Instituto Tecnológico y de Estudios Superiores de Monterrey

Monterrey Campus

GRADUATE PROGRAM IN MECHATRONICS AND INFORMATION TECHNOLOGIES

A Unified Software Security Enhancement Proposal Based on a Thorough Software Security Compendium.

THESIS

PRESENTED AS A PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF:

Master of Science in Information Technology

BY:

Armando de Anda González

Monterrey, N.L., November 2007

Instituto Tecnológico y de Estudios Superiores de Monterrey

Monterrey Campus

DIVISION OF MECHATRONICS AND INFORMATION TECHNOLOGIES

GRADUATE PROGRAM IN MECHATRONICS AND INFORMATION TECHNOLOGIES

The members of the thesis committee hereby approve the thesis of Armando de Anda González to be accepted as partial fulfillment of the requirements for the Degree of Master of Science, in:

Information Technology

Thesis Committee:

Arturo Galván Rodríguez, Ph. D.
Thesis advisor

Suku Nair, Ph. D.
Synodal

MSc Alejandro Parra Briones
Synodal

Graciano Dieck Assad, Ph. D.
Director of the Graduate Programs in Mechatronics and Information Technologies

November 2007

A Unified Software Security Enhancement Proposal Based on a Thorough Software Security Compendium.

BY:

Armando de Anda González

THESIS

Presented to the Graduate Program in Mechatronics
and Information Technologies

This Thesis is a partial requirement for the degree of
Master of Science in:

Information Technology

Instituto Tecnológico y de Estudios Superiores de Monterrey

Monterrey Campus

November 2007

To God, for the wonderful gift of life and
with whom anything is possible.

To the love of my life, Paola,
for being my inspiration,
my strength, my everything.

To my parents, Armando and Layda,
the greatest example of love and sacrifice.

To my aunt Martha,
for believing in me and
keeping this dream alive.

To my aunt Sayde,
my second mother,
for your prayers and guidance.

To my brother Juan, and to
my sisters Layda and Diana,
no matter the distance always
united.

To my many relatives and
friends who are such an
important part of my life.

Acknowledgements

First of all, I want to thank the members of my dissertation committee. Thanks to my principal advisor, Dr. Arturo Galván for his time, guidance, advice, and for believing in my proposal. While at Southern Methodist University, I worked under the supervision of Dr. Suku Nair, and I thank him for introducing me to me this fascinating subject and giving me the freedom to explore it. And to professor Alejandro Parra, whose technical competence is admirable, thank you for being so kind in accepting my invitation to review this work.

I must thank Dr. David Garza and Dr. Hesham El-Rewini for making my time at SMU possible; it was such a gratifying and growing experience.

Special thanks to Ana Isabel Cerda, whose advice on the little things made a huge difference.

Finally, I must thank all the support I have received from professors and fellow students during all these years of graduate and undergraduate school at UAC, SMU, and ITESM. They all contributed to the preparation and development of my skills and values to face all challenges.

Abstract

The research presented is to fulfill the requirements of master degree in Science of Information Technologies of ITESM (Instituto Tecnolígico y de Estudios Superiores de Monterrey). The use of information systems has augmented enormously. Computers have become a widely used tool in all disciplines and a medium to facilitate many aspects of our lives, but at the same time, vulnerabilities that endanger the personal data, reputation of organizations and lives of users are being discovered regularly. This dissertation is a fusion of knowledge which aims to provide a panoramic view at the problem and main countermeasures to developers, testers and end users. It will also serve as a break down of the problem, so the dissertation may be employed as a starting point for interested entities, whether research individuals or organizations, to facilitate the communication, exchange of data, combination and evaluation of information in the subject. This research will be useful in the avoidance of overlapping work and assist in the development of parallel efforts that could easily converge to bring us closer to making software behave. Several techniques and strategies have been proposed to mitigate the threats, but it has been a disseminated effort which makes more difficult the tasks of gathering, exchange and comparison of information, hence making it difficult to identify where we stand. This work is motivated by the current importance of security in software, and the work pretends to be a guide of Software Security Topics being its main contributions a proposed Questionnaire for Security Enhancement in the Software Life-Cycle, the coherent fusion of knowledge in the field, and at the same time break down of the problem of security.

Contents

Chapter 1

Introduction

Today, information systems are indispensable; they are a part of our every day lives. From space exploration to medical interventions, whether used for shopping or to pay your bills, to play games or talk to friends, there almost isn't a science, profession, or person that doesn't use or is affected by them in some way. Technology makes our lives easier because it decreases work times, hence, increases productivity. The arrival of the Internet became the "big bang" for computer-related business. It opened doors of opportunity by providing an infinite source of clients, but it also left a window open for criminals. In a world with exponential development of faster computers, faster networks, larger hard disk capacity, larger volumes of information, and larger amount software, is security catching up?

1.1 Problems Behind Software Security

System Security is not a new subject. However, enterprise security is currently viewed as a commodity, the implementation of password schemes and integration of devices such as firewalls are providing a bad sense of security; the problem is not properly addressed. The fact is that software security is the genesis of network security, since a correct design of the protocols and algorithms and of the way our systems communicate would prevent the occurrence of unexpected malign events.

> *The wire protocol guys don't worry about security because that's really a network protocol problem. The network protocol guys won't worry about it because, really, it's an application problem. The application guys won't worry about it because, after all, they can just use the IP address and trust the network.* -Marcus J. Ranum

Developers already know that one can not magically make software secure, and they also have known about vulnerabilities and their consequences for some time now. So, if there is awareness of the need for security, why are the number of incidents and vulnerabilities increasing exponentially? We can not solve what we can't understand. But why can't we fully understand software security?

1.1.1 Software Security Difficulties

1. **Lack of common basic terminology.** The essential terms such as bug, flaw, vulnerability, weakness, exploit, attack, and security are not standardized and are being applied at the convenience of researchers (rfc2828 [153], X.800, IEEE 1990. Glossary of Software Engineering Terminology [41],...). This makes it difficult to compare and combine results. The development of such standard way of communication is very important for the fusion of knowledge, understanding of the field, and development of tools and procedures that can be used for the coherent, comprehensive and systematic analysis.

2. **There is not a standard Taxonomy, Collection or Classification of Vulnerabilities or Weaknesses.** There exist many different taxonomies, classifications, enumerations, or different ways of organizing the software security problem space (see appendix A), but no standard has been adopted. The complexity provoked by the large variety of threats is what truly makes security weaknesses difficult to catalog and understand.

   For example:

   - There are some who divide the problem in terms of malicious software (viruses, worms and spy-ware) and bad implemented software. However, malicious software usually takes advantage of software bugs, so this classification is not mutually excluding.
   - There are some others others who classify the vulnerabilities in terms of where, during the software life cycle, the vulnerability appears, such as design, implementation or configuration. The disadvantage in this case is that it is often unclear the single place it came from.
   - Some incidents are sometimes triggered by rare, non-intentional specific situations (such as the y2k bug), and there are others purposely taken advantage of by evil doers (such as SQL injection).
   - Some platforms and languages are more responsible for a great deal of vulnerabilities reported such as PHP, C, C++ and SQL.
   - There are many vulnerability collections.

     There are different organizations that serve as a database for vulnerabilities and bug reporting, each of these with their own formats.

     There are private and commercial databases such as the CMET database at the Air Force Information Warfare (AFIW); the collection of Mike Neuman(1995); the database of the Australian Computer Emergency Response Team (AUSCERT); the internal vulnerability database at Netscape and the one at Sun. Others, such as iDefense and Tipping Point, handle a vulnerability market. They buy vulnerabilities users find (the prices are not published) and maybe resulting in knowledge that would not otherwise be found.

There are also public databases such as the one of Internet Security Systems (ISS), the Common Weakness Enumeration, Common Vulnerabilities and Exposures, Kao's Unix Security Library, NegativeZero and academic publications. There are also computer security mailing lists, such as BUGTRAQ, NTBUGTRAQ, IDS, Best of Security. Moreover, there are advisories like CERT, CIAC, UNAM-CERT, AUSCERT, L0pht Security Advisories and Vendor Security Bulletins.

- Software trends affect vulnerability trends. The massive emergence of web applications has placed web vulnerabilities as the leading causers of advisories in the last 2 years. The evolution of software has also reached new types of devices, and with this several new threats (such as in cell phones and game consoles). Software security is a moving target.

3. **It is difficult to measure the impact of vulnerabilities.** There does not exist a universal ruler to measure and compare the impact of all types of incidents. How does one evaluate the impact of a security threat? There are small bugs that affect millions of computers but cause no considerate amount of harm (for example freezing instant messaging software). And there are large bugs that may affect just one organization, but produce stratospheric economic damages or ruin the organization's reputation. How can one measure a damaged reputation and accurately calculate the number of users affected by a flaw?. More importantly, how can one calculate the time and money to invest in a given system before and during it's development and deployment, in order to guarantee secure software, or at least greater profits than losses?.

4. **There is not complete information disclosure.** History tends to repeat itself. This is why prompt and complete information has to be deployed in order to learn from past mistakes. The historical data of failures will aid in making them less common since this may be used to generalize, compare and communicate findings within the software security individuals and organizations.

   It could be useful to look at the economic loss by companies in proportion to their investments in order to understand the real impact of lack of computer security. However, security breach events are not something organizations want to advertise to their clients. This makes it possible for the same incident to happen to several organizations, maybe even by the same criminal.

5. **There is a disseminated effort.** Divide and conquer is suggested between software engineers for system organization, but in terms of security we are too divided. As mentioned above, even though there are several research institutions and individuals working on mitigation of software security problems, the lack of a standard way of communicating the achievements in the field has been the cause of disseminated work, sometimes redundant and overlapping research that has made the software security field a complex one to approach. Sure, there is no silver bullet for software, but software security is making the werewolf scarier.

6. **All for One and One for All.** It takes everyone in a project to cooperate for the system to be secure, however it only takes one person to make it insecure.

1.1.2 The Trinity of Trouble

Greg Hoglund and Gary McGraw identified three factors that work together to make software risk management a greater challenge. These factors are the "Trinity of Trouble" [85] .

Connectivity

Growing Internet connectivity has increased the number of attack vectors as well as the ease of exploiting software. As people, businesses, and governments become more dependent on communication that information systems provide, they become vulnerable to exploitation from remote sources. From cell-phones to cars and refrigerators, new devices are getting on-line implementing new features and bringing along new vulnerabilities.

Extensibility

Extensible systems make security harder. Analyzing the security of an extensible system is a much more complicated task than focusing in an unchangeable system. Sun Microsystems' Java platform and Microsoft's .NET Framework are designed to accept mobile code updates and extensions that let system functionality evolve incrementally. Applications such as word processors, spreadsheets, and Web browsers allow extension through scripting, controls, components, and applets. Preventing software vulnerabilities from slipping in as unwanted extensions is a major challenge. Understanding how a system may be extended in the future is essential to getting a handle on system risks.

Complexity

The size of programs used to be only a couple of kilobytes, where you could distribute them in diskettes. Now a days, software installation takes one or more CD's or DVDs. The more code, the greater the complexity and the more opportunity to make mistakes. The complexity of operating systems has increased dramatically during the past decade. For example, as we can see in Figure 1.1, Microsoft's Windows XP has 40 million lines of code compared to the 3 million in Windows 3.1, and windows vista is estimated to have over 50 million lines of code [11].

1.1.3 The Thirteen Security Snares

Even though the Security Snares [64] are described in the Service Oriented Architecture security context, their knowledge in Software Security in general is of great importance.

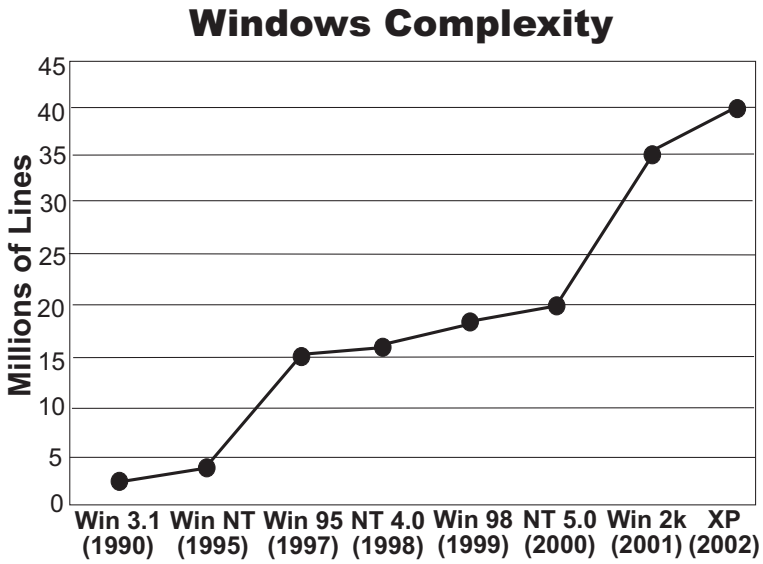

Figure 1.1: Windows Complexity

1. **Assuming the vendor will take care of security.** Buildings and cars go through thorough security inspections and are not put to use until these are passed. There isn't such examination of software and many vendors wills "check off the security box" by throwing in some crypto features and calling it a day.

2. **Not asking about security at all.** It is common to find IT organizations and large companies without a dedicated internal security staff.

3. **Asking about the wrong kinds of security things.** Firewall is not absolute security. There are also companies who just invest in reactive approaches rather than integrating the security from the start.

4. **Allowing discomfort with the technology to overcome the need for software security.** Familiarity with firewalls, SSL and operating systems is of great importance, but one should not avoid questions like, "How can you demonstrate to us that this product is secure?" Getting outside your technology comfort zone is often elucidating and educational.

5. **Relying on a cursory risk assessment.** Smart organizations know how to manage risks, and they make conscious decisions about where to focus their limited resources. However evil people's choice of attack shifts quickly and the destination of the mitigation resources and effort needs to keep up.

6. **Believing you're secure for no apparent reason or for the wrong reasons.** The lack of evidence of being insecure does not mean you are secure.

7. **Misapplying vulnerability metrics.** Rather than asking the vendor directly about, some security engineers incorrectly rely on public metrics such as the

number and severity of publicly reported bugs to determine the product's quality. Whether these metrics are correlated with actual product security remains an open research question.

8. **Trusting the vendors (too much).** Vendors might intentionally or unintentionally give inaccurate results. A vendor who performs penetration testing, for example, might not have tested the product or version being considered, thus the testing's value might be reduced.

9. **Building a proof of concept that ignores security "for now."** This concept is common in prototypes that evolve into systems, postponing security. Don't leave security for later-ever.

10. **Believing security is somebody else's job.** This could be a variant of "assuming that the vendor will take care of security," or it could be a symptom of an organization in which security specialists aren't responsible for the security of the development systems in use. Software security is everybody's job.

11. **Giving up hope.** The security specialist only has a limited amount of influence over purchasing decisions. Why spend the time questioning a boss or vendor or analyzing security when his or her actions are unlikely to impact the procurement or deployment decision?

12. **Putting too much weight on security standards and security features.** Standards such as SSL (for Web servers), and S/MIME (for email) are widely perceived to provide security. Too many organizations fail to understand that although these standards are important, they don't actually do anything to secure a system. An implementation bug or an architectural flaw in a product can leave a system that's completely standards- compliant completely insecure as well.

13. **Doing it all yourself.** Organizations don't ask the security question because they plan to come to their own conclusions by performing their own hard-core analysis and testing.

This dissertation aims to provide a understanding in the current status of software security by presenting the information regarding the justification for the field, by identifying the threats and their root causes and by presenting the current main stream work of countermeasures.

1.2 Problem Definition

The importance of security in software is already known; however this effort has not yet been reflected or concentrated in one single place. People in the field have their own definitions, taxonomies, strategies and opinions that have served them to work on a part of the problem. This approach has made it difficult to understand the global

status of the progress in solving the problem. Many definitions are not universal within the software security domain, fact which causes work to be overlooked by some, and maybe guide others to a dead end path that someone else had previously encountered. Without a standard way of communication between those in the software security effort, we are dividing without conquering our dilemma.

1.3 Objectives

The main objective of this dissertation is to bring understanding of the software security field through a fusion of knowledge of the main topics.

Particular objectives:

1. To explain and justify why security is important (Health, reputation, money, information loss).
2. To explain why Software Security is difficult.
3. To explain why software is insecure.
    - It inherits from the fact that there is no silver bullet for software.
    - Vulnerabilities brought into the system through the design, implementation and configuration.
    - Complexity leads to insecurity.
    - Features VS Security.
    - Identify which software is vulnerable: COTS, Open source, legacy, in-house development, PDA's, Hand held Cell phones Game Consoles, Medical applications, military applications, bank systems, scientific, etc..
    - Describe what is under software? Hardware, OS, libraries, components, insecure environments.
4. To present a catalog of attackers and describe their skills, motives and objectives.
5. To expose and describe the leading threats and trends. [78, 176]
6. To present the principal mitigation tools, practices and strategies.
    - To present the practices, techniques, and procedures [32, 179], and describe when in the development process these should be applied.
    - To present a catalog of tools for software security [120, 121, 57, 96], along with different available tool evaluation results [197, 103, 155, 24, 109, 201].
7. Describe the current Software Security Metrics and the need for them.

1.4 Hypothesis

This fusion of knowledge will provide a starting point for anyone interested in the topic, and a comprehensive descriptions of the problem and main countermeasures. It will provide a clear, non-redundant, break down of the problem and coherent integration of the information which will aid interested people, whether research individuals or organizations, to facilitate the communication, exchange of data, combination and evaluation of information within the subject. This research will aid in the avoidance of overlapping work and help to develop parallel efforts that could easily converge to bring us closer to making software behave.

1.5 Methodology

The research parted from a bibliographic revision of the state of the art and a study of Tools, methodologies, practices and recommendations widely respected. The primary source of reference for the research will be the information presented in vulnerability Databases such as the Computer Emergency Response Team (CERT's) web sites, since they provide the historically important information concerning vulnerabilities and their effects, and they also point to the relevant documentation of the advances in the topic. Important incidents of breaches and computer crimes are obtained from news sources to identify the tendencies, along with the legislative and technical countermeasures adopted to mitigate them. From there, this work analyzes the secondary sources of this documentation.

Chapter 2

The Software Security Context

Information systems have become a pillar for the function of organization's daily activities and many individual's day to day routines. Information is obtained, services are provided, and communication is established with clients, providers, partners, and sadly with evil-doers as well. A criminal act is an event that usually occurs when one man has and another wants. Unfortunately, this has always been part of human history, and it will most likely always be, for we have seen how criminals evolved and adjusted themselves to the information age in no time. We must remember that the main functions of computer systems are storage, manipulation, transformation, and control of information; any unwanted alterations to the system will bring unwanted consequences.

How does software fit in the information security scene? Mark Stamp pointed out that "Software implements practically all information security and serves as foundation on which all the other measures sit" [169]. Schneier went further when he said that if we didn't have as bad software security, the time, money, and effort spent on network security wouldn't be as much as today [186].

The revolutionary waves of technological advances, have also brought along a wide variety of gadgets or devices; all of which are run by software. We have game consoles, cars, medical instruments, cell phones, PDA's, digital cameras, military systems to name a few, and most of these with some type of inter-connectivity. This results in the endangerment of our information, money, health, etc...

2.1 Fundamental Concepts

Software Security can be summarized as follows:

Definition 1. ***Software Security****: The ability a system has to prevent failure and to achieve its design objectives in spite of failure, to resist or withstand anticipated attacks and to recover rapidly, with minimum damage, from attacks that cannot be resisted or withstood.*

Other important concepts within the software security scope are listed below:

Definition 2. ***Security Policy****: Set of guidelines that indicate what and how resources should be protected by explicitly specifying all the valid ways in which the components of a system are allowed to interact.*

Definition 3. ***Security Mechanism****: A process (or a device incorporating such a process) that is designed to detect, prevent or recover from a security attack.*

2.1.1 Security Services

Definition 4. ***Security Service****: A processing or communication service that enhances the security of the data processing systems and the information transfers of an organization. The services are intended to counter security attacks, and they make use of one or more security mechanisms to provide the service.*

The Security Services are the following:

Definition 5. ***Confidentiality****: The property that information is not made available or disclosed to unauthorized individuals, entities, or processes [i.e., to any unauthorized system entity].*

Definition 6. ***Integrity****: The assurance that data received is trustworthy; exactly as sent by an authorized entity (i.e., contain no modification, insertion, deletion, or replay).*

Definition 7. ***Availability****: To ensure that a resource is accessible at the time it is required, Denial of service, or DoS, attacks try to reduce or impede information access or any system resource.*

Definition 8. ***Non Repudiation****: Service which Provides protection against denial by one of the entities involved in a communication of having participated in all or part of the communication.*

There are two types of Non-repudiation:

1. Non-repudiation, Origin: Proof that the message was sent by the specified party.
2. Non-repudiation, Destination: Proof that the message was received by the specified party.

Definition 9. ***Authentication****: The assurance that the communicating entity is the one that it claims to be.*

This service concentrates in the assurance of authenticity of both the source and destination entities participating in the communication.

In the context of software, what is a vulnerability? From the RFC 2828 [153] we get the following (“I” identifies a recommended Internet definition, “C” identifies commentary or additional usage guidance.):

2.1.2 STRIDE

After describing the security services, it is important to describe the threats faced by the application that interrupt the services. STRIDE is the acronym used at Microsoft [88] to categorize different threat types. STRIDE stands for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of privilege:

- Spoofing. Spoofing is gain access to a system by using a false identity. This can be accomplished using stolen user credentials or a false IP address. After the attacker successfully gains access as a legitimate user or host, elevation of privileges or abuse using authorization can begin.

- Tampering. Tampering is the unauthorized modification of data, for example as it flows over a network between two computers.

- Repudiation. Repudiation is the ability of users (legitimate or otherwise) to deny that they performed specific actions or transactions. Without adequate auditing, repudiation attacks are difficult to prove.

- Information disclosure. Information disclosure is the unwanted exposure of private data. For example, a user views the contents of a table or file he or she is not authorized to open, or monitors data passed in plaintext over a network. Some examples of information disclosure vulnerabilities include the use of hidden form fields, comments embedded in Web pages that contain database connection strings and connection details, and weak exception handling that can lead to internal system level details being revealed to the client. Any of this information can be very useful to the attacker.

- Denial of service. Denial of service is the process of making a system or application unavailable. For example, a denial of service attack might be accomplished by bombarding a server with requests to consume all available system resources or by passing malformed input data that can crash an application process.

- Elevation of privilege. This occurs when a user, program, or process with limited privileges assumes the identity of a privileged user to gain privileged access. For example, an attacker with limited privileges might elevate his or her privilege level to compromise and take control of a highly privileged and trusted process or account.

Definition 10. ***Vulnerability****: (I) A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. (C) Most systems have vulnerabilities of some sort, but this does not mean that the systems are too flawed to use. Not every threat results in an attack, and not every attack succeeds. Success depends on the degree of vulnerability, the strength of attacks, and the effectiveness of any countermeasures in use. If the attacks needed to exploit a vulnerability are very difficult to carry out, then the vulnerability may be*

tolerable. If the perceived benefit to an attacker is small, then even an easily exploited vulnerability may be tolerable. However, if the attacks are well understood and easily made, and if the vulnerable system is employed by a wide range of users, then it is likely that there will be enough benefit for someone to make an attack. [153]

Furthermore, a threat is described as follows:

Definition 11. ***Threat****: A potential for violation of security, which exists when there is a circumstance, capability, action, or event that could breach security and cause harm. That is, a threat is a possible danger that might exploit a vulnerability.*

Finally, the concept of attack is described below:

Definition 12. ***Attack****: An assault on system security that derives from an intelligent threat; that is, an intelligent act that is a deliberate attempt (especially in the sense of a method or technique) to evade security services and violate the security policy of a system.*

2.2 What software is vulnerable?

"Not Mine!!!" come to mind? The real answer is all. As Cheswick and Bellovin put it [34]: "Any program no matter how small can harbor security holes."

- Web Applications.-Web applications have become a widely used form of interaction in our daily lives. E-commerce systems are the most affected.
- Operating systems.- These are fundamental pieces of software and their complexity is increasing at gigantic steps (the more complex the software is, the more insecure it is).
- Legacy systems.- These are the black boxes of organizations. We use and depend on what we do not understand.
- COTS (commercial of the shelf).- This is another enigmatic black box which, unless the software counts with some type of certification, you have no warranty that the system is error free but to test it yourself.
- Open and Closed Source.- Neither of them is free of bugs. Security through obscurity has never been successful, however liberating the intestines of the system has not been proved to decrement vulnerability occurrence [116].
- PDA, Hand held, Cell phones and similar.- It is now common to find advisories for software running on these devices.
- Network Devices.- Routers, Switches, and other similar devices may run vulnerable code.

- Game Consoles.- These is a multimillion industry with connectivity where treats have been discovered.
- Medical applications.- Critical systems, but not free of bugs.
- In house development, military and bank applications, cars, appliances... in a near future, many more...

2.3 Why is software vulnerable?

We have already mentioned the "Trinity of Trouble". Complexity, Connectivity and Extensibility are definitely great vulnerability magnets. Here are some more factors that explain why software is still insecure:

- Security is not a switch that one turns on an everything is fixed.
- Software development. The lack of complete security in software is inherited from the fact that there is no Silver Bullet for software in general[25] for software. Software development is a complex creative process prone to errors by being human made.
- Developers are humans, and as such, make errors in during design, implementation, and deployment.
- Can't afford Security. Sure, security implies different types of investments, but what one can't afford is maintenance costs, legal disputes, damaged reputation.
- "Don't need it" mentality (denial). Developers are artists, and telling them that they are doing something wrong has to be done carefully.
- Pressure to compete in the market (ship it Tuesday and get it right in the next version!!)
- Ignorance of insecurity.

