has particular privileges associated.

The first one is master, this is the main module and it’s the resident process that runs Postfix modules on demand. In order to perform is function master run as root, this implies that a buffer overflow attack on this module can potentially be able to run arbitrary code with root privileges. This occurrence is in fact very dangerous because the attacker can access to a variety of different possibility to carry out the second phase of it’s attack. In order to mitigate
this the portion of code to be audited and carefully programmed is the one related to the master module.

The second “special” module is local. It has been encountered before in this analysis with reference to user spoolers. In order to perform its task, the process transitions its UID between Postfix pseudo-user and the recipient of the message using the seteuid call. In order to be able to perform this it is owned by root. The family of setuid is considered dangerous in term of security (and it has been explained through previous chapters) and so particular care has to be adopted when using one of them. Such capability of switching between users, and then figuratively possessing multiple identities at the same time, put the basis for attacks of the type privilege escalation. Once the attacker gains control of the process through other techniques, he’s able to take advantage of the possibility of switching to a privileged status to gain additional permissions and continuing his hostile action. In the Postfix case it is reasonable to assume that, given these premises, the local module should go through a careful auditing to make sure it is clean and robust again common vulnerabilities.

In the Xmsg case there is no need to transit any kind of ownership on processes, and no process have any privileged status. This is possible again thanks to the virtual process mechanism described in 2.1.2, that allows Ethos to perform remote authentication avoiding the necessity of using a bridge process (like in the Postfix local case), privilege escalation is then eliminated by design. Moreover given the choice of GO as preferred APL application are buffer-overflow free and the system code employs dynamic resizing techniques to prevent it occurring at lower levels (6.8.2).
7.3.1.6 Stealing files on the machine and mail them as attachment

As summarized in the table this case is about a combination of previous analyzed risks. It covers a broad spectrum of possibilities the attacker has to be able to gain control of a sufficient privileged module and then pack sensitive information inside a well formed message and ship it through some user account to the desired destination. This can be very dangerous also in term of responsibility, because unaware user can be used as medium to anonymously gather sensitive data from the server. In the case of Postfix it has been provided and estimate which is a total of all previously described mitigation, but it has always to be reminded than every possible fragment of code can potentially be exploited and should be secured. Of course reasonably not all the code is really exploitable and the most interesting modules (in a functionality based comparison) are the ones cited so far. For example we haven’t mentioned cleanup, sendmail, maildrop the first is a module delegated to move and check message conformity the second two are for handling messages originating locally, we can assume that maybe security check on a local flow are less restrictive than on a network flow, but in the case the attacker has gained control of a process in the machine could be able to exploit this preferred path. In the Xmsg case same considerations applies, but the attack surface is much more restricted (as specified through the analysis). Potentially in the worst case the whole application has to be checked, data relative to the size of Xmsg and Postfix are at the beginning of the section (functional equivalent of Postfix can be considered 20% of the total size).
7.3.2 Monetary Damage

Delivering a message to a different recipient, Redirecting messages to other users: the estimate consider the best case in which there is an internal leaking inside a company environment, the attacked mail server deliver mails to wrong employers, but the information stay in the company. The worst case is if the hacked mail server belongs to some mail provider in this case users private information can flow to completely strangers and even attackers.

Receiving mails from potentially dangerous senders: in this case the estimated damage has been considered relatively low, since is just a matter to use a good anti-spam filter and getting rid of undesired messages.

Exposing or altering the content of a message: here the estimated damage it’s highly dependent from the target and purpose of the tampering. Regarding what kind of sensitive information have been read and leaked, or what communications have been altered the damage can go from “a prank” to a serious repudiation issue, possibly involving a lawsuit.

Executing malicious programs: even in this case the damage can be estimated regardless the purpose of the attack, executing arbitrary code on the mail server gives the attacker possibility to cause a long list of damages, hence the estimate ranges between the min and the max value according to the entity of the attack.

Stealing file on the machine and mail them as attachment: since is a combination of the above, the range covers all the above.
This comparison has showed that the effort necessary to achieve an high-level of security in an application running on a traditional OS is much higher than one running in the Ethos environment. Ethos provides to Xmsg a variety of integrating facilities, this lowers of 2-3 order of magnitude the amount of code security sensitive to be sanitized. As claimed during the previous chapters, since the security facilities are transparently handled at OS level the developer is relieved from the majority of the burden related to bridging with security services and checking the implementation against known threats.

All the considerations in this analysis can easily be extended to several other specific cases. abstracting from the specific risks, the operation of performing a remote connection and authentication on a server and then exchanging sensitive data is pretty common in every application that follow the classic client/server paradigm. Relevant issues on a server-side application are often related to restricting interaction with outside environment and access to internal information by the user, like in the case study that has been analyzed. Relevant issues on client-side application often involves creating a secure channel with the remote server to avoid spoofing of information during the communication.

Even though it is not the purpose of this comparison to have an absolute evaluation on the security level of Postfix or Xmsg, it has been denoted that the transparency in term of code required to bridge security services to application, and the integrated compulsory nature of security related facilities in Ethos are two fundamental factor that tend to result in a decreasing of possible flaws. This because some of them are eliminated by design (e.g. encryption is not
by-passable, hence we can’t have missing encryption). Given a reduction in the possible flaws, we can reasonably assume that the average level of security of Ethos application will be higher than in traditional OS. This of course is a general assumption and is subject to a variety of factors like in the traditional OS case.
CHAPTER 8

CONCLUSIONS

This thesis has been deeply driven by the Ethos OS project, and the work developed in the context of its research group. We believe that Ethos can be a concrete answer to address security threats that are affecting our computers every day. This work started because of the desire of understanding the impact that a different system architecture can have on the security properties of the application build on top of it. Ethos was a natural candidate to be involved in a process of this kind.