  "Ignorance is Bliss"

  - Thomas Gray.
- No Validation.
- Lack of exception handling.
- The use of obscure legacy systems. We should fear what we do not understand.
- Incorrect testing. My computer is a controlled environment, uncontrolled environment: the real world.

- Featuritis!!. KISS (Keep it small and simple, or Keep it simple stupid!!). As the Trinity of trouble indicates, Complexity leads to insecurity. People buy features, not security, but the more the features the greater the chance of having bugs.
- Is all code ours? How can we be sure that libraries, platforms, operating systems, frameworks, compilers, components, protocols, algorithms are secure? Figure 2.1 shows the code under applications that represents areas where control is out of our hands, making it difficult to assure quality and security.

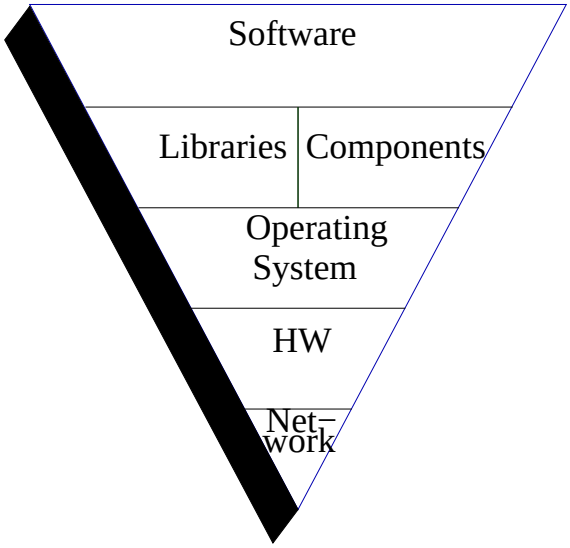

Figure 2.1: What is Under Software?

Chapter 3

Attackers and Motives

Technological advances and interconnectivity implementations have brought along a new type of sophisticated delinquents who are able to terrorize cities with the press of a button and rob enormous amounts of money. They can do this from the opposite side of the world; maybe places where their activities are not illegal. Adding the fact that organizations being attacked do not consider it worthwhile to pursue these individuals, and are probably not willing to risk the adverse publicity of admitting the success of an attack once arrests are made, the evil-doers realize that there is little chance of getting caught or convicted.

An attacker is defined as follows:

Definition 13. ***Attacker****: Individual who exploits the vulnerabilities of systems in order to make them behave in an unexpected way, to provoke the violation of one or more of the security services in order to reach their goals.*

It is very important to comprehend who the attackers are, undestand their incentives, and be aware of their skills in order to be prepared against them. Understanding an attacker's motivations can aid in the identification and protection of the modules of a system most likely to be exploited while the knowledge of skills and techniques of attackers allows one to test systems as an attacker would, but before him. This chapters describes the attackers and the motives behind their actions.

From the novice to the expert adversaries, there are variety of skill levels and motivations [140, 72], where particular kinds of attackers tend to have certain motivations [87]. Figure 3.1 shows the trends in attack sophistication and the skills of attackers.

3.1 Law enforcement Entity

We start with the good guys, although some might have a different point of view in the adjective employed. These are organizations who have motives of political influence, blocking illegal (violation of Availability) or subversive content. They also perform "Eavesdropping for our protection" (violation of Confidentiality) in search for threats.

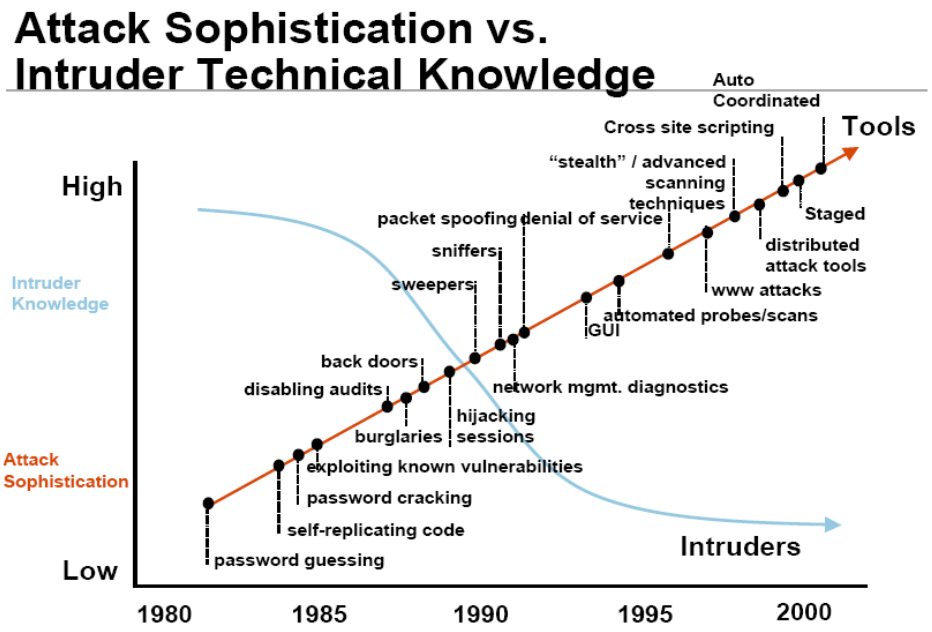

Figure 3.1: Trends in Attack Sophistication and Intruder Knowledge [29]

These organizations do not worry about non-traceability, have a high skill level, and will use maximal stealthiness. Due to their expertize, they have a high-expected success.

The NSA, for example, is considered by some the largest spy agency [130, 63] (larger than the CIA and FBI, for example), but it possesses the most advanced technology for intercepting communications. There is a current scandal which has indicated two new and significant elements of the agency's eavesdropping[182]:

1. The NSA has gained direct access to the telecommunications infrastructure through some of America's largest companies.

2. The agency appears to be not only targeting individuals, but also using broad "data mining" systems that allow them to intercept and evaluate the communications of millions of people within the United States.

This type of agencies have the advantage of being able to tap directly into the major communications switches, routing stations, or access points of the telecommunications system.

3.2 Security Auditors

This is a highly skilled individual who's goal is to improve security. This person who uses advanced computer skills to attack computers but not with a malicious intent. They use their skills to expose security flaws with the final goal correcting and protecting. The auditors do not worry about non-traceability but may require stealthiness.

The Indian Computer Emergency Response Team defined a set of expectations from an Auditor [175], here are some of them:

- Verifying possible vulnerable services only with explicit written permission from the auditee.
- Refrain from security testing of obviously very insecure and unstable systems, locations, and processes until the security has been put in place.
- With or without a Non-Disclosure Agreement contract, the security auditor is ethically bound to confidentiality, non-disclosure of customer information, and security-testing results.
- Clarity in explaining the limits and dangers of the security test.
- Seek specific permissions for tests involving survivability failures, denial of service, process testing, or social engineering.
- The scope clearly explains the limits of the security test.
- The test plan includes both calendar time and man-hours and schedule of testing.
- The security auditors know their tools, where these come from, how they work, and have them tested in a restricted test area before using the tools on the customer organization.
- High risk vulnerabilities such as discovered breaches, vulnerabilities with known, high exploitation rates, vulnerabilities which are exploitable for full, unmonitored or untraceable access, or which may put immediate lives at risk, discovered during testing are reported immediately to the customer with a practical solution as soon as they are found.
- Reports state clearly all states of security found and not only failed security measures.
- Reports use only qualitative metrics for gauging risks based on industry-accepted methods. These metrics are based on a mathematical formula and not on feelings of the auditor.
- All communication channels for delivery of report are end to end confidential.

3.3 Unethical Hackers

The definition of hacker varies. Some people use the term to generalize for all the cybercriminals [170], other experts define the term as technologically skilled good guys who perform harmless acts with the objective of learning. The Unethical descriptor is employed in this description in order to describe the highly-skilled individual who looks

to Harm Systems. They are people who violate system security with a malicious intent and are characterized for having advanced knowledge of computers and networks and the skills to exploit them. This attackers look to destroy data, deny legitimate users of service, or otherwise cause serious problems on computers and networks. They may also target you to show off their skills and expertise to their peers and maintain a certain reputation. Other motivations for these attackers is recreation, sense of belonging, economic gain, intellectual gain, and malevolence. Even if they have little malicious intentions, they can cause extreme damage to your systems.

In March 1997, one teenage hacker penetrated and disabled a telephone company computer that serviced the Worcester Airport in Massachusetts. As a result, telephone service to the Federal Aviation Administration control tower, the airport fire department, airport security, the weather service, and various private airfreight companies, was cut off for six hours. Later in the day, the juvenile disabled another telephone company computer, this time causing an outage in the Rutland area. The lost service caused financial damages and threatened public health and public safety. On a separate occasion, the hacker allegedly broke into a pharmacist's computer and accessed files containing prescriptions[142].

3.4 Script Kiddies

It is important to distinguish between the sophistication of the attacker and the sophistication of the attack. Persons with very limited technical ability can now launch very sophisticated attacks thanks to the availability of highly sophisticated tools. Script Kiddies are considered the most prevalent but the least dangerous of the attackers. They are unskilled users who must rely on downloading automated hacking software from web sites and use it to break into computers, but they lack the knowledge to develop the tools themselves or understand how the attacks work. They tend to be young computer users with almost unlimited amounts of leisure time, which they can use to attack systems. This group represent the lower-end of a continuum of attackers with a variety of skill levels, resources, and organization. A script kiddie will not worry about stealthiness or non-traceability. The motive of these cyber vandals is more often than not curiosity. Other incentives might include economical gain, recreation, reputation, sense of belonging, malevolence, damage, or recognition.

3.5 Industrial Spies

These are the corporate raiders who look for competitive intelligence through industrial espionage. Their intentions may be recruitment, subversion, commercial advantage or damage, tacit collusion, and misinformation. A person is hired to break into a specific network or computer and steal information, aiming to increment their market shares.

In June 1, 2007, Oracle filed the latest amended complaint, claiming that individuals from SAP stole a large ration of software code. Oracle sued SAP for breach of contract and copyright infringement, alleging that SAP's TomorrowNow business

unit violated copyright law by distributing Oracle material to its customers.

In the complaint Oracle alleged[67]:

"Oracle brings this lawsuit after discovering that SAP is engaged in systematic, illegal access to – and taking from – Oracle's computerized customer support systems. Through this scheme, SAP has stolen thousands of proprietary, copyrighted software products and other confidential materials that Oracle developed to service its own support customers. SAP gained repeated and unauthorized access, in many cases by use of pretextual customer log-in credentials, to Oracle's proprietary, password-protected customer support website. From that website, SAP has copied and swept thousands of Oracle software products and other proprietary and confidential materials onto its own servers. As a result, SAP has compiled an illegal library of Oracle's copyrighted software code and other materials. This storehouse of stolen Oracle intellectual property enables SAP to offer cut rate support services to customers who use Oracle software, and to attempt to lure them to SAP's applications software platform and away from Oracle's."

3.6 Insider

Unlike the rest of the types of attackers, Insiders attack you from the inner corporate territory boundaries. One of the largest information security threats to business come from the people closest to you. The skills of this group of attackers may vary, however, it is likely there is much knowledge of the system with which less technical knowledge is needed in order to attack. They know what your most valuable information assets are, where they are stored, and how to access them. But not all inside enemies are full-time employees of your company. Contractors, temporary workers, and former employees may have privileged access to your systems with little control over or oversight of their activities.

In 1992, a fired employee of Chevron's emergency alert network disabled the firm's alert system by hacking into computers in New York and San Jose, California, and reconfiguring them so they'd crash. The vandalism was not discovered until an emergency arose at the Chevron refinery in Richmond, California, and the system could not be used to notify the adjacent community of a noxious release. During the 10- hour period in which the system was down, thousands of people in 22 states and 6 unspecified areas of Canada were put at risk[53].

A study by the secret service in 2005 [105] revealed:

- A negative work-related event triggered most of the insiders' actions.
- Sixty-two percent of incidents were planned in advance.
- Eighty percent of the insiders exhibited unusual behavior in the workplace prior to carrying out their activities.

- Fifty-seven percent of insiders exploited systemic vulnerabilities in applications, processes and/or procedures.
- Thirty-nine percent used relatively sophisticated attack tools.
- Sixty percent of insiders compromised computer accounts, created unauthorized backdoor accounts or used shared accounts in their attacks.
- Most incidents were carried out via remote access.
- Less than half of the insiders (43%) had authorized access at the time of the incident.
- Insider activities caused financial losses (81%), negative impacts to business operations (75%) and damage to the organizations' reputations (28%).
- 78% – Secret Service/Computer Emergency Readiness Team (CERT) Study

3.7 Cyberterrorist

Dorothy Denning described cyberterrorism as "... the convergence of cyberspace and terrorism" [53]. She continues "it refers to unlawful attacks and threats of attack against computers, networks, and the information stored therein when done to intimidate or coerce a government or its people in furtherance of political or social objectives". Cyberterrorists motivation may be defined as ideology, or attacking for the sake of their principles or beliefs. One of the targets highest on the list of cyberterrorists is the Internet itself identity theft Social protesters (hactivists) publicity, hindering and disruption, patriotism, and social or political change. They may want to deface your public Web site and use it as a venue for their political messages. Such political events occur relatively frequent, numbering in the hundreds per year.

Their objectives are identification and information, publicity and propaganda, recruiting, political action, disruption, intimidation, economic espionage, training, preparation for information warfare, misinformation, and sabotage.

In 1997, ethnic Tamil guerrillas swamped Sri Lankan embassies with 800 e-mails a day over a two-week period. The messages read "We are the Internet Black Tigers and we're doing this to disrupt your communications." Intelligence authorities characterized it as the first known attack by terrorists against a country's computer systems[14]. In the same year, Spanish protestors bombarded the Institute for Global Communications (IGC) with thousands of bogus e-mails. These affected the San Francisco based ISP's users and support lines were tied up with people who couldn't get their mail. The protestors also spammed IGC staff and member accounts, clogged their Web page with bogus credit card orders, and threatened to employ the same tactics against organizations using IGC services. They demanded that IGC stop hosting the Webs site for the Euskal Herria Journal, a New York-based publication supporting Basque independence[53].

Chapter 4

Top Vulnerabilities

There exist thousands of vulnerabilities identified, some of these very specific for a particular software or platform, and it only takes one vulnerability in your system in order to be insecure and have drastic consequences. Trying to describe the whole population of vulnerabilities would be a marathonic task, and even if such thing was attempted, new vulnerabilities are constantly discovered, so it would be virtually impossible to ever be done describing them all. In learning about vulnerabilities one should focus in those related to the system to be implemented, or those similar to the one we will adopt or develop. Nevertheless, a few vulnerabilities account for more than 60% of the total of errors reported, so it is very important to become familiar with them. This chapter aims to describe these top vulnerabilities that represent the majority in the last 5 years.

4.1 Buffer Overflows

Buffer Overflows are one of the most important and frequent types of software vulnerabilities. According to [37], in 2005 this vulnerability was overthrown as the leading type of vulnerability reported, even though the number of advisories for this type is still growing at a fast pace. The connectivity that the Internet brings has stimulated other types of threats, such as Cross-site Scripting and SQL Injection, to grow rapidly and surpass the number of incidents reported by buffer overflows. The circumstances and strategies that trigger a buffer overflow are many and entire books are dedicated to explaining them, however, this section aims to provide a clear understanding of the characteristics of the vulnerability and ways it is exploited.

As mentioned in [70], buffer overflows originate from poorly constructed software programs, however they originate from malicious software as well. Such was the case first reported and widely publicized instance, which arrived in 1988 as part of Robert T. Morris' Internet Worm. From then on, buffer overflow vulnerabilities accounted for a large portion of the vulnerabilities. A buffer is a contiguous block of memory where a program stores different types of data. A buffer overflow occurs when data is written or read beyond the upper or lower limits of the buffer (this is why it is also referred to

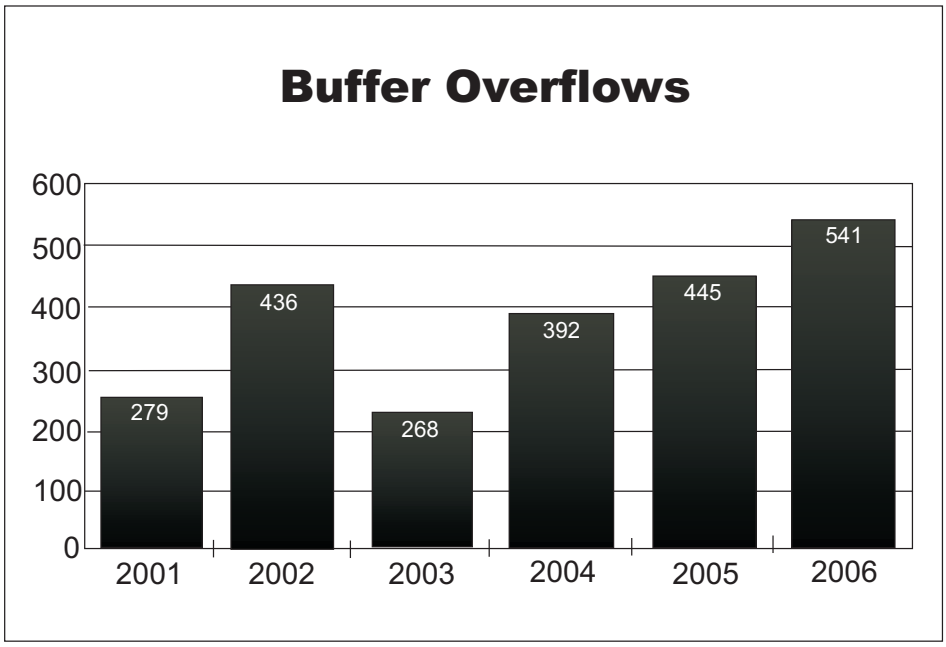

Figure 4.1: Buffer Overflow vulnerabilities reported by year (CWE)

as buffer overrun or underrun). Buffer overflows may occur on the stack, on the heap, in the data segment, or the BSS segment (used for uninitialized global data), and may overwrite from one byte to an unlimited number of bytes of memory outside the buffer, which allows an attacker to overwrite information such as return address on the stack, a function pointer, or a data pointer. The result is a change in the program's control flow.

Basic Example of a Buffer Overflow [199]:

```
char check_login( char *name)
{
char isAdmin = ‘‘N’’;
char usr_buff[10];

if (strcmp(name, ‘‘admin’’) == 0)
isAdmin=‘‘Y’’;

strcpy(usr_buff,name);

return isAdmin;
}
```

In this example, the user previously inputs the login name without validation of the size of the string. Then the check_login function is called and the variables isAdmin and usr_buff are created on the stack from high memory values to low ones as shown

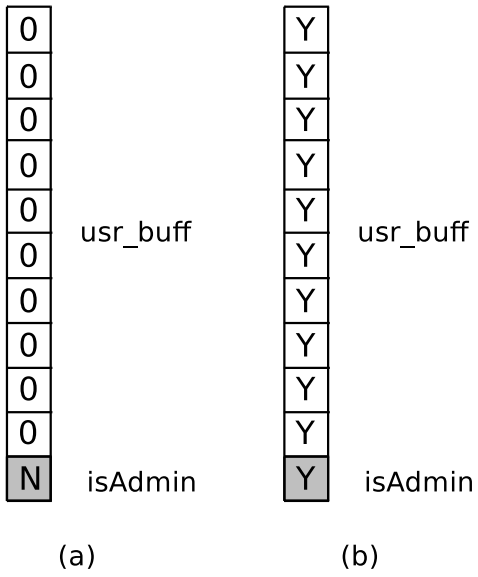

Figure 4.2: Logic Change Example

in the figure 4.2 (a). A user input of "YYYYYYYYYY" made up of 11 characters causes the usr_buff variable to overflow when the strcpy function is called, thus also changing the value of isAdmin (it also modifies the next value of the stack with a 0 that indicates the end of the string; the next value is most likely the EBP register which typically holds the memory address of the current stack frame). Figure 4.2 (b) shows the result of the input received.

4.1.1 Consequences

Buffer Overflows have evolved over the years. Part of the great importance and continuous discussion of the topic is the wide variety of consequences these errors may lead to.

The exploitation of these can result in a simple Denial of Service through a system crash (mitigation of availability). In the worst case scenario, an attacker is able to inject his own code and open a shell, enabling him to execute arbitrary commands and achieve completely control the victim's machine, have access to critical information (break the confidentiality), and change passwords or delete data (mitigate the integrity of the information).

Most serious buffer overflow exploits occur in programs written in languages which provide the programmers a lot of power and liberty in the manipulation of memory such as C or C++. Other languages, such as Java and Python, are considered safer because they provide automatic bounds checking of buffer and pointer access during run-time. The disadvantages of the latter is that they are considerably slower than C and C++. Therefore, C and C++ are the popular choice for many speed-critical applications (Linux OS, Apache and BIND for example), not to mention legacy code.

4.1.2 Buffer Overflow Taxonomy

In order to solve a problem, it is important to comprehend it completely. Taxonomies aid in the understanding of topics, by breaking it down and providing a clear classification. In [201] Zitser presents a taxonomy made up of thirteen attributes of buffer overflows, which was later modified and expanded by Kratkiewicz in [109] from which we get the following attributes (the complete list of the values of the attributes is presented in Appendix A along with examples):

1. Write/Read: This attribute poses the question "Is the buffer access an illegal write or an illegal read?"

2. Upper/Lower Bound: This attribute describes which buffer bound gets violated, the upper or the lower.

3. Data Type: Describes the type of data stored in the buffer. The values may be Character, Integer, Floating point, Wide Character, Pointer, Unsigned Integer and Unsigned Character.

4. Memory Location: The Memory Location attribute describes where the overflowed buffer resides (Stack, Heap, Data region, BSS or shared Memory).

5. Scope: Describes the difference between where the buffer is allocated and where is it overrun. It is "same" if the buffer is allocated and overrun within the same function. Other values are Inter-procedural, Global scope, inter-file/inter-procedural, and Inter-file/global.

6. Container: "Is the buffer inside of a container?" Buffers may stand alone, or may be contained in arrays, structures, or unions.

7. Pointer: Indicates whether or not the buffer access uses a pointer dereference.

8. Index Complexity: Describes the complexity of the array index, if any, of the buffer access causing the overflow.

9. Address Complexity: "How complex is the address or pointer computation, if any, of the buffer being overflowed?"

10. Length Complexity: The Length Complexity attribute describes the complexity of the length or limit passed to the C library function, if any, that overflows the buffer.

11. Alias of Buffer Address: Indicates if the buffer is accessed directly or through one or two levels of aliasing.

12. Alias of Buffer Index: This attribute indicates whether or not the index used in the buffer access is aliased.

13. Local Control Flow: Describes what kind of program control flow, if any, most immediately surrounds or affects the overflow (such as “if”, “switch” and “cond”).

14. Secondary Control Flow: The values of this attribute are the same as the types described by the Local Control Flow; the difference is the location of the buffer overflow with respect to the control flow construct. Only control flow that affects whether or not the overflow occurs is classified.

15. Loop Structure: Describes the type of loop construct, if any, in which the overflow occurs. Values include the do-while, for, and other.

16. Loop Complexity: The Loop Complexity attribute, indicates how many of the three loop components described under Loop Structure (i.e., init, test, and increment) are more complex than the following baseline:
    init: initializes to a constant.
    test: tests against a constant.
    increment: increments or decrements by one if the overflow does not occur within a loop.
    If none of the three loop components exceeds baseline complexity, the value assigned is “none.” If one of the components is more complex, the appropriate value is “one,” and so on.

17. Asynchrony: The Asynchrony attribute asks if the buffer overflow is potentially obfuscated by an asynchronous program construct. These functions are often operating system specific (i.e. threads, forks, and signal handlers).

18. Taint: This attribute describes how a buffer overflow may be influenced externally. These functions may be operating system specific, depend on command line or stdin input from a user, or on the value of environment variables, file contents, data received through a socket or service, or properties of the process environment, such as the current working directory.

19. Run-time Environment Dependence: This attribute indicates whether or not the occurrence of the overrun depends on something determined at runtime.

20. Magnitude: The Magnitude attribute indicates the size of the overflow. Values range from none, 1, 8 or 4096 bytes.

21. Continuous/Discrete: This attribute indicates whether the buffer overflow is continuous or discrete (access of consecutive elements or jumps directly out of the buffer).

22. Signed/Unsigned Mismatch: This attribute indicates if the buffer overflow is caused by a signed vs. unsigned type mismatch.

4.1.3 Attacks

Overwriting Code Pointers

Pointers corrupted by buffer overflows commonly are function pointers, longjmp buffers or pointers in the stack (such as a return address). These overflow attacks modify a pointer to some other code that gets executed later. This classic buffer overflow attack is known as "stack-smashing" [3]. An attacker overwrites the return address with the address of the attack code, and when the function returns it executes the attack code instead of the normal return point (figure 4.3). Such attack code may contain instructions to start a shell, thus gaining complete access to the victim's machine (at the level of privilege in which the process was running). An attacker can then gain root privileges through other techniques and steal user's passwords, read, delete, or modify important information, set up back-doors, or cause other damage to the victim's system.

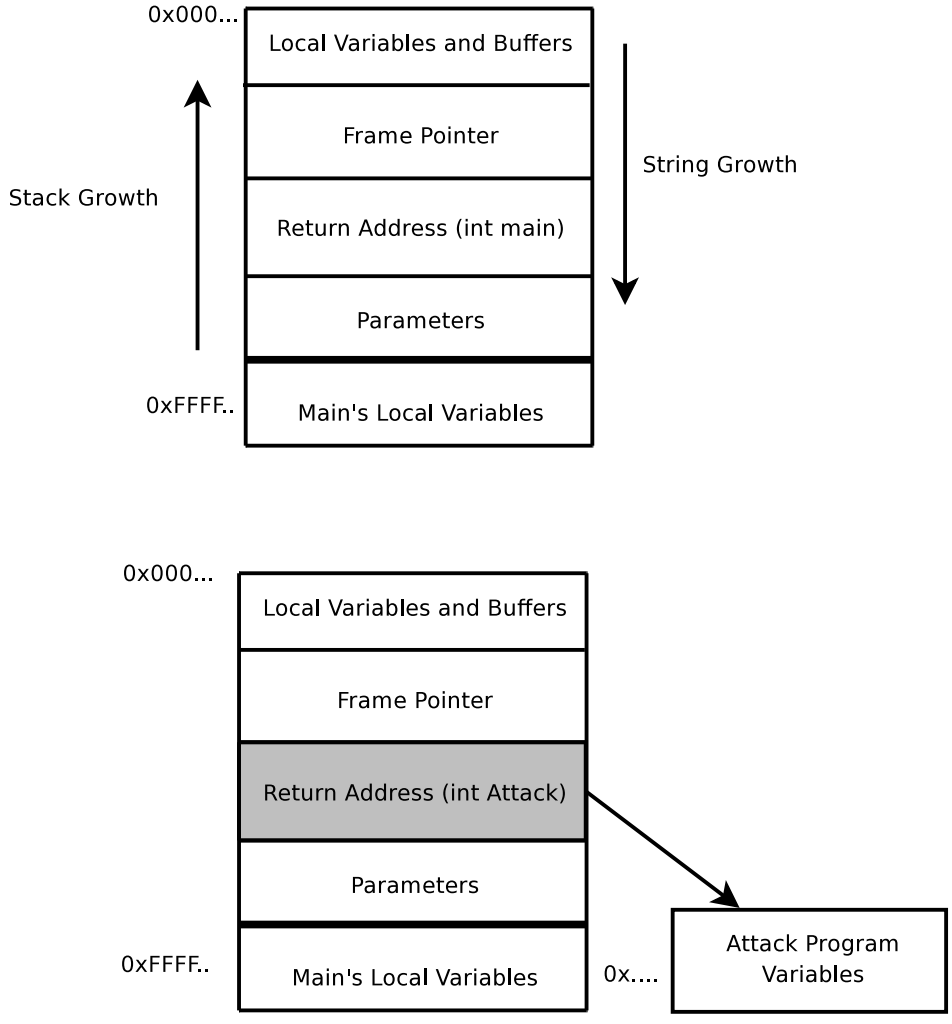

Figure 4.3: Return Address Modification

An alternative to overwriting the return address of the function, an attacker can overwrite the saved frame pointer (stored right before the return). A frame pointer is used to keep track of the stack frames corresponding to the different function calls, and it points to the first local variable in the current stack frame. This way, an attacker would construct an equivalent malicious stack frame somewhere in memory with the return address pointing to the attack code.
It is also possible to change the execution flow of a program by overwriting a function pointer. To carry out this attack, an attacker needs to find a function pointer that is stored adjacent to a buffer. This buffer can be located in any of the four regions of process memory, the stack, the heap, the bss region or the data region. The attacker overflows the buffer and overwrites the function pointer, it ends up jumping to the attacker's desired location. This type of attack was used against the superprobe program for Linux [47].

Logic-based Attacks

It is possible to for the attacker to change the logic of the program by overwriting a program variable. Figure Stack Variables is an example of this type of attack. Also, the Morris Worm used a logic-based buffer overflow attack that corrupted the name of a file that would then get executed. These attacks can involve variables on the stack, the heap, the data or the bss regions of memory. Even though these attacks are not as common as some of the others, they are just as dangerous.

Code Injection

An attacker may inject code via the string that is used to overflow a buffer. If this is the case, in stack-based attacks, the attacker usually overwrites the return address with a pointer back to the buffer. This attack may be used when the unchecked, overflowable buffer is not large enough to have bound checking, but the bound check is done incorrectly. In other words, it may be possible to overflow the buffer by a few bytes, but not enough to insert the attack code. The two used buffers may be located in different regions of memory. For example, the buffer containing the attack code may be on the stack, but the overflowed buffer and the code pointer may be stored on the heap. If the buffer containing the code is on the stack, it is possible to prevent this type of an attack by making the stack non-executable, as was done with Open Wall'a Linux kernel patch[38]. Finally, an attacker may be able to use an environment variable, it may be possible to write the attack code into the environment variable, and then overflow some buffer on the stack, overwriting a function's return address with the address of the environment variable. Alternatively, it may also be possible for an attacker to overwrite some function pointer and point to the environment variable.

Attacker Use Existing Code

An attacker can circumvent a non-executable stack defense by overwriting the return address of the vulnerable function with the address of another function in the program,

or even a shared library function. Such attacks do not involve code stored in the overflow string. The attacks that cause a jump into a standard libc function are commonly known as return-into-libc attacks. The basic idea of the classical return-into-libc attack is to overwrite a function's return address with the address of a standard C library routine, such as system(). By also cleverly placing the arguments to this C routine on the stack, it is possible to execute function calls such as system("\bin\sh"), i.e. start a shell. When the attackers do not know the exact location of the return address on the stack, they overwrite the return address by repeating the desired address many times in the buffer until it hits the desired location.

Heap-based Buffer Overflows

Heap-based buffer overflows are becoming more common because of non-executable stacks. Although it is possible to have non-executable heaps, heaps are more likely to be executable than stacks. Other major reason to unfavor stack-based buffer overflows are dynamic tools such as StackGuard, StackShield, and Propolice. Unfortunately, a similar method for protecting heaps is much more complex and does not yet exist. A fairly recent attack involving a heap-based buffer overflow was the infamous Code Red Worm of 2001[19]. Heap based attacks are usually more complex and are more difficult to mount than stack-based attacks mainly because of the more complex operation of the heap.

4.2 Cross-Site Scripting (XSS)

Due to the exponential growth in website emergence, in 2006 cross-site surpassed buffer overflows as the leading reported vulnerability. The cross-site scripting vulnerability has been found in websites such as fbi.gov, yahoo.com, ebay.com and many other popular and important websites. Little attention is given to XSS attacks by some, because they either don't know much about them or they do not see them as a threat. An XSS vulnerability can result in a very powerful attack, which can be exploited by a skilled attacker or even a novice. The acronym XSS is used instead of CSS, since the latter could cause confusion with the "Cascading Style Sheets".

This attack occurs when web pages such as Forums, Email, Web-Stores or other trusted by the user, dynamically generate contents without proper sanitation or input validation. This allows attackers to embed malicious code into the page (could be JavaScript, VBScript, ActiveX, HTML or similar) and then execute the script on the client side. Web page structure is very flexible, and it allows different types of content to be presented. Therefore, once the web browser encounters the [script] tags, it triggers the corresponding interpreter and runs it [73].

4.2.1 Consequences

An important factor to mention with XSS is that the direct impact of the attack is suffered more often by the clients who execute scripts in their web browsers, rather than

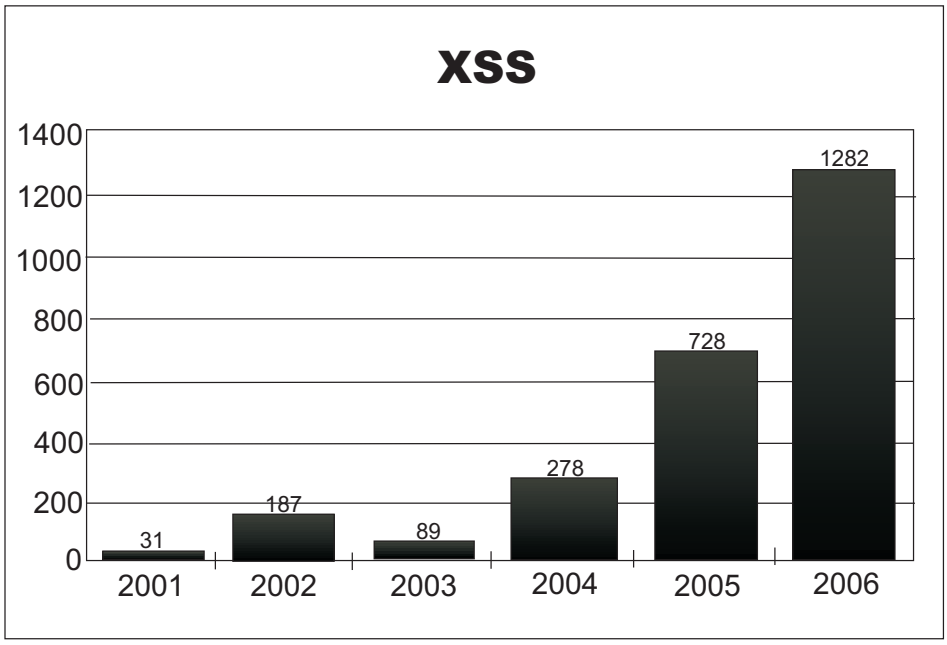

Figure 4.4: Cros-Site Scripting vulnerabilities reported by year (CWE)

the vendors or providers of the trusted service (these get affected in their reputation). An XSS vulnerability may result in the mitigation of one or several of the following Security Services:

- Confidentiality: This involves the disclosure of information stored such as credentials, and monitoring of your actions (digital stalking).
- Access control: It may be possible to run arbitrary code on a victim's computer to enable an attacker to impersonate the victim and jump access control.
- Integrity: Creation of phony user interface to provoke misinformation from a trusted site or modification of the information users post.
- Availability: This is done with code that results in a Denial of service.

4.2.2 Attacks

Reflective Attacks

This is one of the most popular presentations of XSS. Reflective attacks occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the

attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other activities. In 2000 Microsoft was forced to shut down Hotmail since a script that intercepted Hotmail authentication cookies and took over users' accounts was detected. It is important to point out that for a reflected XSS attack to work, it is the victim who must submit the attack to the server. Here is an example of how this might take place:

Stealing the Cookie Example.
So you visit MyFavoriteShoppingSite.com, which uses cookies to remember credentials, in order to maintain state when one moves from page to page on the site without re-entering the password. These cookies are files stored locally in the client side, which contain secrets like a SessionID or nonce to achieve their goal. This way, anybody else who is able to get my cookie and provide it to the site, will be believed to be me.

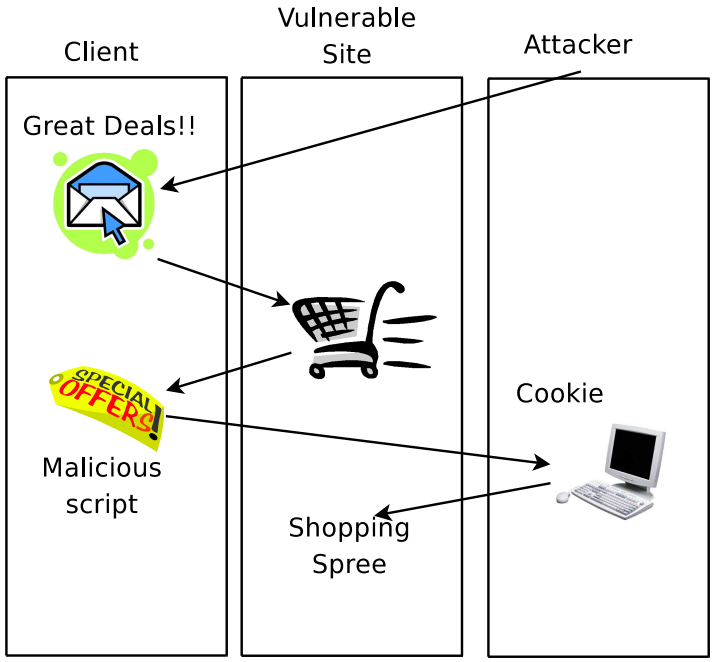

Figure 4.5: Stealing the Cookie

Then the attacker finds that there is an XSS vulnerability in the web application software that the site uses, he sends the victim an email with the following HTML:

```
<A HREF="http://myfavoriteshoppingsite.com/greatdealsofthemonth/?
tw=<script>document.location.replace('http//evilsite.com/ph4r/
steal.cgi?'+document.cookie);</script>">Check Out Our Labor Day
Specials!</a>
```

The user would click the link and they would be lead to the offers, but at the same time be directed to the specially crafted URL, where the attacker obtains the user's

cookie. Using a cookie editor the attacker copies the cookie, and impersonating the user buys all he can and having it sent to him. The end of the month arrives, and with it the $6,000 credit card bill; and the items bought already delivered.

This is just one of the many scenarios that are constantly presented.

Inline Frame / Script Include

It is also possible for a web page with malicious code, to modify a frame in another open window containing a trusted site.

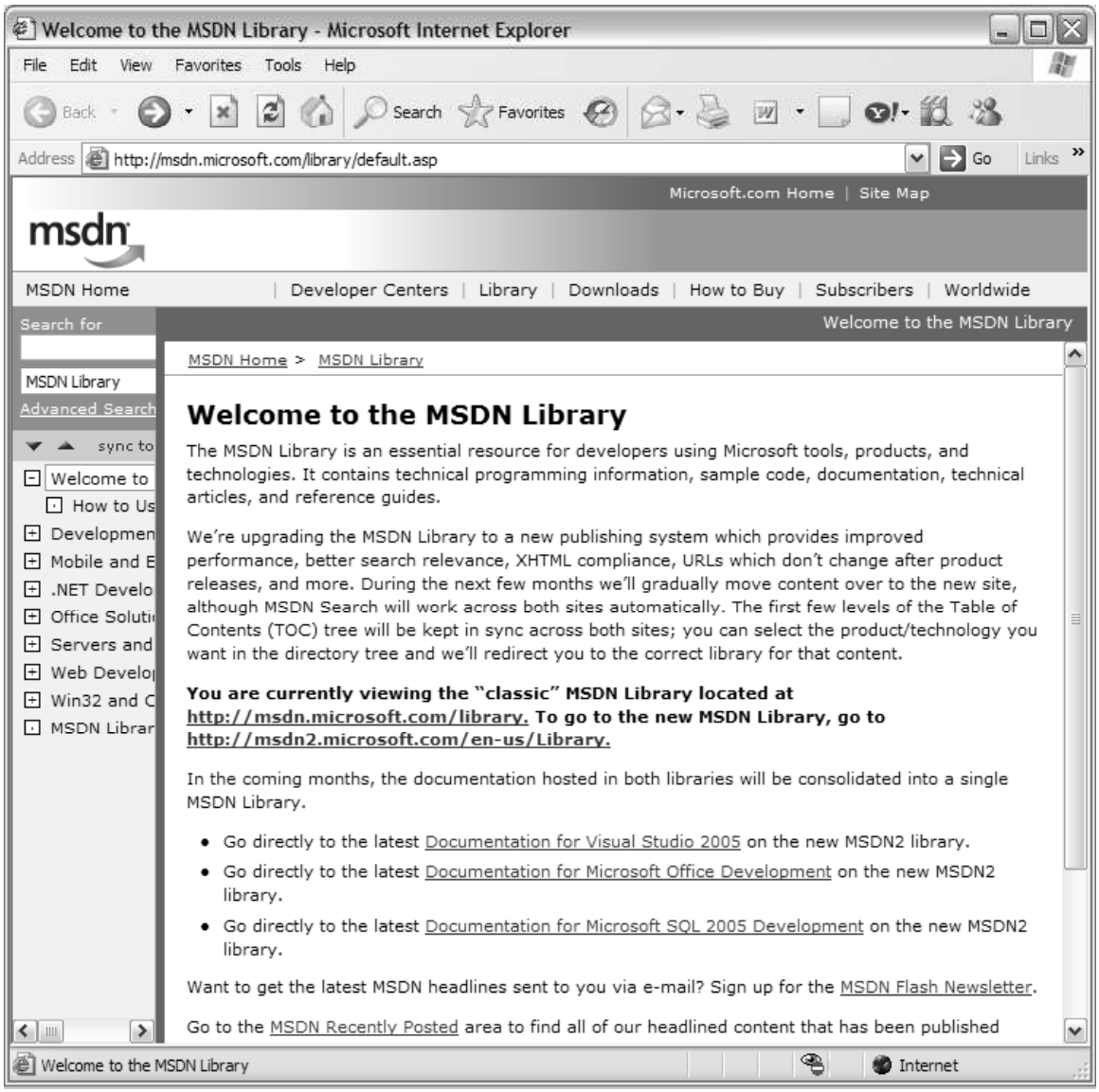

Figure 4.6: MSN

The attacker can take advantage of this situation in several ways. Javascript forwarded cookies to another site, for example, the following Phishing Scenario:

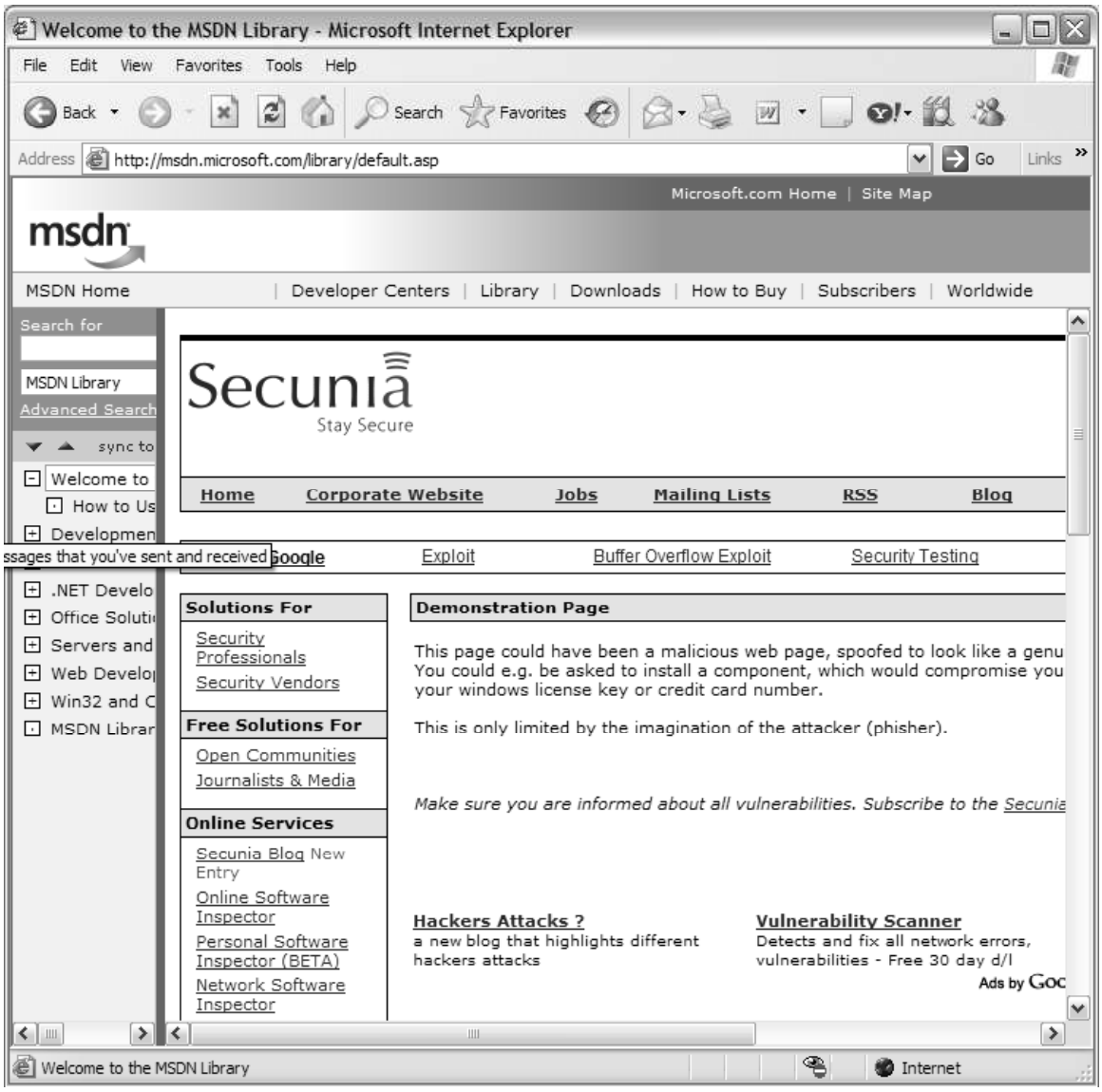

Figure 4.7: MSN

1. Victim Logs into a web site.
2. The Attacker has set up "mines" using an XSS vulnerability.
3. The victim stumbles upon an XSS mine.
4. The victim gets a message saying that his session has expired and he needs to authenticate again.
5. The victim's username and password are sent to the attacker.

Stored Attacks

Stored attacks are those where the injected code is permanently stored on the target servers in a database, message forum, visitor log, and so forth. The data stored is later

read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

Other Media

The XSS problem is not restricted only to websites. There are many types of media files that contain URLs, such as MP3s, video files, PDFs and Flash animations. The programs used to manage this type of content may interpret the embedded URL data directly or transfer the HTML to a Web Browser. Once this happens, the same type of problems are exploited here (see figure 4.8).

Figure 4.8: XSS in Quicktime

4.2.3 Conclusion

XSS attacks are discovered on a daily basis in practically every form of web software there is. If people don't educate themselves about XSS attacks attackers are going to continue to exploit XSS vulnerabilities, and this could lead to some very dangerous and powerful attacks. Combining XSS with social engineering or phishing what makes these extremely dangerous. The variety of attacks based on XSS is almost limitless, but they all include transmitting private data like cookies or other session information

to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under hiding behind a vulnerable site.

4.3 SQL Injection

Structured Query Language (SQL) is a textual language used to interact with relational databases[193]. The typical unit of execution of SQL is the 'query', which is a collection of statements that typically return a single 'result set'. SQL statements can modify the structure of databases (using Data Definition Language statements, or 'DDL') and manipulate the contents of databases (using Data Manipulation Language statements, or 'DML'). There are many implementations SQL and database applications are very common.

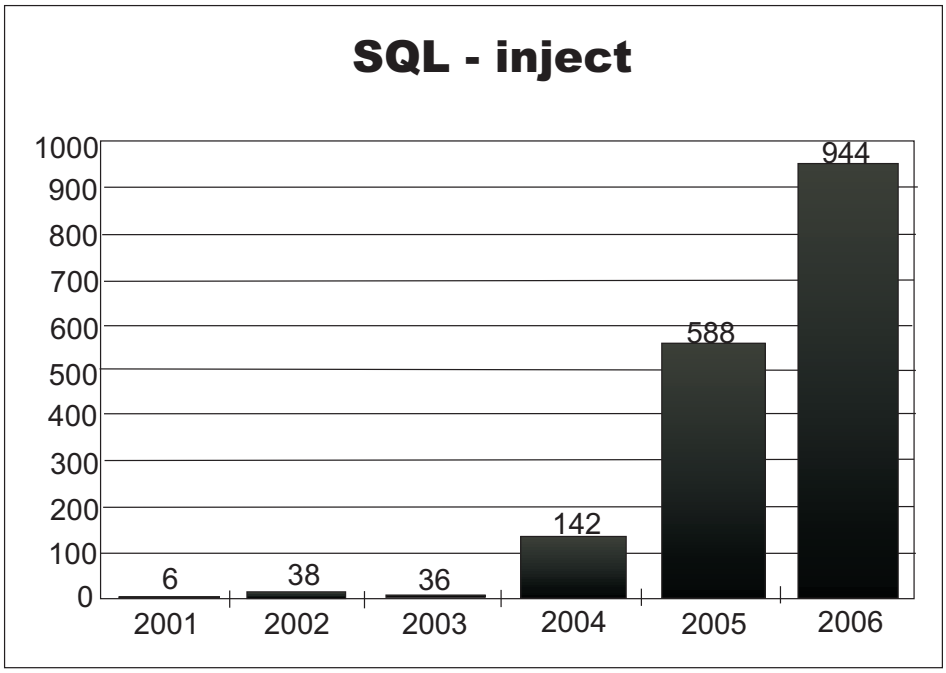

Figure 4.9: SQL Injection vulnerabilities by year (CWE)

SQL Injection is a technique which consists in passing SQL code into an application in a way not intended by the developer, to either gain unauthorized access to a database or to retrieve information directly from the database. The technique takes advantage of bugs or non-validated inputs to "inject" the SQL commands for the execution by the database. Attackers take advantage of incorrectly filtered strings, and can therefore embed the code inside these parameters. The principles behind a SQL injection are simple and these types of attacks are easy to execute and master. Web applications with data access are the most commonly affected.

SQL Injection can be present in platforms such as C, C++, Java, and .NET.

4.3.1 Consequences

SQL Injection can result in the violation of one or several of the following Security Services:

- Confidentiality: Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities.
- Authentication: If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password. You want to make sure people don't get access to what they are not supposed to. It could be to restrict them to only their account, or prevent them from getting Company secrets.

  You want to make sure people don't get access to what they are not supposed to. It could be to restrict them to only their account, or prevent them from getting Company secrets.