It has been showed that this process needs at first to understand the underlying architecture and environment in order to produce meaningful considerations on the software that will run on a particular system. Systems are very complex entities, so the previous requirement of understanding the underlying architecture, in order to define the security impact on applications, is not so straightforward.

After a quick overview on Ethos (Chapter 2) in order to make the reader familiar with some of the key concepts that have been used in following phases of the work, the focus has moved to the core of the problem, understanding how we can study, compare systems and ultimately the application that will run on them from the point of view of security.

It has been showed (Chapter 3) that approaching the problem from a detailed perspective it’s not sufficiently effective to produce general considerations, but instead what it has been called instances provided useful information to move to classes of problems. In the context
of these classes it has been possible to evaluate previously studied approaches, and understand their limits.

It’s been proposed in (Chapter 4) a taxonomy with the purpose of being widely applicable in term of coverage, and not dependent from the kind of system analyzed and its layering. Systems layer their infrastructure in ways that sometimes can be very different, in some cases part of these infrastructures and services are not even provided at system level, but are added subsequently, and so reside at different levels. The taxonomy provided a clear, layer-independent way of identifying sub-areas and a consistent naming across different systems.

After their identification, for every area it has been collected and produced a series of security infrastructures and services related to the specific section of the taxonomy (Chapter 5). Proposed mechanisms refer in general term to mitigations and preventions methodologies that have a direct impact on their corresponding area.

On the basis of the set of tools proposed the taxonomy and the resulting methodology has been practically applied to the Ethos case, and an extensive comparison between Ethos and community solutions related to common operating systems (Linux, Windows, OSX), it’s been carried out through all the Chapter 6. In this phase the methodology has been proved meaningful in its application to heterogeneous systems, in order to generate a security oriented understanding of their infrastructure.

The subsequent step it has been the extension of the analysis to the concrete case of applications running in different environments. In this context it has been shown how, by starting from a series of risks described in an informal language, and subsequently translating them
in practical mechanisms related to the specific application, it has been possible to relate all the information generated to areas of the previously proposed taxonomy. Using this mapping (Chapter 7) and the previous investigation on the systems carried out under the guidance of same structure (Chapter 6), it has been shown how it has been possible to isolate, compare and analyze related security efforts in both cases. This methodology managed to point out also information in term of where is distributed the security effort required, in the layering of different systems.

Using the taxonomy to analyze a case, that we have reasons to believe resulting an a lower security effort for the application developer (since Ethos operating system, rather than traditional OSs, is developed with the main purpose of security), produced a result coherent with the expected outcome. This showed the validity of this process even in the case of completely heterogeneous systems, by mean of the layer independence. The resulting methodology it has also been showed useful to selectively identify areas of improvements in the security infrastructure of an application and its underlying system.

8.1 Future Work

The Ethos project in which this work has been developed it’s a long term project started out in 2006. The system is in a prototype phase, so every aspect analyzed and exposed in this work is subject to improvements and modifications. The proposed approach is intended to be refined in its formulation, the taxonomy structure, and enriched in its enumeration of mechanisms, the preventions and mitigations associated. Another aspect is related to the evaluation of the proposed approach in different application cases, this could be performed along with the
growth of the Ethos user space environment, currently in development, and its usability. Final consideration of future work is related to the in-depth evaluation and testing of the Ethos operating system, this work can be the starting point of a future more accurate and rigorous evaluation and verification of the final system.
CITED LITERATURE


22. ORACLE: Java cryptography architecture (jca) reference guide.  

23. Microsoft: Cryptographic services.  


25. Microsoft: Cryptography.  


   Public Key Cryptography, pages 207–228, 2006.


   initialization in java. In
   Proceedings of the 15th European conference on Research in computer security,

   [Online; accessed 5-December-2011].


64. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A.,
   Wagle, P., and Zhang, Q.: Stackguard: automatic adaptive detection and
   prevention of buffer-overflow attacks. In
   Proceedings of the 7th conference on USENIX Security Symposium - Volume 7,


   implementations and weaknesses. http://www.blackhat.com/presentations/bh-
   9-November-2011].

68. LLVM: Automatic reference counting.


VITA

Name Francesco Costa
Birth date December 19, 1987
Email francesco.costa19@gmail.com
Mobile +1 312 218 8922

Education

2009–present Master of Science in Computer Science, University of Illinois at Chicago, GPA 4.0/4.0.

A Layer-Independent Taxonomy for Evaluating Application Security and its application to the Ethos OS


A Layer-Independent Taxonomy for Evaluating Application Security and its application to the Ethos OS


Transforms speed-up using CUDA: the case of MDCT for audio compression
VITA (Continued)

Research

2011–present  Computer Security Laboratory at UIC, supervision professor Jon A. Solworth

Experiences

2008  Gianluca Meneghini, Management of the porting of a legacy DB into a new ER structure through scripts and procedures creation on the SQL Server platform.

Reference: Gianluca Meneghini

Skills

Languages  Java, C#, (Proficient), C, C++, SQL, Python (Intermediate), Google GO, Android apps (Beginner)

Other Tools  Git, SVN, Rapid Miner, Weka, Basic Unix Skills, FireBug, LaTeX

Databases  MySql, SQL Server

IDE  Visual Studio, Eclipse, Netbeans, Dreamweaver

Web  HTML, CSS, PHP, Javascript, JQuery