  If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL injection vulnerability.
- Integrity: Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL injection attack. Assign the right permissions to each user, for example unauthorized access might only allow Read-only access but no modification of the data (like catalogs). You wouldn't want to find yourself selling illegal or politically incorrect items on an eCommerce site. It could be very costly to have an expensive item suddenly be extremely discounted.

  Corvettes for $ 100 in our going out of business sale!!!
- Availability: It can generate a DoS with the elimination of data and shutting down the data server.

Some secondary consecuences can be the control over the database host, and other machines. Once inside the network, there is no stopping to what the attacker can do. He can get the privileges of any other user through the other users passwords and this way get unlimited access. It could be the case where the database itself is not the main target but rather the means of getting control of the rest of the network.

4.3.2 Attacks

Access through Login Page

This type of access is the most common and the easiest one to bypass since is usually the first thing we see, but it does not necessarily have to be through the username and

password fields. This can practically be done through any field where one in allowed to input a string of information. There are several commands that may be run in order to get access hand learn about the structure of the database. The following are examples of code that is often maliciously executed:

Using 'or' condition.

Let's think of the login case, where we have the username and password fields to be validated for access to a specific account. With this in mind, we would probably have a validation string that looks as the following:

```
"select * from store.guest.clients where username = ' " + username + " '
  and  password = ' " + password + " ' ";
```

What an attacker could do, is write anything for the username field, for example: John, and for the password field he would write: ' or 1=1 –
This combination would map into the validation string the following way:

```
select * from clients where username='John' and password= '' or 1=1 --'
```

With the ' , the attacker has closed the string expected for password, and has added an or condition that is always true. The final - - are being used to ignore the rest of the code that could be in the command (other validations maybe). The always true 1=1 allows access to the attacker positioning him on the first record of the user table and giving him access to the whole account of such user.

Figure 4.10: Bypassing the Login

Using 'having' clause
The last example demonstrated how the attacker could get access to the account. So far we have an undetectable passive attack. But what stops the attacker from trying something like this:

```
John'; DROP TABLE clients; --
```

In order to do this he would need to know the name of the tables. Still, the attacker could make an educated guess (which is why some people recommend Security through Obscurity to use table and field names harder to guess). Developers are not too fond of this Idea, since they probably already have dozens or even hundreds of things to remember when coding, and they want to make the project as clear and agile as possible. However, the hacker could learn about the structure of the database, by entering strings rejected by the SQL parser, such as:

```
' having 1=1--
```

This command will cause an error because the keyword "having" needs the "group by" operator.

```
Error:
[Microsoft][ODBC SQL Server Driver][SQL Server]
Column 'store.guest.clients.id' is invalid  in
the select list because it is not contained
in an aggregate function and there is no GROUP
BY clause.
```

This error provides important information to the attacker:
Name of the database store, Name of the main user guest, Name of the table clients, and name of the field of the primary identifier id.

Using multiple queries.

Several Database Systems delimit queries with a semi-colon. The use of a semi-colon allows multiple queries to be submitted at once and be executed sequentially, for example:

```
Username: ' or 1=1;
drop table users;
drop table debits;
-- Password: password
```

Another very dangerous alternative is to shut down the SQL service. This is one of the most powerful commands done, and the SQL Server's function to do this is:

```
SHUTDOWN WITH NOWAIT
```

Where the query would look like this:

```
Username: '; shutdown with nowait; --Password: John
```

This would create a very effective and interesting form of Denial of Service Attack. This attack is difficult to detect since there probably isn't any high traffic detected, hence making it more difficult to find the source of the problem for a network administrator.

Using Remote Execution of extended stored procedures

There is a whole set of hazardous stored procedures [193] that come by default in many of the Database systems. For example, with the remote execution of:

```
exec master.xp\_cmdshell 'net stop sqlserver'
```

We can also turn down the service and cause a Denial of Service Attack as described before. Of course, you would the necessary privileges to run such procedures.

Access by manipulating the query string in URL

Basically, the same commands can be executed through the query string in URL.

The URL may look like this:

```
www.sqlproduct.com/sqlproducts.asp?p\_id=7
```

In order to know the field name of products table attacker can write:

```
http://sqlproduct/sqlproducts.asp?p\_id=0\%20having\%201=1
```

Where the %20 represents the space character.

Using the obtained field products.prodName, we now call up the following URL in the browser:

```
http://localhost/products.asp?productId=0;
insert\%20into\%20products(prodName)\%20values(left(@@version,50))
```

This performs an INSERT query on the products table, adding the first 50 characters of the SQL Server's version in the products table as its last record. The attacker could now get the version of SQL server by writing:

```
http://localhost/products.asp?productId=
(select\%20max(id)\%20from\%20products)
```

Now he can perform a specific set of commands for the particular version of the SQL server.

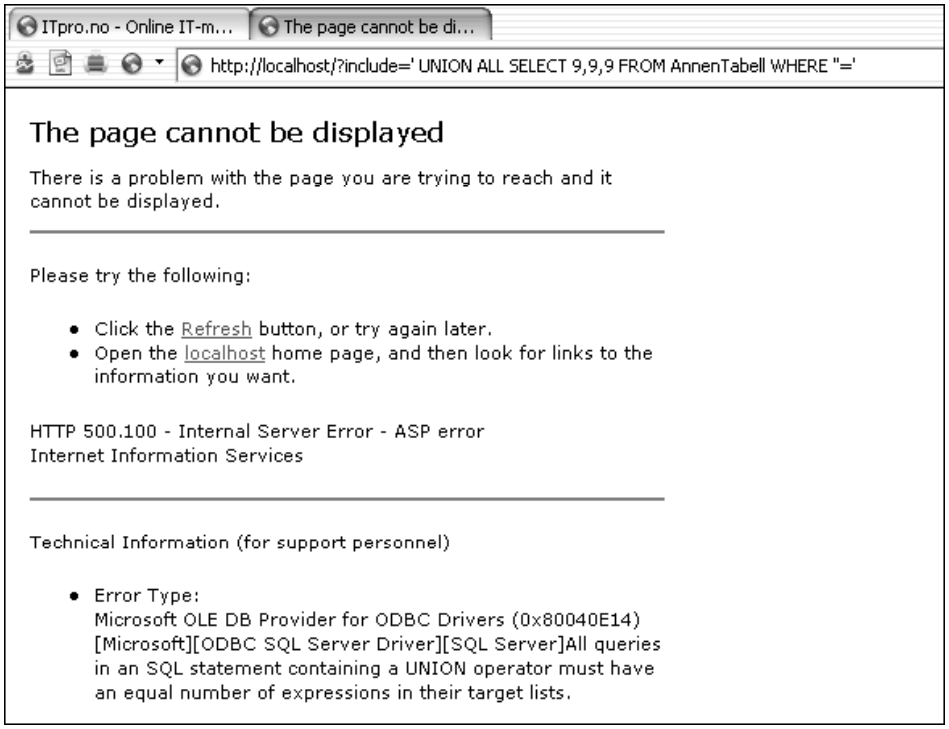

Figure 4.11: URL SQL Injection

4.3.3 Conclusion

SQL injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind. Essentially, the attack is accomplished by placing a meta character into data input to then place SQL commands in the control plane, which did not exist there before. This flaw depends on the fact that SQL makes no real distinction between the control and data planes. Usually data destruction is not very common since there is low economic motivation for this.

If successful, SQL Injection attacks can give an attacker access to backend database contents, the ability to remotely execute system commands, or in some circumstances the means to take control of the Windows server hosting the database. Dynamically generating queries that include user input can lead to SQL injection attacks. An attacker can insert SQL commands or modifiers in the user input that can cause the query to behave in an unsafe manner. Constructing a dynamic SQL statement with user input may allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

SQL Injection is one of the most important problems in web application security because it can, with a small amount of access, put at risk the knowledge and control of multiple servers and consequently very important information of the company and

the customers.

4.4 Race Conditions

Race conditions are vulnerabilities that originate when concurrent processes or threads interfere with each other. This interference is usually provoked from access to a shared resource (variables, files, memory, devices, etc...) without the implementation of the proper mutual exclusion protection mechanisms. Many developers pay little attention to this issue, since they consider highly unlikely for the situation to emerge and do not consider the consequences to be severe. Attackers take advantage of the developer's assumptions and race with their programs to try to invalidate a particular resource, hence the name of the vulnerability.

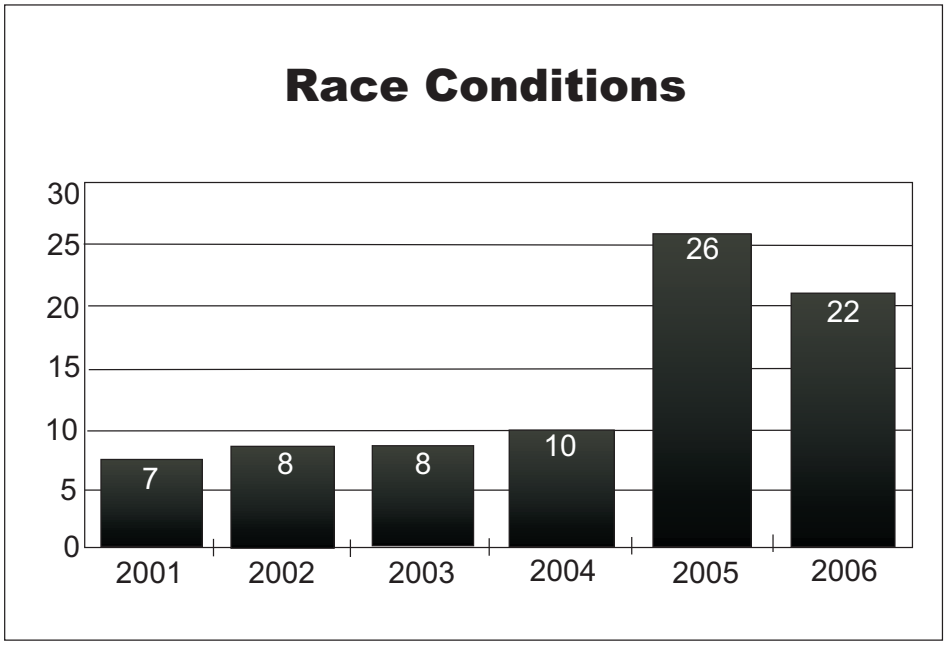

Figure 4.12: Race Conditions by Year (CWE)

Distributed computing is achieved through efficient control of time and state. Computers execute a great amount of instructions very quickly, and they won't often complete their execution before another process or thread gets assigned some time. In fact, in multi-core, multi-CPU, or distributed systems there can be two events taking place at the same time, thus making race conditions easily available in such systems [89]. The difference between the programmer's mental model and the reality of the execution gives place to unexpected interactions between threads, processes, time, and information[44].

Race condition are tied to the timing of events within a piece of software. They are usually associated with synchronization errors that provide a window of opportunity

during which one process can interfere with another, possibly introducing a security vulnerability.

Consider the situation illustrated in figure 4.13:

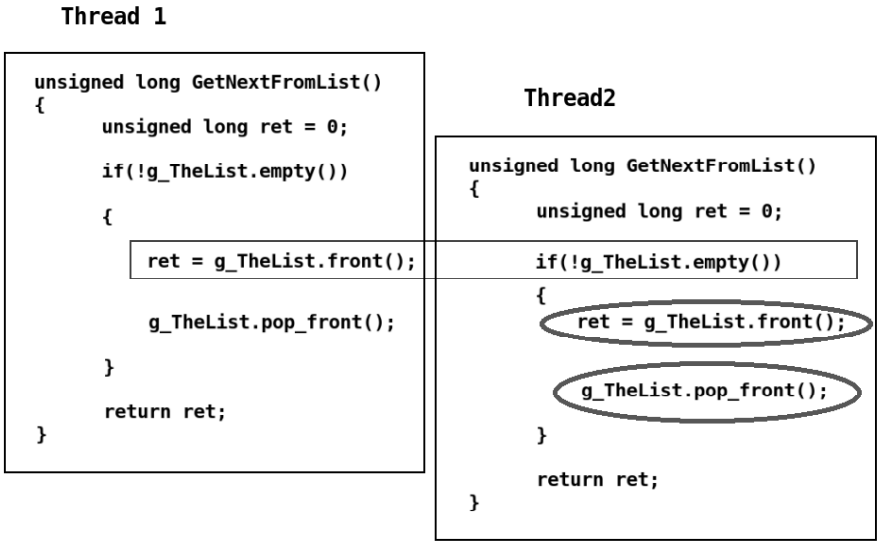

Figure 4.13: Race Condition Example

Let's consider we only have one element in our list. Trouble arrives as one thread passes the check as to whether the list is empty just before another calls pop_front on the remaining element. The unexpected situation will certainly bring unexpected results. As you can see in the example, when more than one thread or process access a same resource in an uncontrolled manner, there are diverse conflicts. The resource could be shared memory, global variables, the file system (for example, by multiple web applications that manipulate data in a shared directory), other data stores like the Windows registry, or even a database.

Now consider the example in figure 4.14. The image describes a specific scenario in a bank application, where there are different tasks being done at the same time over the stored information. We have a process A calculating the total savings of all the clients in the bank and at the same time we have a process B in charge of a transfer between two clients. However, process B starts executing when process A is in the middle of its operation. Process A gets to Paul and eventually to the end and counts $300 less than what there really is, therefore mitigating the integrity of the information.

Race conditions are well studied threats and there exist different ways to exploit them. They are not restricted to a particular language or platform. In fact, even though some high-level languages are not vulnerable to some kinds of race conditions due to their lack support for threads or forked processes, their slower performance often makes them more susceptible to other types.

In the following section are described some of the most important types of race condition, as well as the impact each of these have on the security services.

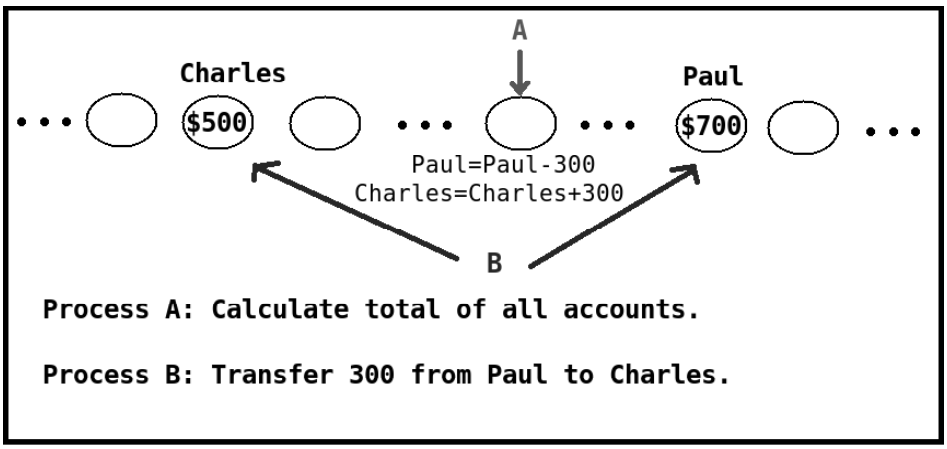

Figure 4.14: Conflicting Transactions

4.4.1 Attacks

Signal Handler

Signal handling race conditions started to be exploited relatively recently [200]. These issues occur when non-reentrant functions, or state-sensitive actions occur in the signal handler, which is expected to receive calls one at a time. Many applications implement signal handlers, and sometimes they just map the same handler to more than one signal. When the attacker sends a rapid-fire pair of signals the program becomes multithreaded. If it is complicated to write multithreaded code, imagine the problems that arrive when you're not expecting concurrency problems.

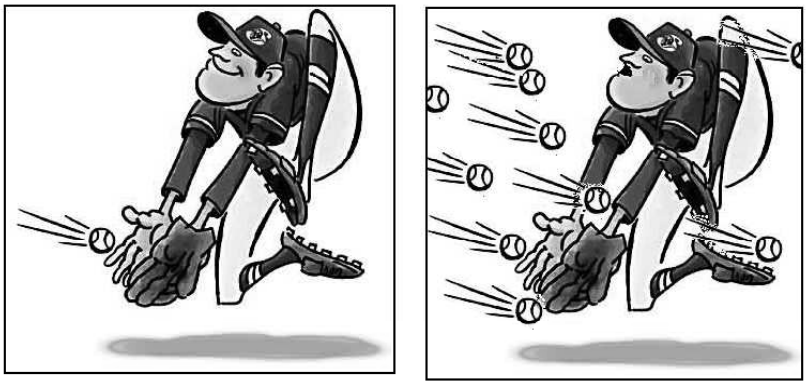

Figure 4.15: Nonreentrant to Threaded (image edited from [135])

Example:

```
#include <signal.h>
#include <syslog.h>
```

```
#include <string.h>
#include <stdlib.h>

void *global1, *global2;
char *what;
void sh(int dummy) {
  syslog(LOG_NOTICE,"\%s\n",what);
  free(global2);
  free(global1);
  sleep(10);
  exit(0);
}

int main(int argc,char* argv[]) {
  what=argv[1];
  global1=strdup(argv[2]);
  global2=malloc(340);
  signal(SIGHUP,sh);
  signal(SIGTERM,sh);
  sleep(10);
  exit(0);
}
```

In this case, a massive amount of calls to the signal handling function results in a multiple call of free. Race conditions occur frequently in signal handlers, since they are asynchronous actions. These race conditions may have any number of root-causes and symptoms.

Common Consequences:

- Authorization: With the combination of other techniques, it may be possible to execute arbitrary code.
- Integrity: Signal race conditions often result in data corruption.

Time-of-check Time-of-use race condition

This type of exploit emerges when a program checks for a particular property of an object, and later executes instructions assuming that the property still holds when in fact it does not. TOCTOU race conditions are usually presented when a program needs to have access to files, checking first if the file already exists, and if not, then creating it. The attack starts when the attacker figures out how you name the files and starts creating links back to something important. The application opens a link that's really the file chosen by the attacker, provoking corruption of important files or achieving escalation of privilege through other techniques.

Common Consequences:

- Access control: The attacker can gain access to otherwise unauthorized resources.
- Authorization: race conditions such as this kind may be employed to gain read or write access to resources which are not normally readable or writable by the user in question.
- Integrity: The resource in question, or other resources (through the corrupted one), may be changed in undesirable ways by a malicious user.
- Accountability: The concurrency of the activities might prevent logging to occur.
- Non-repudiation: In some cases it may be possible to delete files a malicious user might not otherwise have access to, such as log files.

Consider a port scan detector application which writes a log in "/tmp/log" of the activity perceived. Such program would need root permission in order to be able to read from the network device. Let's analyze the following code:

```
if(!access(filename,W_OK)){
   monitorNet();
...
   f = fopen(filename,"w+");
   /*write to file*/
  }
```

This code segment first checks if the file is accessible by the user, then opens it for writing, monitors the network, and finally writes to it. The first access check is passed successfully, but between the time that the first access check is made and the time that the file descriptor f is created, an attacker can delete the file "/tmp/log" and create a hard link named "/tmp/log" referencing a configuration file such as "/etc/shadow". Now, when the file "/tmp/log" is opened for writing, the shadow file containing the passwords will be opened instead and the attacker will be able to write arbitrary data to it. The program uses its root privileges to write the file even though the attacker does not count with such permissions. TOCTOU is not restricted to programs with super user permission; as long as the program counts with any privilege above those of the attacker, then it is probably of interest to him.

It is important to point out that TOCTOU vulnerabilities not always involve symbolic links, and not every symbolic link issue is a TOCTOU problem. Another important aspect to consider is that symbolic links are not restricted to UNIX systems, but are implemented in Windows systems as well. Windows has a large number of different named objects-files, pipes, shared memory sections, desktops, and other vulnerable resources that could be exploited.

Context Switching Race Condition

These vulnerabilities usually present themselves when a product performs a series of non-atomic actions to switch between contexts with different security schemes or levels, but a race condition allows an attacker to alter the product's behavior during the transition. This is more often found in web browsers, in which the attacker can perform certain actions while the browser is transitioning from a trusted to an untrusted domain, or the opposite, and the browser performs the actions on one domain using the trust level and resources of the other domain.

An Example of such case is the one registered in the Common Vulnerabilities and Exposures database under the following ID:

CVE-2004-2260 - Browser updates address bar as soon as the user clicks on a link, instead of updating it when the page has loaded, allowing spoofing by redirecting to another page using onUnload method.

4.4.2 Conclusion

Race conditions represent a small but constant percentage of the vulnerabilities reported. However, we must remember that it only takes one vulnerability in your software to impact the security in a severe way, and race conditions have the potential of creating such malicious consequences.

4.5 Design Vulnerabilities

In the previous sections we described the Buffer Overflows, SQL Injection, XSS, and Race Condition Vulnerabilities. They are vulnerabilities that originate during the implementation of systems. However, vulnerabilities may also come from Software design. A design vulnerability is a problem that arises from an error or oversight in the software's design [58]. In this case, software isn't secure because it does what it was designed to do; only it was designed to do the wrong thing. These types of flaws often originate from assumptions made about the environment in which a program will run. Design flaws are also referred to as high-level vulnerabilities.

The design specifies the architecture and the interaction between components, and reflects the requirements previously established. This includes the system's security mechanisms and how they work. For example in multiuser systems, the authentication and authorization of users should be reflected in the design [199]. According to Michael Howard [85], 50% of the problems discovered during Microsoft's "Security Push" of 2002 were in the design-level.

Design vulnerabilities are hard to detect, since they can range from restraining from doing something, or incorrect way of doing it. These can be access control, logging, timing, encryption, and many more.

The Y2K bug, for example, could be considered a design flaw. The decisions behind the two-digit date format were of efficiency in memory when this was scarce.

The use the abbreviated form of the year does not represent any problem for humans, but it is not this way for computers. The Y2K last minute correction effort is estimated in the billions [15], should we start thinking about Y10K?.

The TELNET protocol allows users to connect to a remote machine as though it were connected to a local terminal. From a design perspective, TELNET has a vulnerability in that it uses unencrypted communication. This enables attackers to monitor and hijack TELNET sessions. If an administrator connects via TELNET and enters a username and password to log in, a sniffer could monitor and obtain this information.

If design does not include security considerations, it is very difficult, or maybe impossible, to "add on" the security later in the development.

4.6 Deployment Vulnerabilities

Deployment consists in the installation and configuration of systems. Usually, the person responsible for this activity was not involved in the development process and requires guidance to perform his task [199]. A deployment error usually comes from assumptions made about how the deployment will occur (for example assumptions with configuration files and high privileges during installation).

User Supplied Configuration File.

A setuid utility program accepts command-line arguments. One of these arguments allows a user to supply the path to a configuration file. The configuration file allows shell commands to be inserted. This way, when the utility starts up, it runs the given commands. The utility program may not have root access, but may belong to a group or user context that is more privileged than that of the attacker.

Insecure Defaults

Insecure defaults are predefined options that create a risk in a deployed application. This problem occurs when a software vendor attempts to make the deployment as simple as possible; usability ends up conflicting with security. Requiring a configuration change out of the box is often forgotten, leaving the window open for attackers.

Commercial Wireless access point devices are a known case of insecure defaults. They come preconfigured without security. The companies try to facilitate the process to the users, and they achieve it, but end up exposing their communication to anyone within a few hundred yards.

Default Site Installations

Some Web servers include a variety of predefined sites and applications as part of a default installation. The goal is to provide guidance and reference for configuring the server and developing modules. The problem is that these sample sites are unnecessary

services and insecure defaults. For example, ColdFusion's Web-scripting technologies installed some applications by default that allowed files to be uploaded and executed on the system.

4.7 Authentication and Password Vulnerabilities

People dislike passwords, specially if they're asked to choose them with size and character restrictions, and told to use a different one for each account they might have. There is a great dilemma in balancing between a hard enough password an attacker can't guess and easy enough a user would not write it down on a post-it on the computer.

Authentication systems can fail in several ways. We want to prevent attackers from being able to log into an account that isn't theirs and they have no right to use. This may be achieved without the password, for example, in a replay attack, someone might be able to thwart the password protocol and log in just by sending a duplicate of some encrypted data.

Another problem presented with frequency is leaving around default accounts with default passwords. This is common in public places such as libraries, where computers are shared by a wide range of users. The particularity of this problem is that it makes the average joe an attacker. The next person in the computer is motivated by the effortlessness of the attack combined with the curiosity the opportunity brings.

Attackers can also get passwords, and other critical information, through the use of key logging software or hardware, or otherwise eavesdrop on password entry; for example with cameras. In 2005, thieves masquerading as cleaning staff managed to break into Sumitomo Mitsui Bank's branch in London and installed hardware key loggers and attempted to transfer $440 million to accounts in other countries [171]. This types of loggers are difficult to detect since there's no process running in the background, just an artifact connected between your keyboard and computer. The criminals were able to capture URL's accounts, passwords, and other sensitive information but were captured before they were able to cause any real harm. It is interesting to point out that the bank's strategy against future attempts of this type, was attaching with superglue the keyboards to the PC's.

Another very important aspect in password security is password storage. To avoid a server-side capture of a password, it is a common practice not to store passwords directly, either on a server or in a database; why should a user trust a system admin? One-way hash of the password is the common solution to the problem. The hash is used to validate that the user knows the password by comparing the value stored to the one provided at the input after it has gone through the irreversible process.

Brute-force attacks are usually the last choice of attackers, where they simply try to log in as the user numerous times, each time with a different guess. If users are allowed to have weak passwords, then this strategy is worthwhile.

Paris Hilton Hacked? [126]. In 2005, it was reported that someone got access to Paris Hilton's T-Mobile Sidekick cell phone, making public it's contents in the Internet (including contact information for a number of celebrities). The fact is the phone had

little to do with the breach. The Sidekick architecture stores copies of data so it can be accessed by the subscriber over the Internet and via phone. The attacker managed to get her username and claimed be the legitimate client and to have forgotten the password to the account. The password reset personal question established by the heiress was "What is the name of your favorite pet?". Her dog Tinkerbell is somewhat of a public figure, the attacker knew this, got access to the account, and the rest is history.

4.8 Encryption Vulnerabilities

Cryptography is an essential technique to secure certain aspects of systems. Encryption algorithms are commonly used to provide confidentiality of transactions and as methods to authenticate users. However, cryptography is a complex technology and there are ways it can make systems vulnerable.

Storing Sensitive Data Unnecessarily

Often, a systems are designed to maintain sensitive data without any real cause, typically because of a misunderstanding of the system requirements. Passwords are one of the most typical cases of storing data unnecessarily, but not the only case. Application often have designs that fail to classify sensitive information or just store it for no reason.

Lack of Necessary Encryption

Generally, a system doesn't provide adequate confidentiality if it's designed to transfer clear-text information across untrusted environments (TELNET is the case). Communication containing sensitive information should be encrypted when it travels over public networks. Sensitive information should be encrypted as it's stored in a database or on disk. In some situations, such as password storage, hashed values of sensitive data can be stored in place of the actual sensitive data.

Insufficient Encryption

It's also possible to use encryption that isn't strong enough to provide the level of security required. If the data is valuable enough, attackers will be willing to wait exhaustive or timely attacks. For example, 56-bit single Digital Encryption Standard (DES) encryption is a bad choice in the current era of inexpensive multigigahertz computers. The security of the information should be established in proportion to the value of the information. It is often the case that information is needed to be confidential for a specific period of time, and afterwards loose its value and be of no use to attackers. For the last case one may settle for an encryption that can be broken in days or weeks, as long as is greater than the period of value.

It's also important to remember that encryption implementations do age over time. Computers get faster, holes are found in the encryption algorithms. Key sizes eventually become inadequate for the data they protect.

Security by Obscurity

Security by obscurity (or obfuscation) has earned a bad reputation in the past several years, but it has always been a tempting strategy for programmers. Encryption algorithms undergo exhaustive testing by large audiences, therefore are being reinforced. With obscurity, if the attacker spends enough effort and time he will most likely break the code. Obfuscation it's an insufficient technique for protecting data from attackers, however, it is a commonly employed strategy to deter casual snoopers and slow down dedicated attackers.

Chapter 5

Security in the Software Development Process

Security is not really contemplated during most softwares' development lifecycles, but is rather considered an afterthought, where security verification and testing efforts are done until the software has been developed. We have already seen in previous chapters how the inception of vulnerabilities may originate throughout the design, implementation, and deployment phases, so the natural step is to implement security measures at all these stages. Security is definitely not an add-on one can easily integrate after the software has been developed. It has been demonstrated that the earlier a defect is unveiled, the cheaper it is to fix. Chris Wysopal said in [199]:

> "A full lifecycle approach is the only way to achieve secure software."

In [52], Noopur Davis presented a Technology Scouting Report[1] where he described a collection of development processes, frameworks and standards. Here Davis points out how:

> "There is no guarantee that even when organizations conform to a particular process model, the software they build is free of unintentional security vulnerabilities or intentional malicious code. However, there is probably a better likelihood of building secure software when an organization follows solid software engineering practices with an emphasis on good design, quality practices such as inspections and reviews, use of thorough testing methods, appropriate use of tools, risk management, project management, and people management."

Noopur's work starts off with the definition of some essential concepts for the context of Software Development Process, and they are:

[1] Copyright 2005 Carnegie Mellon University.

Definition 14. ***Process****: "a sequence of steps performed for a given purpose"[41] .*

Davis takes it a step further and defines:

Definition 15. ***Secure Software Process****: The set of activities performed to develop, maintain, and deliver a secure software solution. Activities may not necessarily be sequential; they could be concurrent or iterative.*

Definition 16. ***Process model****: A process model provides a reference set of best practices that can be used for both process improvement and process assessment. Process models do not define processes; rather, they define the characteristics of processes. Process models usually have an architecture or a structure. Groups of best practices that lead to achieving common goals are grouped into process areas and similar process areas may further be grouped into categories. Most process models also have a capability or maturity dimension that can be used for assessment and evaluation purposes.*

The following sections describe strategies that are based in the ideology of working in securing software during the development process. The processes mentioned in these sections have several important practices in common (and they will be discussed in more detail later in this dissertation).

5.1 Secure Software Development Lifecycle

In [199] Symmantec proposes the SSDL, which represents a structured approach toward implementing secure software development. This is an incremental approach which integrates the stakeholders at an early stage of the process, and maintains their involvement in the following stages. Security issues are considered and addressed early in the system's lifecycle, during business analysis, the requirements phase, design and development of each software build. This early involvement allows the security team to provide a quality review of the security requirements specification, attack use cases, and software design. This will also enable the team to more thoroughly understand business needs and requirements (and the risks associated with them), this way, developing the most appropriate system environment using secure development methods, threat-modeling efforts, and so on to generate a more secure design.

The SSDL puts emphasis on early involvement, since the requirements comprise reference point from which success is measured. The security team needs to review the system or application's functional specification. The security test strategies are also to be determined during the specification/requirements phase.

In the SSDL the vulnerabilities are identified, and once it's been determined that it has a high level of exploitability, the respective mitigation strategies need to be evaluated, designed and implemented. Further in the process, it is important to ensure a secure deployment, which means that the software is installed with secure defaults. Afterwards, its security needs to be maintained throughout its existence. This is accomplished with an all-encompassing software patch management process, where emerging

threats are evaluated, and vulnerabilities prioritized and managed. Finally, since it is often unclear whose job security is, Roles and responsibilities need to be defined.

The SSDL has the following six primary components:

SSDL Phase 1: Security Guidelines, Rules, and Regulations

Security guidelines, rules, and regulations must be considered during the project's inception phase. This is described or considered as the "umbrella requirement." Due to the nature of the system, it might have to comply to specific government regulations or legislations, so a system-wide specification is created defining the security requirements that apply to the system. One example could be the Sarbanes-Oxley Act of 2002, which contains specific security requirements.

Not all systems fall under the of these types of guidelines. However, a security policy still should be developed. It is important not only to document the security policy but also to continuously enforce it by tracking and evaluating it on an ongoing basis.

SSDL Phase 2: Security Requirements: Attack Use Cases

Attack Use Cases (a.k.a. Misuse Cases and Abuse Cases) are an important instrument for security requirement documentation and are discussed in more depth later on this dissertation. As mentioned before, it is important not to omit security requirements from any type of requirements documentation. This are the platform on which the software design, implementation, and test case development sit.

Even though security requirements are cataloged as nonfunctional requirements, the SSDL suggests the security engineer should insist that associated security requirements be described and documented along with each functional requirement. They propose that each functional requirement description should contain a "security requirements" section documenting any specific security needs of that particular requirement. Attack use cases can lead to more thorough secure system designs and test procedures.

It is important that these requirements are unambiguous and specific. The authors in [199] give the example that everyone would agree with a statement such as "The system must be highly secure," but each person may have a different interpretation of "highly secure." Security requirements are more commonly described in the form of constraints as to "the system should not" do something ... Attack use cases can be developed that show behavioral flows that are not allowed or are unauthorized.

This phase is also involved with Security defect prevention and requirements traceability. Defect prevention is described as "the use of techniques and processes that can help detect and avoid security errors before they propagate to later development phases". Defect prevention is most effective during the requirements phase, when the impact of a change required to fix a defect is low. If security is in everyone's mind from the beginning of the development lifecycle, they can help recognize omissions, discrepancies, ambiguities, and other problems that may affect the project's security.

Requirements traceability is in charge of the association between each security requirement with all parts of the system where it is used.

SSDL Phase 3: Architectural and Design Reviews/Threat Modeling

Architectural and design reviews and threat modeling represent the third phase of the SSDL. This phase enables developers to comprehend the structure or architecture of the system, in order to implement the adequate security strategies, plans, designs, procedures, and techniques. Having a clear picture of the interacting parts of the system and a clearly defined project, will aid in a construction of a security conscious design, and help eliminate confusion about the application's behavior in later stages of the project lifecycle. This will also help in the identification of the areas of the system that are the most critical or of highest risk. This knowledge enables to give priority on the critical parts of the application first and helps testers to avoid over-testing low-risk areas and under-testing the high-risk ones.

SSDL Phase 4: Secure Coding Guidelines

Secure Coding Guidelines is the fourth phase of the SSDL. This phase has the main purpose of minimizing (or eliminating if possible,) vulnerabilities in the implementation. Developers need to understand how vulnerabilities get into software so they can learn how to prevent them. As mentioned in a previous chapter, a design vulnerability is a flaw in the design that keeps the program from operating securely no matter how perfectly it is implemented by the coders. On the other hand, implementation vulnerabilities are caused by security bugs in the actual coding of the software.

Software developers and testers go through training on how to develop secure code by adhering to these secure coding standards and guidelines. The same coding guidelines aid testers to develop test cases and verify that the standard is in fact being followed.

SSDL Phase 5: Black/Gray/White Box Testing

The fifth phase of the SSDL is Black/gray/white box testing (this topic is later described in more detail.) Testing is not a new activity in software development lifecycle. However, all its aspects need to be correctly planned. The environment, the plan, resources, databases, and the setup of scripts are some of the aspects to keep in mind in the testing.

It is important to conduct evaluation activities to avoid false positives and/or false negatives, and to document security problems via system problem reports.

SSDL Phase 6: Determining Exploitability

Determining exploitability is the sixth phase of the SSDL. The ideal scenario is to have every vulnerability discovered in the testing phase of the SSDL, be easily fixed. A vulnerability's exploitability is an important factor in the risk evaluation. This information is then used to prioritize the vulnerability's remediation among other development requirements, such as implementing new features.

The following five factors are described by the authors in order determine a vulnerability's exploitability:

- The access or positioning required by the attacker to attempt exploitation
- The level of access or privilege yielded by successful exploitation
- The time or work factor required to exploit the vulnerability
- The exploit's potential reliability
- The repeatability of exploit attempts

Exploitability needs to be constantly re-evaluated because it tends to get easier over time.

After the Development

The SSDL considers the following activities that are to be done outside the development process period.

Deploying Applications Securely

The process of deploying and maintaining the application securely occurs at the end of the lifecycle, but designing the application for secure deployment needs to start early.

Additionally, the secure deployment has to be monitored constantly, and vulnerabilities have to be managed.

Patch Management: Managing Vulnerabilities

After you develop the software using the SSDL, it is important to put a patch management process in place to allow for managing vulnerabilities. To accomplish this objective, services maintain comprehensive databases of vulnerabilities, malicious code, security risks, exposures, malicious IP addresses, and other relevant information. Whenever a user is alerted about a potential vulnerability, the exploitability is analyzed and then it is important to determine whether a patch is required.

Roles and Responsibilities

As mentioned before, it is often unclear whose responsibility security really is. If it is the responsibility of the people in charge of the network or the software architect and designer, or anyone else. In order for effective security testing to take place, roles and responsibilities have clear.

The book proposes the following distribution of tasks:
The program or product manager is responsible for establishing the security policies. They can be based on standards or other security practices. The product or project manager also is responsible for handling a security certification process if no specific security role is defined. Architects and developers are responsible for providing design and implementation specifications, determining threats, and performing code reviews. Testers drive critical analyses of the system, take part in threat-modeling efforts, determine and investigate threats, and build white box and black box tests. Program managers manage the schedule and own individual documents and dates. Security process managers can oversee threat modeling, security assessments, and secure coding training.

If security issues are suspected during development, you should just fix the code to remove any doubt that there could be a security issue. But if the software is already deployed, a fix becomes very expensive, so the techniques of determining exploitability should be used so as not to generate additional costs and work for your customers unless absolutely necessary.

5.2 Microsoft SDL

Microsoft also believed that in order to secure software it is important to detect and remove those vulnerabilities early in the development lifecycle, so they proposed the "Trustworthy Security Development Lifecycle"[90]. They believe that this process will not only help them reduce the number and severity of defects, but also withstand security attacks.

The SDL implements several secure practices throughout the development process, but also include other activities outside of it, for example the mandatory security training for its software development personnel. It has been reported that Microsoft has had good results from products developed using the SDL [143], so this section will briefly describe the proposed process.

SDL uses Microsoft's experience by adding specific checks and measures during the development process. Below are described the SDL stages:

Stage 0: Education and Awareness

Microsoft describes that a key part of their success with the SDL has been the executive support, education, and awareness. They make strong emphasis on getting everyone committed to the process, from the workers to the boss, which can be a difficult task.

Leadership is highly encouraged, as is staying updated in security developments and training. Monitoring of vulnerability databases, staying up to date in coding defects and secure design and similar rapidly changing knowledge, are important activities considered by the SDL.

Stage 1: Project Inception

Great attention is paid to security from the start of the project, this stage includes important steps such as:

- Assign the Security Advisor (The security person who guides the development team through the SDL process.)
- Make sure the bug-tracking process includes security and privacy bug fields.

Stage 2: Define and Follow Design Best Practices

This phase consists in developing design specifications that describe how to implement security features. The SDL takes into account some common secure design practices such as the following:

- Reduction of the attack surface.
- Fail-safe defaults.
- Separation of privilege.
- Least privilege.

Stage 3: Product Risk Assessment

The main purpose of this stage is to determine the best way to spend resources when developing the software. The highest-risk components are identified in order to estimate level of effort and priority of the different modules.

Stage 4: Risk Analysis

Here risk is mainly evaluated through threat modeling, which is a way to understand the potential security threats , determine risk, and establish appropriate measures. This technique enables awareness of security dependencies and assumptions and provides an understanding on the assets the product is trying to protect.

Stage 5: Creating Security Documents, Tools, and Best Practices for Customers

The SDL considers important to provide detailed security information to customers. This is mainly concerned with aiding the end user in securely deploying the systems and so that they can understand the security implications of the configurations decisions they make. This stage is also concerned with informing customers on both the threats that exist and the tradeoffs between risk and product functionality. This is of great impact to the user, since they cover their needs of installing, maintaining and using the system in a secure fashion.

Stage 6: Secure Coding Policies

Secure coding best practices are defined in order to have a secure implementation. Some good practices encouraged by the SDL are the following:

- Use the latest compiler and supporting tool versions.
- Use defenses added by the compiler.
- Use Source-code analysis tools.
- Do not use banned functions.

Along with the restriction of certain functions, secure alternatives must be provided. It is important to give good guidance and education to the developers so they can stick to the policies without much trouble.

Stage 7: Secure Testing Policies

Testing is always an important part of development, since it provides validation of a secure implementation. The SDL testing phase requires the following steps:

1. Fuzz testing.
2. Penetration testing.
3. Run-time verification.
4. Re-reviewing threat models.
5. Reevaluating the attack surface.

Stage 8: The Security Push

A security push is about integrating everyone into the secure mentality. It consists of a team-wide focus on threat model updates, code review, testing, and documentation. This is not a process of defined duration, but it is rather defined by the amount of code that needs to be reviewed throughout the development process, until the code is fairly stable and obtains a certain level of quality. The push itself has a focus on legacy code, but it is not a quick fix to this. Bugs are filed but not fixed at this stage; this is done afterwards.

Stage 9: The Final Security Review

The goal of this stage is to determine if the software is ready to be shipped, from a security point of view. This is mainly performed by the central security team, and commonly a few months before the software is complete. The review verifies that the followed the SDL was followed correctly by the development team during the product's entire development lifecycle.

Stage 10: Security Response Planning

This stage is mainly concerned with responding correctly to the discovery of security vulnerabilities in your software. The SDL takes into consideration the possibility of making mistakes in the development and the likely emergence of new kinds of vulnerabilities. The reality is that one can not be sure that the system is 100% secure, so the team should be prepared for the discovery and elimination of insecurities.

Stage 11: Product Release

The SDL assumes the existence of a formal "sign off" process for releasing software to users. The process should be concerned with a satisfactory agreement that the SDL was followed correctly, the requirement that no bugs of a specific severity exist and that the software is in compliance with the corresponding legal requirements.

Stage 12: Security Response Execution

This phase of the SDL reinforces the concern to protect the customers. The Security Response execution consists of two major parts: To respond to security defects and work with people who find security issues in your code, and to learn from mistakes. This knowledge is then used to adjust SDL (it is updated twice a year).

5.3 CLASP

The Comprehensive, Lightweight Application Security Process is an initiative based on the field work by Secure Software employees from decomposing many development life

cycles. It is an application security process and plugin to the Rational Unified Process (RUP) [185].

CLASP is an activity-driven, role centric process that proposes a set of best practices to integrate security into your software development lifecycles.

Practices

CLASP proposes the following practices:

1. Institute awareness programs
2. Perform application assessment
3. Capture security requirements
4. Implement secure development practices
5. Build vulnerability remediation procedures
6. Define and monitor metrics
7. Publish operational security guidelines

Activities

Table 5.1 shows in the Activities of CLASP in chronological order along with the people associated with the activity. These are the activities to integrate during the development of the system, however it is not always necessary to do all of them. It is best to integrate them one at a time, giving priority to those that are the most appropriate to the team or project. It is also possible that some activities are only applicable to certain applications (for instance one that will use a back-end database.)

Implementation Guide

CLASP provides an implementation guide to aid project managers in the evaluation of activities to determine whether or not to adopt them. The following information is provided of each activity:

- Activity applicability.
- A discussion of the risks associated with not performing the activity.
- An indication of implementation cost in terms of frequency of activity, calendar time, and man-hours per iteration.
- A discussion of dependencies between the various process pieces.

As an additional aid, CLASP provides several "example roadmaps", focusing on common organizational requirements.

Supporting Artifacts

In order to perform the activities in an efficient manner, CLASP provides a complete glossary describing important concepts, principles and standards, and a knowledge base of dozens of classes of vulnerability.

The vulnerability information is in the form of a Root-cause database and it provides comprehensive background information on several kinds of problems with illustrative examples and detailed information to avoiding, detect, and fix the problem.

Some other artifacts CLASP provides include the following:

- A list of common security requirements and a checklist of security concerns to consider when building new requirements.
- A guide to building supplementary specifications surrounding security (with sample business rules and common constraints.)
- A guide for performing architectural security assessments (or threat models).
- A security testing checklist.
- A guide for visually expressing security properties of a system (such as a set of extensions to many UML 2.0 diagram types to accommodate security concerns.)

5.4 iCMM and CMMI security

In order to evaluate the software capability of contractors, in 1986 Watts Humphrey, the SEI, and the Mitre Corporation created a software maturity framework (based on IBM's concepts). By 1991, the SEI published the Capability Maturity Model version 1.0, which described principles and practices which define software process maturity [2]. The CMM[2] helps software organizations improve along an evolutionary path into a disciplined software processes. As result of several revisions, and with the sponsorship of the U.S. Department of Defense teamed up with the National Defense Industrial Association (NDIA), the Capability Maturity Model Integration, and the FAA-iCMM (Federal Aviation Administration Integrated Capability Maturity Model) were eventually released. The figures 5.1 and 5.2 presented in [52] describe the Areas of Focus of these.

While both the CMMI and iCMM provide a framework in which safety and security activities may take place, some specific security practices not addressed. Security concepts are mentioned in descriptions but not in the inherent components of the models [92]. The models lacked an emphasis in vulnerability reduction, this is why in 2002 an initiative to extend the CMMI and the iCMM for Safety and Security was developed.

[2]CMM, Capability Maturity Model, and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. Team Software Process and TSP are service marks of Carnegie Mellon University.

Source Documents for the Extension

As part of the effort to integrate security and safety to the Maturity Models, a process of identification of "best practices" was involved, and the following source documents were used with this purpose:

Source Documents for security:

- ISO 17799: Information Technology - Code of practice for information security management (discussed later in more detail.)
- ISO 15408: The Common Criteria (v 2.1) (discussed later in more detail.)
- Systems Security Engineering CMM (v2.0.)
- NIST 800-30: Risk Management Guide for Information Technology Systems.

Source Documents for safety:

- MIL-STD-882C: System Safety Program Requirements.
- IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Systems.
- DEF STAN 00-56: Safety Management Requirements for Defence Systems.

Most of these documents deal with concepts out of the scope of this dissertation, and for this reason are not covered in more detail.

Goals and practices of the application area

The purpose of the Safety and Security areas is to establish and maintain a safety and security capability, define and manage requirements based on risks. The Goals and practices of the application area are [93]:

Goal 1 - An infrastructure for safety and security is established and maintained.

- Ensure safety and security awareness, guidance, and competency.
- Establish and maintain a qualified work environment that meets safety and security needs.
- Ensure integrity of information by providing for its storage and protection and controlling access and distribution of information.
- Monitor, report, and analyze safety and security incidents and identify potential corrective actions.
- Plan and provide for continuity of activities with contingencies for threats and hazards to operations and the infrastructure.

Goal 2 - Safety and security risks are identified and managed.

- Identify risks and sources of risks attributable to vulnerabilities, security threats, and safety hazards.
- For each risk associated with safety or security, determine the causal factors, estimate the consequence and likelihood of an occurrence, and determine relative priority.
- For each risk associated with safety or security, determine, implement, and monitor the risk mitigation plan to achieve an acceptable level of risk.

Goal 3 - Safety and security requirements are satisfied.

- Identify and document applicable regulatory requirements, laws, standards, policies, and acceptable levels of safety and security.
- Establish and maintain safety and security requirements, including integrity levels, and design the product or service to meet them.
- Verify and validate work products, delivered products, and services to assure that safety and security requirements have been achieved.
- Establish and maintain safety and security assurance arguments and supporting evidence throughout the life cycle.

Goal 4 - Activities and products are managed to achieve safety and security requirements and objectives.

- Establish and maintain independent reporting of safety and security status and issues.
- Establish and maintain a plan to achieve safety and security requirements and objectives.
- Select and manage products and suppliers using safety and security criteria.
- Measure, monitor, and review safety and security activities against plans, control products, take corrective action, and improve processes.

CLASP Activities	**Related Project Roles**
Institute security awareness program	Project Manager
Monitor security metrics	Project Manager
Specify operational environment	Owner: Requirements Specifier, Key Contributor: Architect
Identify global security policy	Requirements Specifier
Identify resources and trust boundaries	Owner: Architect, Key Contributor: Requirements Specifier
Identify user roles and resource capabilities	Owner: Architect, Key Contributor: Requirements Specifier
Document security-relevant requirements	Owner: Requirements Specifier, Key Contributor: Architect
Detail misuse cases	Owner: Requirements Specifier, Key Contributor: Stakeholder
Identify attack surface	Designer
Apply security principles to design	Designer
Research and assess security posture of technology solutions	Owner: Designer, Key Contributor: Component Vendor
Annotate class designs with security properties	Designer
Specify database security configuration	Database Designer
Perform security analysis of system requirements and design (threat modeling)	Security Auditor
Integrate security analysis into source management process	Integrator
Implement interface contracts	Implementer
Implement and elaborate resource policies and security technologies	Implementer
Address reported security issues	Owner: Designer, Fault Reporter
Perform source-level security review	Owner: Security Auditor, Key Contributor: Implementer; Designer
Identify, implement and perform security tests	Test Analyst
Verify security attributes of resources	Tester
Perform code signing	Integrator
Build operational security guide	Owner: Integrator, Key Contributor: Designer; Architect; Implementer
Manage security issue disclosure process	Owner: Project Manager, key Contributor: Designer

Table 5.1: The CLASP Activities with their associated Roles [30]

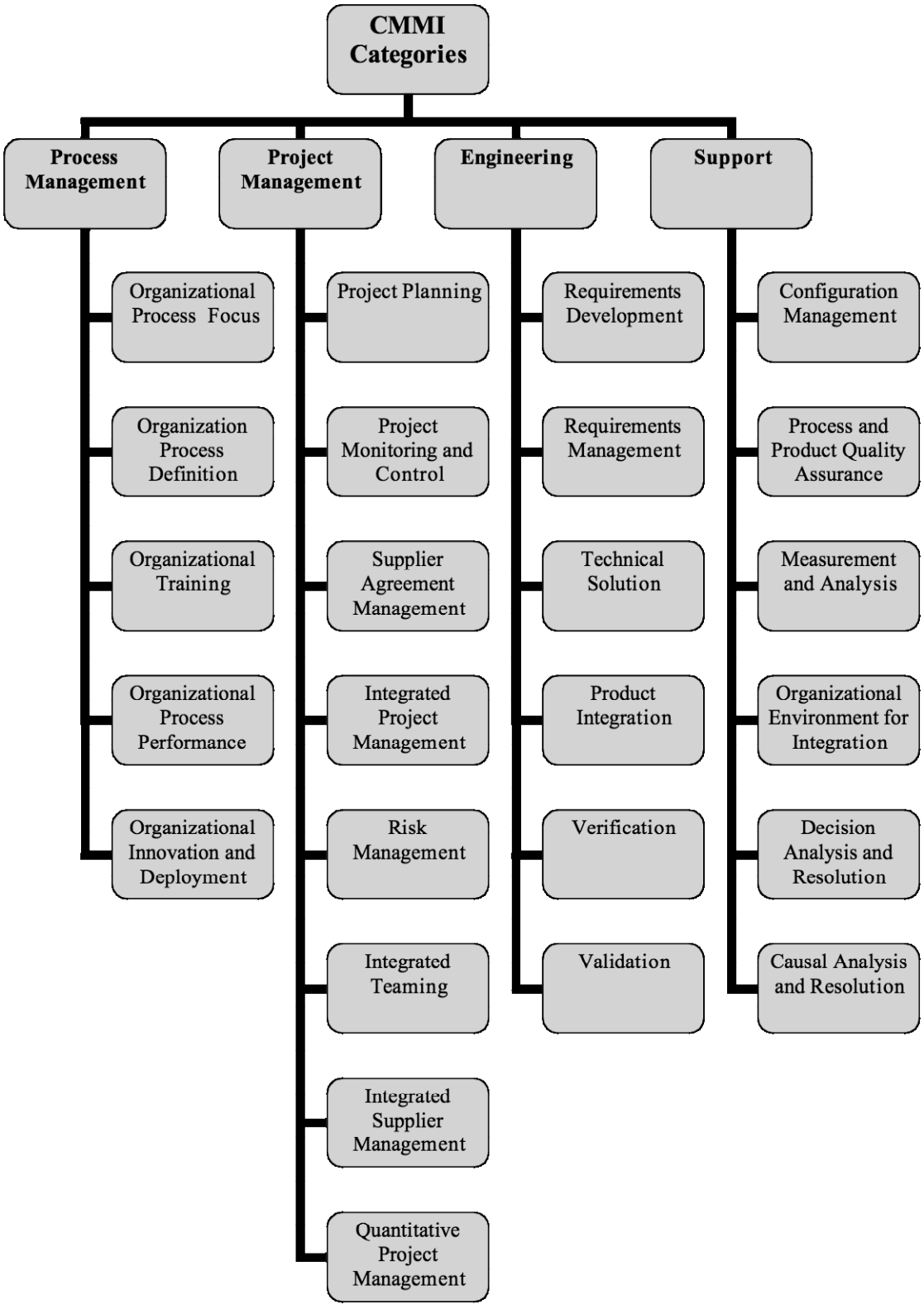

Figure 5.1: CMMI Process Areas [52]

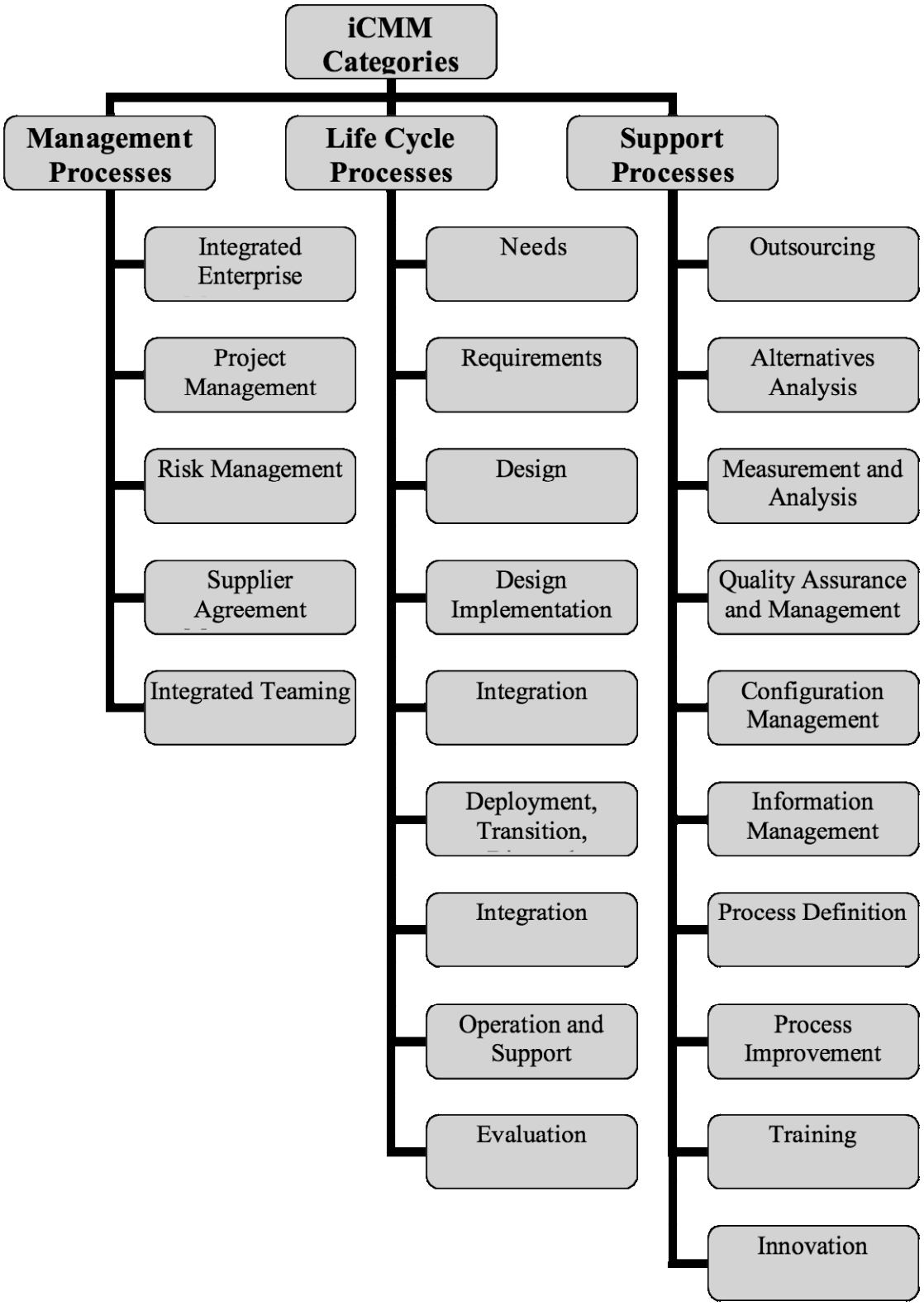

Figure 5.2: iCMM Process Areas [52]

5.5 Correctness by Construction

Correctness by Construction was developed by Praxis Critical Systems as a strategy to produce high-integrity software and extremely low defect rates (fewer than 0.1 defects per thousand lines of code according to [80]). This method is governed by two main principles:

1. Don't introduce errors in the first place.
2. Remove any errors as close as possible to the point that they are introduced.

These principles enable one to eliminate defects at the earliest possible stage of the process. In support to this 2 essential principles, there are seven more presented in [48], and they are:

1. Expect requirements to change.
2. Know why you're testing.
3. Eliminate errors before testing.
4. Write software that is easy to verify.
5. Develop incrementally.
6. There is no silver bullet, focus on the difficult problems first.
7. Software is not useful by itself (user manuals, business processes, design documentation, well-commented source code, and test cases are needed as well.)

Strategies

In order to achieve the Principles, the following strategies are employed:

1. Use a sound, formal notation for all deliverables (unambiguous specification).
2. Use strong, tool-supported methods to validate each deliverable (i.e. static analysis.)
3. Carry out small steps and validate the deliverable from each step.
4. Saying things only once (eliminate redundancy in specification.)
5. Design software that is easy to validate.
6. Do the hard things first.

Process

The process of Correctness consists of the following steps:

1. Requirements
2. Specification
3. High Level Design
4. Detailed Design
5. Test Specifications
6. Module Specifications
7. Code
8. Building
9. Commissioning

The Core process is depicted in figure figure 5.3:

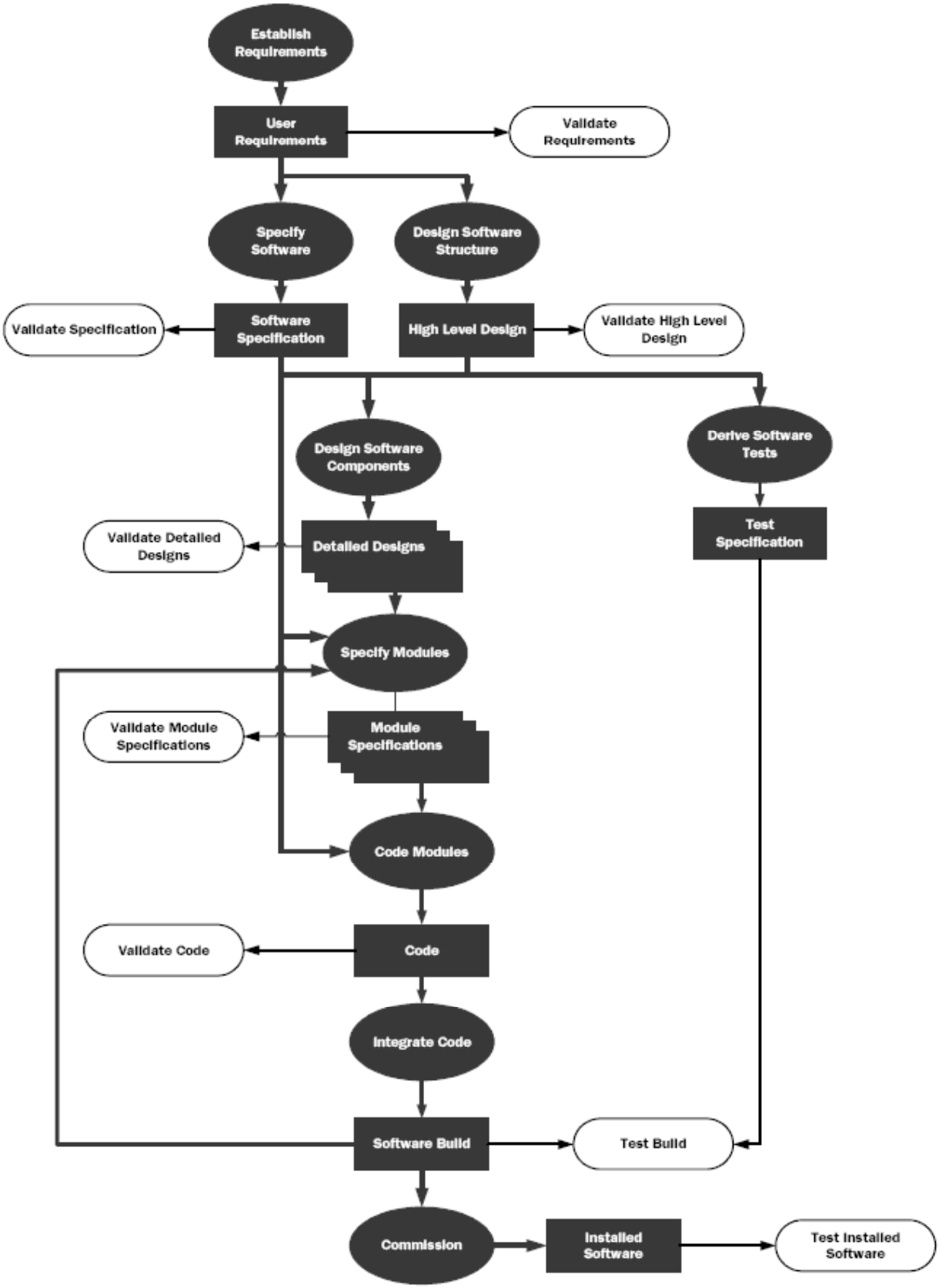

Figure 5.3: The Correctness by Construction Core Process

Chapter 6

Standards and Best Practices

This chapter describes very important standards that, even though they are not necessarily focused at the development process, they cover practices that can be applied during the process or enhance the security through other activities or principles. In addition to describing several important strategies and activities, some security aspects of management and compliance are covered by these standards.

In this chapter we can also find some very important, effective and generally accepted practices. Several of these have already been mentioned as part of the Software Development models in the previous chapter.

6.1 ISO/IEC 27002

The ISO/IEC 27002 (a.k.a. Code of practice for information security management) was first issued in 1995 to provide a set of controls comprising best practices in information security. It is an internationally-accepted standard meant to be a single reference point for good practices of information security.

The standard has evolved throughout the years, and this has been reflected in several name changes. The British Standards Institute, or BSI, first released it as the British Standard 7799. It consisted of 2 parts the "Information Technology - Code of practice for information security management" and the "Information Security Management Systems - Specification with guidance for use." After a period of review, in December 2000, BS 7799 was adopted by ISO/IEC and was released as ISO/IEC 17799. By 2005, several updates were made and was renamed accordingly as ISO/IEC 17799:2005. The current ISO/IEC 27002:2005 is simply a number change from ISO/IEC 17799:2005 made in July 2007, in order to bring it into the same numbering sequence as other information security standards.

An important particularity of the standard worth noticing, is that it is concerned with the security of information assets, not just IT systems. In practice, however, a large percentage of the information is processed, stored, and managed by the IT systems, so the security incidents of information are more common here.

The standard contains hundreds of best-practice information security control mea-

sures that organizations should consider to satisfy their objectives. These practices are not mandatory, but the decisions are left up to the users to select as to implementing those that best suit them. Liberty is also given to adopt controls not listed in the standard, as long as their objectives are satisfied.

Not mandating controls makes the standard very flexible to implement, but it also makes it difficult to assess whether an organization is fully compliant to it, hence there are no formal compliance certificates against ISO/IEC 27002.

Critical Success Factors

The standard describes a set of factors important to the successful implementation of information security within an organization:

- Security policy, objectives and activities that reflect business objectives;
- An approach to implementing security that is consistent with the organizational culture;
- Visible support and commitment from management;
- A good understanding of the security requirements, risk assessment and risk management;
- Effective marketing of security to all managers and employees;
- Distribution of guidance on information security policy and standards to all employees and contractors;
- Providing appropriate training and education;
- A comprehensive and balanced system of measurement which is used to evaluate performance in information security management and feedback suggestions for improvement.

The mind map in figure 6.1 summarizes structure of the standard. The main sections of the ISO/IEC 27002 are listed below:

The Sections of ISO/IEC 27002

Section 0: Introduction

This section introduces the concepts surrounding information security. It also serves as a guide to using the standard.

Section 1: Scope

Section 1 presents recommendations for the distribution of responsibility concerned with initiating, implementing, or maintaining security.

Section 2: Terms and definitions

Here "Information security" is defined as the security services we mentioned early in this dissertation. "Preservation of confidentiality, integrity and availability of information." Other key terms are further defined in this section.

Section 3: Structure of this standard

Section 3 explains the contents of the standard. It is focused on control objectives, suggested controls, and implementation guidance.

Section 4: Risk assessment and treatment

ISO/IEC 27002 covers a brief description of the topic of risk management.

Section 5: Security policy

In this section, management is motivated to define a policy and support information security.

Section 6: Organization of information security

This section deals with handling the security of information within the organization. It emphasizes the need for the Senior management to provide direction and commit their support. The section also describes the importance of how the definition of Roles and responsibilities should take place and the important links that should be established with the authorities and stakeholders.

Section 7: Asset management

The organization understands what information assets it has, and to manage their security accordingly. All assets should have a defined owner, and inventory should be elaborated and maintained describing the assets and their location.

Section 8: Human resources security

Suitable security awareness, training, and educational activities should be taken under consideration. Not only are the roles necessary, but the security responsibilities should be taken into account at recruitment. Employees and IT users should be made aware, educated and trained in security procedures. Another important concern is the security aspects of a person's exit from the organization, for example, the return of assets and removal of access rights.

Section 9: Physical and environmental security

This section covers the physical aspects of security. The protection of the equipment and cabling against intentional or accidental damage or loss, and maintenance aspects. This section also describes the need for controls to protect sensitive IT facilities from unauthorized access.

Section 10: Communications and operations management

The standard covers aspects related to the documentation of responsibilities and procedures and changes to IT facilities. It also describes the importance of security requirements when outsourcing or considering other third party service. Other important topics of anti-malware controls and user awareness are described. The importance of back-ups of information and appropriate network management (including activities regarding private networks and managed firewalls etc..)

Security in communication procedures should be in place to protect information in transit, including mediums such as electronic messaging and business information systems. Finally, other practices such as audits and fault logging and system monitoring should be implemented.

Section 11: Access control

Section 11 presents several recommendations to prevent unauthorized access to IT systems, networks, and data. Access should be in harmony with access control policy, which establishes permissions according to the job necessities (rights of access defined profiles). Appropriate documentation should be done regarding the allocation of privileges and management of passwords. Users should be made responsible for choosing strong passwords (that they can remember) and keep them confidential.

Access to network services is also restricted and controlled inside the organization and between organizations. Policy for secure, remote access to systems should be established and reviewed periodically. Correct authentication through unique user IDs should be used and inactivity timeouts should be applied. There should also be formal policies about the secure use of portable PCs, PDAs, and similar devices.

Section 12: Information systems acquisition, development and maintenance

This section is focused in paying attention in the security aspects in the development, acquisition testing, and implementation of the systems. Activities range from the early stages of the systems development, to the data entry, processing and output validation controls. Cryptography necessities should be defined, such as digital signatures, non-repudiation, management of keys, digital certificates, and adequate algorithms. It is also important to stay up to date in vulnerabilities in systems used to apply relevant patches promptly.

Section 13: Information security incident management

This section exposes the importance of having an incident reporting/alarm procedure, along with the response procedures. This is an activity that should achieve continuous improvement with a correct administration of knowledge and collection of forensic evidence.

Section 14: Business continuity management

The purpose of this section is to educate on disaster recovery planning, business continuity management, and contingency planning. Plans are documented and tested with continuity in order to minimize the impact of security incidents that happen (despite the preventive controls having been in place.)

Section 15: Compliance

It is important to have compliance with corresponding legislation such as copyright, data protection, protection of financial data and other vital records, cryptography restrictions, etc. This is achieved with the correct definition of policies, security reviews, testing, and other practices of the standard.

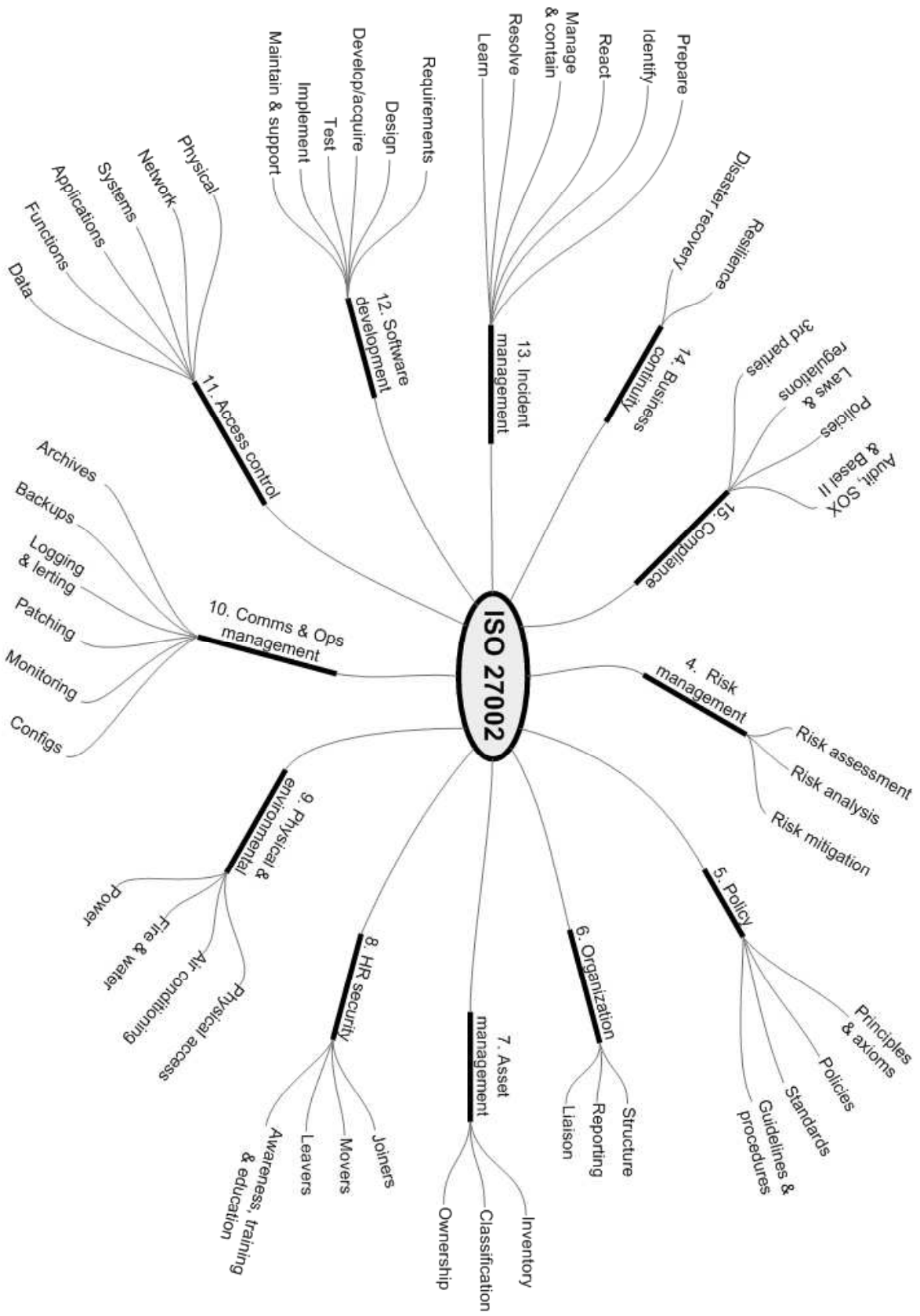

Figure 6.1: ISO/IEC 27002 Mind Map [68]

6.2 Common Criteria (ISO 15408)

The Common Criteria (CC) is a publicly available international standard (ISO/IEC 15408 [99]) that provides an assurance framework of computer security. This standard assists in the specification of security requirements, it helps vendors implement and/or make claims about the security of their products, and enables testing laboratories to evaluate the products in order to determine if they actually meet the claims. The CC documentation is presented in three documents. The first one is introductory and describes key concepts and principles of security evaluation and describes the model of evaluation. The second section describes the security functional requirements to serve users of products in the specification and provides templates for security functional requirements. The third and final document includes security assurance requirements, and defines the seven Evaluation Assurance Levels (EALs).

Participating Organizations

The following organizations contributed to the development of the Common Criteria [100]:

- Australia/New Zealand: The Defence Signals Directorate and the Government Communications Security Bureau respectively;
- Canada: Communications Security Establishment;
- France: Direction Centrale de la Sécurité des Systèmes d'Information;
- Germany: Bundesamt für Sicherheit in der Informationstechnik;
- Japan: Information Technology Promotion Agency
- Netherlands: Netherlands National Communications Security Agency;
- Spain: Ministerio de Administraciones Públicas and Centro Criptológico Nacional;
- United Kingdom: Communications-Electronics Security Group;
- United States: The National Security Agency and the National Institute of Standards and Technology.

Source Standards

The CC originated from the unification of the following three standards:

- ITSEC - Developed in the early 1990s by France, Germany, the Netherlands, the UK, and others.
- CTCPEC - The Canadian standard first published in May of 1993.

- TCSEC - The United States Department of Defense DoD 5200.28 Std developed in the late 1970s and early 1980s.

Below is a list of key artifacts and concepts of the standard:

Target Of Evaluation (TOE)

This is the product in the form of software, firmware and/or hardware (accompanied by guidance) that is to be evaluated. Such evaluation is to validate the target's security features.

Protection Profile (PP)

This is an artifact that identifies security requirements desired for a particular product type (for example, smart cards used to provide digital signatures, or network firewalls). Section two of the CC provides a set of requirements for specific types of products, from which users may choose to form their Protection Profile (to use as templates if such product type is defined.) PP's can include both the functional and assurance requirements. On the other hand, customers may choose to acquire only those products certified against the PP.

Security Target (ST)

This artifact defines the security properties and capabilities of the target of evaluation. For instance a network firewall will not have the same functional requirements as a smart cards. The ST is commonly made public in so the potential customers analyze the security features that have been certified by the evaluation.

Security Assurance Requirements

The Security Assurance requirements consists of descriptions of the metrics used and the evaluation made during development and evaluation of the system in order to assure compliance. A standardised language is used to assist in creating an exact description and avoid ambiguity. The Common Criteria provides a catalogue of these requirements, and they may be different from one evaluation to the next.

Evaluation Assurance Levels

The Evaluation assurance level represents the numerical rating assigned to the target to describe the assurance requirements fulfilled. Each EAL represents a "level of strictness" verified by the assurance requirements for a product. Common Criteria defined seven levels, EAL1 being the most basic (and easiest to achieve and evaluate) and EAL7 being the most strict, but not necessarily more secure. This only means the TOE has been more extensively validated.

The seven evaluation levels are:

- EAL1: Functionally Tested. Applies when you require confidence in a product's correct operation, but do not view threats to security as serious. An evaluation at this level should provide evidence that the target of evaluation functions in a manner consistent with its documentation and that it provides useful protection against identified threats.

- EAL2: Structurally Tested. Applies when developers or users require low to moderate independently-assured security but the complete development record is not readily available. This situation may arise when there is limited developer access or when there is an effort to secure legacy systems.

- EAL3: Methodically Tested and Checked. Applies when developers or users require a moderate level of independently-assured security and require a thorough investigation of the target of evaluation and its development, without substantial reengineering.

- EAL4: Methodically Designed, Tested, and Reviewed. Applies when developers or users require moderate to high independently-assured security in conventional commodity products and are prepared to incur additional security-specific engineering costs.

- EAL5: Semi-Formally Designed and Tested. Applies when developers or users require high, independently-assured security in a planned development and require a rigorous development approach that does not incur unreasonable costs from specialist security engineering techniques.

- EAL6: Semi-Formally Verified Design and Tested. Applies when developing security targets of evaluation for application in high-risk situations where the value of the protected assets justifies the additional costs.

- EAL7: Formally Verified Design and Tested. Applies to the development of security targets of evaluation for application in extremely high-risk situations, as well as when the high value of the assets justifies the higher costs.

The main goal of the CC was to reduce the evaluation of computer products for defence or intelligence to be done only against one set of standards. Figure 6.2 depicts the specification framework for the TOE or product/system. Common Criteria evaluations are performed on computer security products and systems, for instance, the Sun Java System Identity Manager, Microsoft Internet Security and Acceleration Server 2004, 3Com Embedded Firewall V1.5.1,Cisco Firewall Services Module (FWSM), and many more [23].

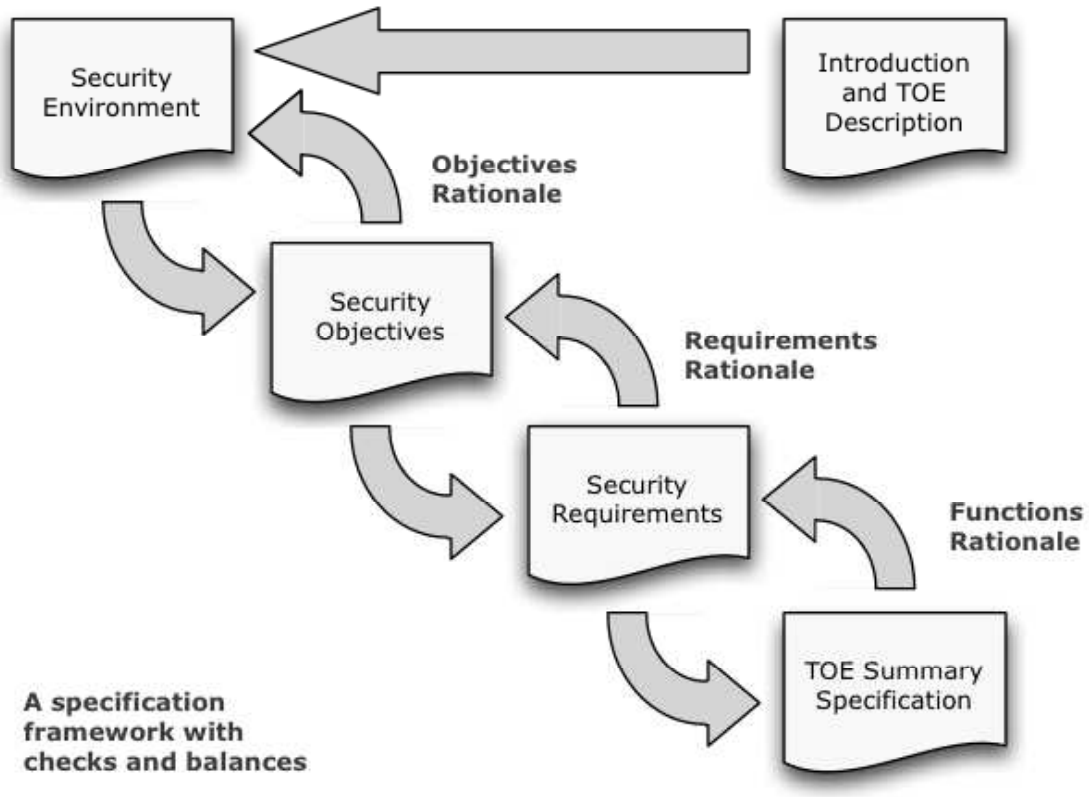

Figure 6.2: PP/ST specification Framework [117].

6.3 The Standard of Good Practice

The Standard of Good Practice for Information Security was developed by the Information Security Forum, with the purpose of producing an international standard. It is mainly based on ISF knowledge and international and national standards (such as ISO 17799). It contains practices that not only involve the development of secure systems, but go further in other important areas where security needs to be contemplated. The standard was made publicly available for the following effects:

- To promote good practice in information security in all organizations worldwide.
- To help organizations which are not Members of the ISF to improve their level of security and to reduce their information risk to an acceptable level.
- To assist in the development of standards that are practical, focused on the right areas, and effective in reducing information risk.

The Standard of Good Practice covers several areas of security. They are resumed in table 6.1

The information of each aspect of security covered by the standard is divided into areas, and these are the divided into a set of sections. The aspects are briefly described below. The complete structure of the standard is described in the appendix.

Security Management

This aspect deals with keeping risks under control through management commitment, allocation of resources, and promotion of good practices. The areas of involved are:

- SM1 High-level direction
- SM2 Security organization
- SM3 Security requirements
- SM4 Secure environment
- SM5 Malicious attack
- SM6 Special topics
- SM7 Management review

Aspect of Security	Issues Probed	Scope and Coverage
Security Management	The commitment by top management to promoting good information security practices and allocation of appropriate resources.	Security management within: • a group of companies (or equivalent) • part of a group (eg a business unit) • an individual organization (eg a company or a government department).
Critical Business Applications (to the success of the enterprise)	Security requirements of the application and the arrangements made for identifying risks and keeping them within acceptable levels.	Critical business applications of any: • type (including transaction processing, process control, funds transfer, customer service and desktop applications) • size (eg applications of thousands of users or just a few).
Computer Installations	Identification of requirements for computer services and requirements of how the computers are set up and run in order to meet those requirements.	Computer installations: • of all sizes (including the largest mainframe, server-based systems and groups of PCs) • running in specialised environments (eg a purpose-built data centre) or in ordinary working environments (eg offices, factories and warehouses) • driven by any kind of OS (eg IBM MVS, Windows 2000 or UNIX).
Networks	How requirements for network services are identified and how the networks are set up and run in order to meet those requirements.	Any type of communications network including: • WANs or LANs • large scale (eg enterprise-wide) or scriptsize scale (eg an individual department) • based on Internet technology (eg intranets or extranets) • voice, data or integrated.
Systems Development	How business requirements (including information security requirements) are identified and how systems are designed and built to meet those requirements.	The status of developments of all types, including: • projects of all sizes • conducted by any type of developer (eg specialist unit/ departments, outsourced or business users) • based on tailor-made software or application packages.

Table 6.1: Summary of The Standard of Good Practice

Critical Business Applications

In this security aspect, the main concern is to give priority to those applications critical for the operation of business. The level of criticality makes possible the identification of business risks and the level of protection required to keep the given risks within acceptable limits. This aspect is made up of the following areas:

- CB1 Security requirements
- CB2 Application management
- CB3 User environment
- CB4 System management
- CB5 Local security management
- CB6 Special topics

Computer Installations

With this security aspect, the Standard provides support to computer installations. The areas of this security aspect are more focused in the post-development phase. The areas in which Computer Installations is divided are the following:

- CI1 Installation management
- CI2 Live environment
- CI3 System operation
- CI4 Access control
- CI5 Local security management
- CI6 Service continuity

Networks

The Networks aspect covers the security of the channel that provides the access to the information and the systems. It demands robust network design, well-defined network services, and the practices are to be applied equally to local and wide area networks, and to data and voice communications.

- NW1 Network management
- NW2 Traffic management
- NW3 Network operations
- NW4 Local security management
- NW5 Voice networks

Systems Development

This security aspect of the standard is closely related to the concepts described in the last chapter. It deals with building security into systems during their development process with the use of defined disciplines to be observed throughout the stages of the cycle. The main areas are the following:

- SD1 Development management
- SD2 Local security management
- SD3 Business requirements
- SD4 Design and build
- SD5 Testing
- SD6 Implementation

6.4 Misuse and Abuse cases

Misuse case is a technique to elicit security requirements. This artifact serves as complement to the Use cases, which are more focused in functional aspects of the system. The Misuse cases model the possibilities of someone breaking the system. A misuse case is a type of use case which describes what the system owner does not want to occur [156] and how software should react to such illegitimate use [86].

A simple example of a Misuse diagram is shown in 6.3.

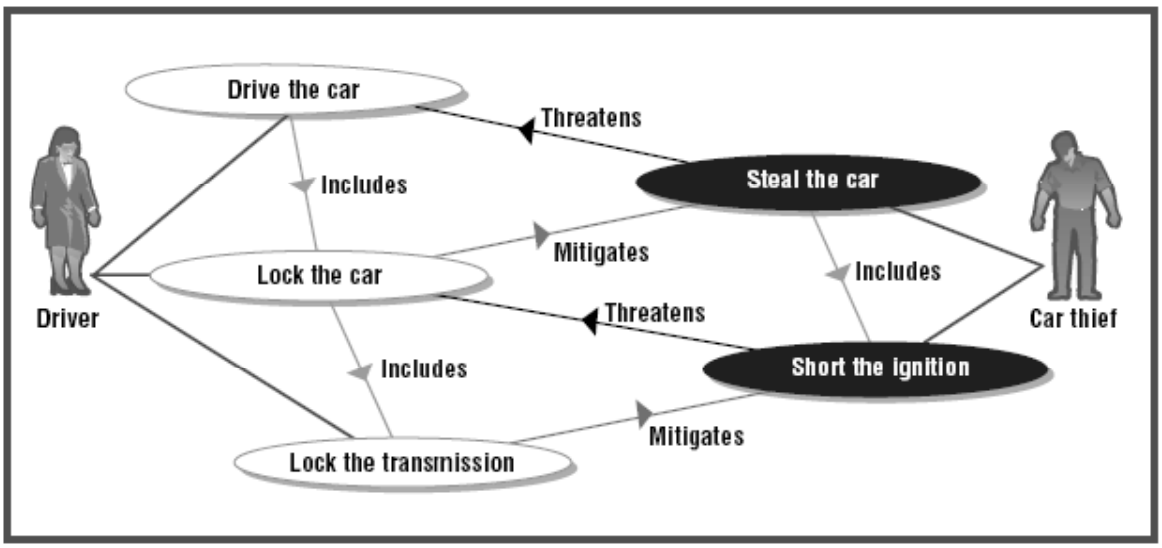

Figure 6.3: Use/misuse-case diagram of car security requirements [4].

The diagram shows the use cases and main actor on the left, and the misuse cases and attacker on the right (although not always will have the malicious actor). The figure also presents the countermeasure to the threat. This is of great use to evaluate priorities from requirements and constraints. These models might be broken down into more detailed documents in order to have a better understanding of the system and evaluate the treats.

Use cases may also serve to describe exception-handling mechanisms that respond to failures and prevent chaos. The response can lead to the resumption of normal operations or to a safe shutdown, as when a train stops after it passes a danger signal. The scenario description can also be useful in defining test cases. Figure 6.4 illustrates an example for web portal security. Misuse cases are a strong system design artifact that should be integrated during the software lifecycle.

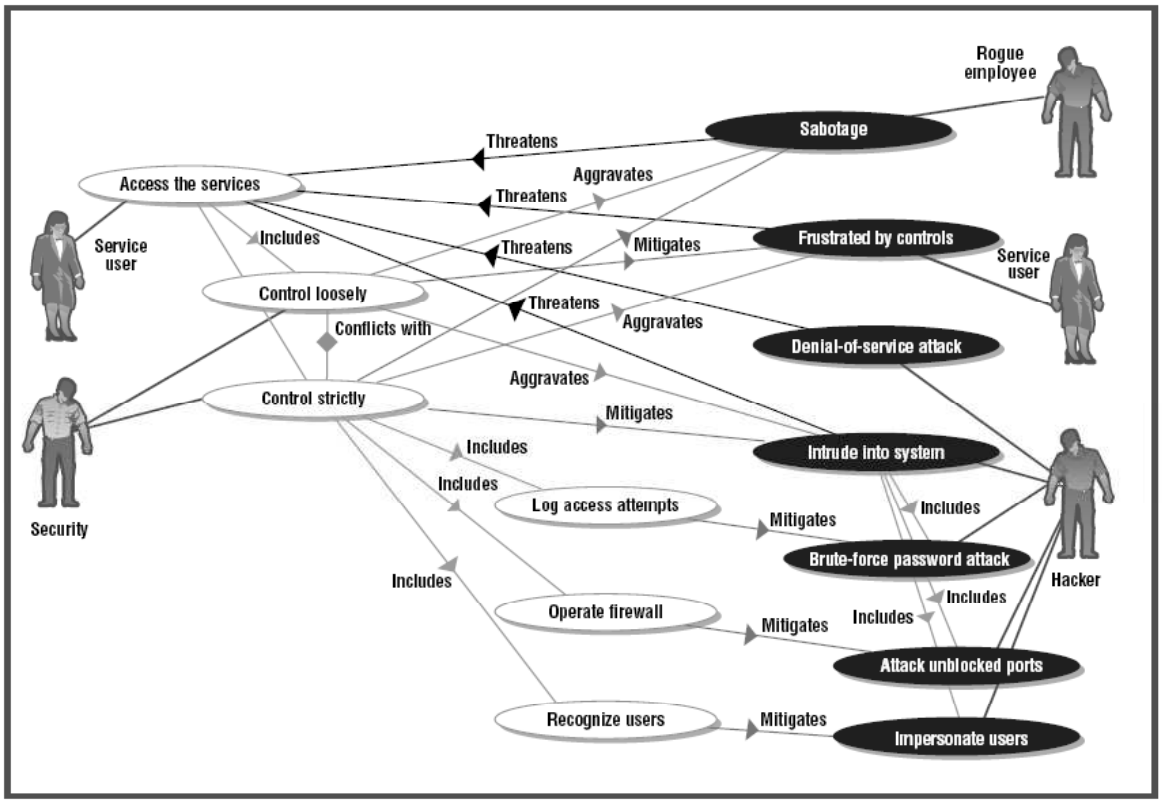

Figure 6.4: Use and Misuse cases for Web portal security

6.5 Reducing Attack Surface

The attack surface of a system is the set of all areas where an attacker can enter a system and possibly mitigate a security service. Therefore, the less "doors and windows" in the system, the smaller the possibility of an attacker to get in. Big Attack Surface = Big Security Work.

The areas of an attack surface are defined through the following three dimensions [91]:

- **Targets and enablers**. An attack target is a specific process or data resource on System. On the other hand, enabler is any accessed process or data resource that is used as part of the means of the attack (and not the main target of it).
- **Channels and protocols**. Communication channels refers to the means the adversary uses to access to the targets on System.
- **Access rights**. These rights are associated with each process and data resource of a state machine.

The attack surface is made up of code, interfaces, services, protocols, and practices available to all users, with a strong focus on what is accessible to unauthenticated users. So reducing the surface implies reducing the amount of code executing by default, reducing the volume of code that is accessible to untrusted users by default and limiting the damage if the code is exploited.

20 RASQ attack Vectors	**Formal**
Open sockets	channels
Open RPC endpoints	channels
Open named pipes	channels
Services process	targets
Services running by default	process targets, constrained by access rights
Services running as SYSTEM	process targets, constrained by access rights
Active Web handlers	process targets
Active ISAPI Filters	process targets
Dynamic Web pages	process targets
Executable vdirs	data targets
Enabled accounts	data targets
Enabled accounts in admin group	data targets, constrained by access rights
Null sessions to pipes and shares	channels
Guest account enabled	data targets, constrained by access rights
Weak ACLs in FS	data targets, constrained by access rights
Weak ACLs in Registry	data targets, constrained by access rights
Weak ACLs on shares	data targets, constrained by access rights
VBScript enabled	process enabler
Jscript enabled	process enabler
ActiveX enabled	process enabler

Table 6.2: Mapping Attack Vectors into dimensions [91]

Waiting to reduce your attack surface late in product development is often very difficult because changing functionality late guarantees regression errors. Other features probably depend on the functionality to work in a specific way, and you just changed it.

The debugger song (to the tune of 99 bottles of beer in the wall):

99 little bugs on the code, 99 little bugs. You fix one bug, compile it again, 100 little bugs in the code.

As each week in the development cycle passes by, you should measure the attack surface of your product. Start with a baseline, and then each week count all the items identified in the previous section using various scanning tools. In some cases, you may need to write your own tools if you have entry points specific to your application. If the attack surface count goes up, determine why it went up, and see if you can drive it back down. When engineers know you are measuring the attack surface, they will try not to stand out by increasing the attack surface in the first place. Define your minimal attack surface early in development, and measure it during development. After all of this, if you decide that you must ship an application with a large attack surface, obviously, that's bad. Not only does it mean your customers may be attacked by default, but it means you have a lot of code to review.

6.6 Shades of Analysis

Testers should bit limit themselves to only test functionality. There are other important tests that can be done to validate the security of systems. Some of these approaches require source code availability and others don't. This section describes White box and black box testing and analysis methods both attempt to understand the software, but they use different approaches depending on whether the analyst has access to source code.

6.6.1 White Box Analysis

White box analysis (also called clear box, and static analysis) consists in analyzing source code. Such analysis may be done by manually reviewing code. However, this is not convenient for most projects because of the large quantities of code being produced [32]. This strategy is normally very effective in finding programming errors early. The main disadvantage of this approach is the need of source code availability, which is not at all common when acquiring commercial software of even free or tailor made software. Sometimes it is possible to obtain source code by decompiling binary code. White box analysis is commonly automated with tools that look for specific patterns. Another disadvantage of white box testing is that it is common to have a high rate of false alarms due to the lack of understanding the tools have of the code. Finally, static analysis tools are usually aimed at a specific programming language, such as C++ or Java, so the choice of tool to adopt depends first on the support for the language, and then on the detection rates and minimization of false positives.

6.6.2 Black Box Analysis

Black Box Analysis (or fault injection) consists on supplying improperly formatted input to a target software to cause failure in order to analyze and determine if there are errors [85]. The key to this method is to find input unexpected by developers that would make the system behave in unexpected ways. Not all failures may have security consequences, but others may allow attackers to have access to the system. In observing the failure of software is important to determine if it fails insecurely (for example if attackers may gain access, or if error messages display too much information.) This testing modality does not need the source or binary code.

Black box testing is characterized for being the choice of strategy of many attackers. Even though it is not as effective as white box testing, black box testing is much easier to accomplish since it commonly requires less tester skills than white box [85]. It is often impossible to test a real program's input space, but it is important to test as much as possible because black box tests assimilate more closely a real attack on software.

6.6.3 Gray Box Analysis.

As it can be guessed, gray box analysis consists of combining the previous 2 by combining several tools. The common goal of testing methods is to reveal possible risks and the use of gray box techniques combines both methods in an effective way. Figure 6.5 graphically depicts the testing domains.

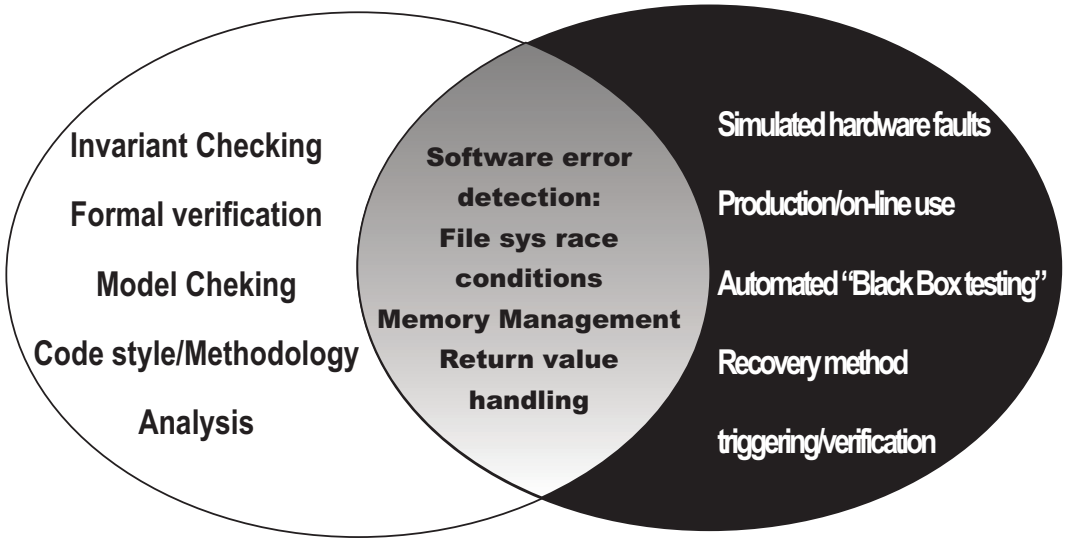

Figure 6.5: The overlapping verification domains of static analysis and software fault injection (black box) [24].

6.7 Penetration Testing

Along with the different shades of testing described in the previous section, penetration testing is of the most important and effective testing strategies (according to [7] it is the most commonly applied mechanism used to gauge software security, but most commonly misapplied). The importance of this technique resides in the fact that it is done from the point of view of the attacker (which is why black box analysis is also considered part of penetration testing).

As Kolodgy described Penetration testing [107] as a "localized, time-constrained, and authorized attempt to breach the security of a system using attacker techniques." It is important to point out that this effort is done without knowledge that an attacker will probably not have, like source code, passwords or others.

During a penetration test, organizations actually try to replicate in a controlled manner the kinds of access an intruder or worm could achieve. With a penetration test, network managers can identify what resources are exposed and determine if their current security investments are detecting and preventing attacks.

Main reasons to perform penetration testing [107, 132]:

- To find vulnerabilities and fix them before an attacker does.
- To verify Secure Configurations (in firewalls, IDS, etc..).
- To discover Gaps In Compliance.
- To test New Technology.
- Tells companies whether critical business information is exposed.
- Helps companies allocate IT security resources more efficiently and effectively.
- To view their network through the eyes of an attacker to prevent attack

6.8 Other Practices and Recommendations

This section presents a group of effective techniques to enhance security. These are commonly accepted practices that several standards and organization include. Build security in present several of these as principles or guidelines (at https://buildsecurityin.uscert.gov/daisy/bsi/articles/knowledge/principles.html).

6.8.1 Keep it Simple

As it has already been mentioned, complexity leads to insecurity. The design, implementation, or security mechanisms should be easy to understand and nothing complex. Problems become harder to find in complex systems, especially with enormous amounts of code.

6.8.2 Acknowledge human imperfection

You should keep in mind that people will introduce Vulnerabilities into Your System.

6.8.3 Validated all Input

Using unvalidated input as part of a command to a subsystem can bring problems (for example SQL Injection.)

6.8.4 Initialize Memory

Failing to initialize storage can result in unexpected system behavior. It is better not to assume null values but to make sure memory is null or with adequate starting values.

6.8.5 Design Safe Default Configurations

Correct configuration procedures and recommendations should be designed to aid a secure deployment.

6.8.6 Ensure that the Bounds of No Memory Region Are Violated

As it has already been mentioned, the violation of memory bounds can introduce vulnerabilities. Conscience of this is a first step, but mechanisms to monitor the bounds should be employed.

6.8.7 Use Correct Authentication

Incorrect use of authorization techniques may introduce vulnerabilities. A software system that requires access checks to an object each time a subject requests access decreases the chances of mistakenly giving elevated permissions to that subject. Remember that caching permissions can speed up systems, but you are storing critical information in vulnerable places. Good password practices should also be enforced. For example, it is not wise to use words from the dictionary and to avoid easily guessable passwords. List of the most commonly used (and cracked) passwords are easy to find and we should make sure these are avoided. Clifford Stoll gave a great recommendation when he said:

> "Treat your password like your toothbrush. Don't let anybody else use it, and get a new one every six months."

6.8.8 Remember it is hard to keep secrets

Relying on an obscure design or implementation does not guarantee that a system is secured. You should always assume that an attacker can obtain enough information about your system to launch an attack. Tools such as decompilers and disassemblers allow attackers to obtain sensitive information that may be stored in binary files.

6.8.9 Least Privilege

In [51] it is advised to assign only the minimum necessary rights to a subject that requests access to a resource and this should be restricted to the shortest duration necessary. It is important that all processes, users, and programs be given only the access to system resources that they need, and no more. "If a process does not need to run as root, then it shouldn't." Keep the information on a need-to-know basis.

6.8.10 Securing the Weakest Link

A system is only as strong as its weakest link. Attackers concentrate on finding weak spots in a software system and exploiting them, than to try to break a strong component. For example, some cryptographic algorithms can take many years to break, so attackers are not likely to attack encrypted information communicated in a network. They probably would use some social engineering or other alternative ways to get what they want.

6.8.11 Fail Securely

When a system fails, it should do so securely. An important example is the rollback transaction operation in databases. The confidentiality and integrity of a system depend on a secure way of failing. It is important not to reveal sensitive information about the system upon failing. As mentioned in the SQL injection discussion, attackers could use the information provided in error messages to elaborate attacks.

6.8.12 Separation of Privilege

A system should ensure that multiple conditions are met before granting permissions to an object. Checking access on only one condition may not be adequate for strong security. If an attacker is able to obtain one privilege but not a second, he or she may not be able to launch a successful attack. If a software system largely consists of one component, the idea of having multiple checks to access different components cannot be implemented. Compartmentalizing software into separate components that require multiple checks for access can inhibit an attack or potentially prevent an attacker from taking over an entire system.

6.8.13 Keep System logs

Logs are the surveillance mechanisms to keep a system secure. Changes in logs is often the first sign that a system has been compromised. There are some programs that help automate the tasks of checking logs and other key files. Without this automation, the tasks turns very difficult since one ends up with too much data and no information.

6.8.14 Coding Practices

A list of important coding practices is presented in [139]. Here are some of them:

- Code with reuse and sustainability in mind.
- Use a consistent coding style throughout the system.
- Make security a criterion when selecting programming languages to be used.
- Avoid common, well known logic errors: use input validation, compiler checks to verify correct language usage and perform code review to ensure conformance to specification.
- Use correct encapsulation.
- Ensure asynchronous consistency (to avoid problems like timing and sequence errors, race conditions, deadlocks, order dependencies, and synchronization errors).
- Use multitasking and multithreading safely.
- Implement exception handling.
- Use information hiding.

6.8.15 Firewalls

Firewalls are today one of the most implemented security measures by companies and individuals. Just because we mention that they do not give complete security does not mean one should stop using them. These devices are effective for what they were designed to do, screen network traffic allowing or blocking it based on a set of rules [106]. There are a wide variety of firewalls; hardware and software, packet-filtering, stateful-inspection and proxy-based are some examples and now a days most operating systems come with one included. While this device is very effective, it does not work by itself. The golden rule is to have a "Block everything, allow only what you need", instead of "Allow everything, block only what harms."

6.8.16 Intrusion Detection Systems

The goal of these systems is to identify activities with intention of compromising the security of resources[114]. The IDS is a critical element of system security. IDS perform real-time monitoring, logging and auditing by analyzing different types of traffic, for example, port scans.

6.8.17 Antivirus and Malware detectors

We must remember that wrongly designed and implemented software presents a dangerous threat. However, there also exists well designed and implemented software for bad purposes, such as viruses, worms and trojan horses. For the purpose of mitigating these malicious software the following solutions were developed:

Antivirus software is the most commonly known threat countermeasure. It's main goal is to prohibit a virus to get into the system, but this is hard to do. The "next best approach" [168] is to detect the infection, identify the specific virus, and remove it.

Malware detection is done mainly through the following three methods [169]:

- Signature Detection (of software).
- Change Detection (of files).
- Anomaly Detection (in behavior of software.)

6.8.18 Detecting and preventing Buffer Overflows

As we have already mentioned, buffer overflows are one of the most important threats. Misha Zitser [201] presented table 6.3, to describe a group of strategies to detect or prevent buffer overflows.

Approach	PROS	CONS
Dynamic Testing	Program Values Known at a runtime. If an error occurs at runtime, tracking it is easy. Source code is not always required. Language independent.	Coming up with test cases to exercise all execution paths is very difficult. Not only that, one must come up with test cases that trigger a buffer overflow. Program execution during testing slows down a great deal, and memory usage goes up. Being able to run the program is not always convenient. (testing device driver code requires the device)
Compiler-based dynamic Prevention tools (Canaries)	Can stop specific types of buffer overflows from causing damage during run-time.	Buffer overflows are transformed into a DoS by terminating the program. Source code needs to be recompiled with special compiler. Compiler patches usually slow down program execution significantly.
Language approach	Stop buffer overflows by using safe libraries or string modules.	Many of these approaches transform buffer overflows into denial of service. Interfacing with old libraries is difficult. Not ideal for securing legacy code.
Static-Dynamic Hybrids	Safer dialects such as Cyclone minimize the chance of buffer overflows. Porting from C to CCured or Cyclone is more realistic than porting to Java. Cyclone and CCured can stop buffer overflow attacks at runtime.	Porting from C to Cyclone is not trivial and requires one to modify 10 % of the code. Cyclone and CCured turn a buffer overflow into a denial of service.
Operating Systems Approach	Can prevent many buffer overflow attacks (esp., stacksmashing)	Linux kernel patch does not prevent heap-based overflows. Making the stack non-executable still leaves room for certain stack-based buffer overflows, i.e. in signal handlers. Porting a legacy program to a new operating system migh not be trivial.
Static Source Code Analysis	All execution paths can be analyzed without running the program. Discover the root of the problem, so that it can be eliminated.	Source code is always required. Imprecision due to analysis heuristics exists. Sometimes there are many false positives. 100% detection rate is theoretically impossible. Sometimes require users to annotate code.

Table 6.3: Pros and Cons of different approaches to detecting/preventing buffer overflows

Chapter 7

Tools for Software Security

Hundreds of tools have been developed by organizations and individuals to aid in the mission of securing systems. These tools cover a wide variety of threats, are made for different platforms, and there are many commercial options as well as free alternatives. There are some tools that serve as preventive measures, and others that are reactive approaches and even some that can serve as both. In [98], insecure.org presents a complete list of the top network security tools, nevertheless, these type of utilities are not the only ones that enhance security. It is impossible to mention them all, so this chapter presents several of the most relevant and important Linux OS products [106, 98]. It is complicated to evaluate the tools in the different categories for several reasons. Most tools don't fit into only one.

The correct use of tools can help us identify vulnerable parts of our system and to automatically detect and mitigate threats [32]. On the other hand, we must keep in mind that correct configuration, maintenance, and use of these tools is needed in order to use them to their full potential. Furthermore, we must be smart when we acquire the utilities and evaluate certain characteristics [195] :

- We must objectively analyze the claims made by the tool supplier.
- Be aware of the extent to which a tool reports false positives.
- Consider aspects of usability of the tool.

We should always remember to use the right tool for the right task, and consider the fact that many tools complement each other.

7.1 NIST Tool Taxonomy

NIST proposes the following tool taxonomy [57] :

Life Cycle Process or Activity

This refers to the phase of the Life Cycle process in which the tool is used. It can take one of the following values:

- Requirements.
- Design.
- Implementation.
- Maintenance.
- Testing.
- Operation.

Automation Level

This classification aims to describe how much human interaction is needed.

1. Manual procedure e.g., code review
2. Analysis aid e.g., call graph extractor
3. Semi-automated automated results, manual interpretation, e.g., static analyzer for potential flaws or Intrusion Detectors.
4. Automated e.g., firewall

Approach

This attribute defines what the goal of the tool is.

- Preclude.
- Proactively make flaws impossible, e.g., correct by creation.
- Detect.
- Find flaws, e.g., checkers, testers.
- Mitigate.
- Reduce or eliminate flaw impact, e.g., security kernel, MLS.
- React. Take actions upon an event.
- Appraise. Report information, e.g., complexity metrics or call graphs.

Viewpoint

Can we see or "poke at" the internals? External tools do not have access to application software code or configuration and audit data. Internal tools do.

- External e.g., acceptance of COTS packages or Web site penetration tester
- Internal (white box) Static e.g. code scanners, Dynamic e.g execution monitoring

Assessment vs. Development

"DO-178B differentiates between verification tools that cannot introduce errors but may fail to detect them and development tools whose output is part of airborne software and thus can introduce errors."

Sponsor

Who fixes it? Can I get it?

- Academic
- Commercial
- Open
- Proprietary
- Used within a company, either as a service or on their own products.

Price

- 0
- $ (nomial, e.g., up to about $17)
- $$ (up to a few hundred dollars)
- $$$ (significant, thousands of dollars)

Platforms

What does it run on? Linux, Windows, Solaris, ...

Languages/Formats

What is the target language or format? C++, Java, bytecode, UML, ...

Assessment/Quality

How well does it work? Number of bugs. Number of false alarms. Tool pedigree. Maturity of tool. Performance on benchmarks.

Run time

How long does it run or do per unit (LOC, module, requirement)? Is it quick enough to run after every edit? every night? every month? For manual methods, how often are, say, reviews? Is it scalable?

Computational complexity might be separate or a way of quantifying run time.

- Simple
- Decidable
  - P (polynomial time)
  - NP (Non-deterministic polynomial time)
- Undecidable

The following sections briefly describe several of the most important software tools relevant in enhancing security. New tools are developed, and many academic and commercial solutions emerge constantly, which makes it impossible to include them all. It would be ideal to categorize the tools according to the taxonomy just described, but unfortunately most of the information required to define each of the criteria is not available.

7.2 Static Analysis

Code review for security is one of the most common software security practices [32]. These are tools mainly focused in finding bugs early in the development, but may be used later on as well (way later on in the case of legacy code for example). As already mentioned previously, this category of tools is language specific.

7.2.1 Lexical Tools

Within the static analysis tools there are some that only employ lexical analysis techniques, which tend to produce high false alarm rates because of the limited understanding of the program flow. Despite this, these tools are typically fast and easy to integrate with the development process. Some examples of these are:

- Flawfinder is an open source Security scanner for C/C++ code written by David A. Wheeler [194] for use in Linux systems. It works with a database of functions with potential risks.

- ITS4 is a freely available program that checks for potentially dangerous function calls in C code [184, 38]. This software was developed by Cigital and it works across Windows and Unix.
- RATS stands for Rough Auditing Tool for Security. It is an open source tool that checks for potentially dangerous function calls in C code and was developed by Secure Software [162]. It scans C, C++, Perl, PHP and Python source code and works in Windows and Unix systems.

7.2.2 Semantic Tools

A more thorough analysis and a better understanding of programs is needed, and this is achieved with the semantic tools. These type of tools are commonly a lot slower than the static analysis ones, but are more effective in finding bugs. The tools listed below use this approach:

C Verifier: This is Commercial tool developed by Polyspace that finds vulnerabilities in C/C++ code. It supports Windows and Unix operating systems [201].

Archer: Archer, or Array Checker, [35] is a research proprietary software focused in detecting out-of-bounds errors and race conditions in C/C++ by following program paths.

UNO: This is an open source tool from AT&T that works under Linux and is designed to analyze C code to detect the following three common types of software defects [16]:

- Use of uninitialized variable,
- Nil-pointer references.
- Out-of-bounds array indexing.

IDA-Pro: This is a commercial, white box analysis tool from DataRescue [60], and it does not require source code. It works for Windows PE, Mac OS X Mach-O, and Linux ELF executables.

BOON: This is a tool that aims to detect buffer overflow vulnerabilities in C source and was developed by David Wagner [187]. It works on Linux OS.

CQual: CQual [71] detects format string vulnerabilities in C programs (there is also jqual for java), but a programmer needs to identify a few variables as either tainted or untainted. It is academic and is freely available for Windows and Linux.

Eau Claire: Eau Claire [33] is a freely available tool for Linux OS made by Brian Chess, and it checks C programs to find problems like buffer overflows, file access race conditions, and format string threats.

MOPS: MOPS was written by Hao Chen [31] to look for violations of temporal safety properties in C code on Linux systems. The programmers model their safety properties, and then employ MOPS to look for privilege management errors, incorrect construction of chroot jails, file access race conditions, and dangerous temporary file schemes.

Splint: Splint is a free and open source application which analyses C programs for security vulnerabilities. It works on Unix, Linux, OS/2 and FreeBSD systems [112].

PREfast: Microsoft propriety light way c scanner[43]. It uses syntactic and semantic analysis to find vulnerabilities and works on Windows.

Blast: The Lazy Abstraction Software Verification is a Linux OS, c code analyzer tool developed at UC Berkeley [19] and it is freely available.

C++ Test: Commercial tool by Parasoft [137] which analyzes c++ code. There are Windows and Linux versions.

CodeAssure: Commercial static analysis tool for windows and Linux developed by Secure Software that analyzes java and c/c++ [192].

CodeSonar: This is a commercial source code analysis tool by Gramma Tech for detecting vulnerabilities and other defects in C and C++ and Ada. Linux, Windows and Solaris platforms are supported. [145].

Coverity Prevent: This tool is also commercial, and it was developed by Coverity. It is a C/C++ and java bug checker and security scanner. Available for Apple Mac OS X 10.4, Cygwin, FreeBSD, HPUX, Linux, Mac OS X, NetBSD (2.0), Solaris Sparc, Solaris X86, Windows [46].

DevPartner Security Checker: -This checker was developed by Compuware, and it is a commercial solution to scan .Net framework code in Windows OS[83].

McCabe IQ: Commercial, static analysis tool which works for C, C++, C#, Java, Fortran, VB, COBOL, and other languages in Unix and windows platforms [62].

Prexis Engine: Commercial vulnerability scanner for C/C++ and Java/JSP developed by Ounce Labs for both Windows and Linux platforms.[110].

Pixy: Freely available, static analysis tool for Windows and Linux, aimed at the

detection of Cross-site scripting in php code[102].

PMD: This is an open source tool to scan Java source code [167] in Windows and Unix systems.

Pscan: A Linux, open source tool by Alan Dekok that checks for dangerous function calls, detecting format string vulnerabilities in C code [40].

7.3 Dynamic Analysis

As it has already been mentioned, dynamic analysis tools have the advantage of not requiring source code, which may sometimes be the case (COTS). Below are listed some tools of this group.

Chaperon: Chaperon [196] is a commercial tool from Parasoft which works with binary executables in order to detect certain types of defects. This software is available for Windows and Linux.

Valgrind: Valgrind is an open-source suite of tools originally developed by Julian Seward [152] that simulates code execution on a virtual x86 processor, this way detecting bugs.

Hailstorm: This is a Windows OS commercial solution developed by Cenzic [118] to find vulnerabilities in web applications.

Holodeck: Holodeck was developed in Florida tech and is a commercially available test tool for Windows Applications and Services using fault simulation.

NTOSpider, ntoinsight, and ntoweb:. The commercial NTOSpider, and freeware ntoinsight and ntoweb, are tools developed by the company NT Objectives aimed to scan for known vulnerabilities in web applications in MS Windows systems [134, 133].

Appscan: A commercial tool developed by Sanctum (acquired by Watchfire in 2004) to audit web applications[188] in Windows and Linux systems.

WebInspect SPI Dynamics (now part of HP) developed this commercially available web application security assessment tool [59] for Windows Operating system.

Achilles Windows web application security assessment tool developed by Robert Cardona. Achilles is freely available, and it acts as a HTTP/HTTPS proxy.[27].

Nikto Open Source web server scanner for Unix and Windows systems created

by Chris Sullo [39].

Odysseus A freely available Windows tool that acts as a Proxy server for testing the security of web applications [20].

WebScarab Tool for performing all types of security testing on web applications and web services developed by OWASP [138]. It is free to download and there are versions for Linux, Windows and MAC OS X.

SPIKE Spike is a freeware tool designed to analyze protocols through the creation of random tests in a simulated network environment in Linux OS [95].

Paros Free tool written in Java to evaluate the security of web applications [177]. It is available for Windows and Linux.

7.4 Library and Compiler Approaches

There are several solutions to make verification at compilation time. They are listed below:

ProPolice: ProPolice is a tool for stack smashing protection developed by IBM, and it works by inserting a "canary" value in different parts of the memory used, along with runtime checks to make sure the values are unaltered [66]. This program runs on Linux.

StackGuard: It is a patch for gcc compiler which also uses Canary values to check for memory corruption. It was developed by Crispin Cowan for use in Linux systems [198].

Tiny C: Tiny C compiler (TinyCC) [17] is a C compiler developed by Fabrice Bellard, which works by inserting code to check buffer accesses at compile time. It is available for download for Linux and Windows Platforms.

Stack Shield: This is a compiler patch for GCC compiler made by Vendicator for Linux systems [183]. It protects against overwriting of the return address and overwriting function pointers

Libsafe and Libverify: These tools employ both static and dynamic intrusion prevention to enhance security in Windows and Linux systems. They patch C functions that are known to constitute potential buffer overflow vulnerabilities. A range check is made before the actual function call which ensures that the return address and the base pointer cannot be overwritten [12].

Prefix: Prefix is a compile-time tool to detect defects in C and C++ source code

through execution simulation [26] (Microsoft proprietary).

CCured: CCured [127] relies on developer annotations c code in order to perform static analysis and classify pointers as SAFE, SEQ, or WILD. After this, checks are inserted into the executable and the program is analyzed at runtime. The work was supported in part by the National Science Foundation and was developed primarily by George Necula, Scott McPeak, Westley Weimer, Matthew Harren and Jeremy Condit. It is designed for Linux but can work in Windows with cygwin.

7.5 Packet Manipulation and Password Cracking Tools

For Network applications it is very important to have tools to check how a hacker would do, remotely. Other important tools to have are Password cracking tools, since they enhance the security by auditing the strength of those already used.

Wireshark (a.k.a. Ethereal: This is an open source network protocol analyzer for Unix and Windows designed to examine data from a live network or from a capture file on disk. It was developed by Gerald Combs, and it supports hundreds of protocols and media types.

John the Ripper: This is a popular and effective open source password cracker developed by Solar Designer, available for Unix, Windows, DOS, BeOS, Mac OS and OpenVMS [136]. Its purpose is to detect weak passwords.

Cain & Abel: This is a password recovery tool developed by Massimiliano Montoro for the Windows platform. This freeware can recover passwords by sniffing the network, cracking encrypted passwords using Dictionary, Brute-Force and Cryptanalysis attacks, recording VoIP conversations, decoding scrambled passwords, revealing password boxes, uncovering cached passwords and analyzing routing protocols. [124]

L0phtCrack: This is a password auditing and recovery application (now called LC5) [147], developed by Mudge from L0pht Heavy Industries. It is used to test password strength and for recovery of lost Microsoft Windows passwords, by using dictionary, brute-force, and hybrid attacks. The application was produced by @stake after the L0pht merged with @stake in 2000, and @stake was acquired by Symmantec in 2004.

pwdump: Free software by Jeremy Allison which dumps the password database of an NT machine that is held in the NT registry into a valid smbpasswd format file [50]. New versions have been developed by other individuals.

ShowPass: Is freeware developed by Octavian Merches to show the cached Windows passwords, without having to crack them [165].

THC Hydra: This is a network authentication cracker that uses brute force to crack a remote authentication service. It can perform rapid dictionary attacks against more then 30 protocols, including telnet, ftp, http, https, smb, several databases, and much more[82]. It is freely available for Unix and Windows (with cygwin) platforms.

Aircrack: Aircrack is a WEP/WPA cracking tool written by Christophe Devine freely available for Linux and Windows systems. It can recover keys once enough encrypted packets have been gathered. It can attack WPA 1 or 2 networks using advanced cryptographic methods or by brute force.[123]

Airsnort: AirSnort is a wireless LAN (WLAN) tool that recovers encryption keys. It was developed by the Shmoo Group and operates by passively monitoring transmissions, computing the encryption key when enough packets have been gathered [131].

RainbowCrack: RainbowCrack tool is an open source hash cracker for windows developed by Philippe Oechslin. It works with a time-memory trade-off strategy that consists in doing all cracking time computation in advance and store the result in files so called "rainbow table." It does take a long time to precompute the tables, but once the one time precomputation is finished, it is said that it can break any windows password up to 14 characters in a few minutes[154].

Nmap: Nmap is a free and open source port scanning tool written Fyodor. Nmap can also determine the operating system of the target computer. Many operating systems are supported, including Linux, Microsoft Windows, FreeBSD, OpenBSD, Solaris, IRIX, Mac OS X, HP-UX, NetBSD, Sun OS, Amiga, and more [97].

Metasploit: Metasploit is a tool, whose objective is to get to a command prompt on the target computer, and if this happens it is quite possible that the target computer will be under total control in a short time. It provides attack libraries and attack payloads that can be put together in a modular manner [125]. The first version was written by H D Moore, using the Perl scripting language and it is freely available for Linux, BSD, Mac OS X, Windows Cygwin.

SecurityForest Exploitation Framework: This is another open-source software to perform penetration testing [113] developed for Oindows and Unix operating systems. This framework leverages a collection of exploit code known as the ExploitTree, and the Exploitation Framework is a front-end GUI that allows testers to launch attacks through a Web browser.

CORE IMPACT: CORE IMPACT is a commercial penetration testing tool for Windows OS [148] aimed at identifying vulnerabilities in a program, exploit them, and clearly document every step. The software has the ability to install an agent on a compromised computer and then launch additional attacks from that computer. It

was developed by Core Security Technologies.

LANguard: A commercial network security scanner for Windows by GFI [163]. LANguard scans IP networks to detect programs that the machines are running and identify the host OS. It also tries to collect Windows machine's service pack level, missing security patches, wireless access points, USB devices, open shares, open ports, services active on the computer, key registry entries, weak passwords, users and groups, and more. It also includes a patch manager which detects and installs missing patches.

Retina: Commercial vulnerability assessment scanner by eEye for auditing Windows and Linux systems [61]. Retina's scans all the hosts on a network and report on any vulnerabilities found.

7.6 Personal Firewalls (software implementations)

We have mentioned that firewalls should never be the only security measure, but it should definitely be there. A few options are listed below:

Zone Alarm: Freely available (for individual and not-for-profit charitable entity use) and very popular firewall developed by Check Point for Windows operating systems [202]. Advanced features are available in a commercial version.

Netfilter: Netfilter is a packet filter implemented in the standard Linux kernel [81, 128]. Iptables tool is used for configuration and it supports packet filtering (stateless or stateful), all kinds of network address and port translation (NAT/NAPT), and multiple API layers for 3rd party extensions. It includes many different modules for handling unruly protocols such as FTP.

Openbsd PF: The OpenBSD Packet Filter [18] is the firewall tool in OpenBSD. It handles network address translation, normalizes TCP/IP traffic, provides bandwidth control, and packet prioritization.

IP Filter: IP Filter is a software package that can be used to provide network address translation (NAT) or firewall services [141]. It can either be used as a loadable kernel module or incorporated into the UNIX kernel and is distributed with FreeBSD, NetBSD, and Solaris.

Other Firewalls:

- CA Personal Firewall [191].
- Comodo [158].

- Core force [148].
- Lavasoft Personal Firewall [129].
- Kaspersky Internet Security [56].
- Kerio Personal Firewall [178].
- Look 'n' Stop Firewall [161].
- KPTools Skeet for Windows, Linux, Macintosh [108].
- Outpost Firewall and Outpost Firewall Pro [74].
- PCTools [180].
- Routix NetCom for Windows [159].
- Sunbelt Personal Firewall [22].
- Tiny Personal Firewall [166].
- TuxGuardian [28].
- Windows Firewall [122].

7.7 Antivirus and Malware detection Tools

Antivirus is the main countermeasure against malicious software, those programs and scripts meant to do harm.

Norton AntiVirus: Commercial antivirus software by Symmantec available for Windows OS and MAC OS X[173].
Panda: Commercial antivirus Software by Panda Software International for Windows systems [160].
McAfee VirusScan: This is a commercial antivirus and IDS by McAfee for Windows OS[115].
Nod32: Commercial antivirus program developed by Eset for Windows Systems [65].
Ashampoo AntiVirus: Commercial antivirus software by Ashlampoo for windows Systems [8].

7.8 Intrusion Detection Tools:

The following tools are meant to detect

Cybercop: Commercial tool by Network Associates which works on Windows NT, Netware, Solaris, AIX, HP-UX. [106, 49]

Internet security systems Internet scanner Commercial: Commercial software by Network Associates which runs in Windows NT, Solaris, AIX, HP-UX, AS/400 [144].

Nessus: Nessus [151] is a popular and FREE tool for configuration auditing, asset profiling, and sensitive data discovery, but vulnerability updates are delayed 7 days (the commercial version is direct and immediate). It supports the following Operating systems: Windows NT, XP, Netware, Solaris, AIX, HP-UX, AS/400, Mac OS X, FreeBSD.

Snort: This is network intrusion detection and prevention system [157] specialized in traffic analysis and packet logging on IP networks. Through protocol analysis, content searching, and various pre-processors, Snort detects thousands of worms, vulnerability exploit attempts, port scans, and other suspicious behavior. Snort uses a flexible rule-based language to describe traffic that it should collect or pass, and a modular detection engine. It is a commercial tool but there is also an open source version.

OSSEC HIDS: An Open Source Host-based Intrusion Detection System [10] which performs log analysis, integrity checking, rootkit detection, time-based alerting and active response.

Fragroute/Fragrouter: A network intrusion detection evasion toolkit [146] that takes advantage of the lack of reconstruction of packets for a coherent view of the network data (via IP fragmentation and TCP stream reassembly). Fragrouter helps an attacker launch IP-based attacks while avoiding detection. It is part of the NIDSbench suite of tools by Dug Song.

BASE: The Basic Analysis and Security Engine [13] is a PHP-based analysis engine to search and process a database of security events generated by various IDSs, firewalls, and network monitoring tools. Its features include a query-builder and search interface for finding alerts matching different patterns, a packet viewer/decoder, and charts and statistics based on time, sensor, signature, protocol, IP address, and others.

Sguil: Sguil is aimed at network security analysis [172]. It provides real-time events from Snort/barnyard and facilitates Network Security Monitoring and event driven analysis of IDS alerts.

7.9 Cryptography

It is important to include the following programs and algorithms that enhance the security services in different ways.

7.9.1 Symmetric Cryptography

Symmetric cryptography consists in using the same key to encrypt and decrypt the information. Below are described the 2 main algorithms.

DES/3DES: DES stands for Data Encryption Standard (DES) and it is an encryption algorithm adopted as an official Federal Information Processing Standard (FIPS) for the US in 1976. DES is now considered to be insecure for many applications. 3DES, or Triple DES, is the modality of the algorithm where DES is run 3 times to elevate the complexity and eliminate possibilities for statistical analysis [168].

AES: The Advanced Encryption Standard is a block cipher developed by Vincent Rijmen and Joan Daemen, with 128-bit block size and key size of 128, 192, or 256. Published by NIST, who issued a call for proposals for an algorithm to replace DES [169].

7.9.2 Asymmetric or Public Key Cryptography

Asymmetric encryption basically consists in using 2 different keys, one for encryption and one for decryption.

Diffie Hellman: The first public-key algorithm with the main goal of exchanging the keys for symmetric cryptography algorithms. [54].

RSA: Public Key Algorithm developed by Ron Rivest, Adi Shamir, and Len Adleman [149].

7.9.3 Hash and MAC

Hash algorithms are widely used to authenticate information and provide integrity of such information. Below are listed some of the most important algorithms:

MAC: This an algorithm that outputs is a short piece of information used to authenticate a message. A MAC algorithm accepts as input a secret key and an arbitrary-length message to be authenticated. MAC protects both a message's integrity as well as its authenticity [168].

MD5: Message-Digest algorithm was designed by Ronald Rivest as an integrity validation tool based on checksums [168].

SHA: Cryptographic hash algorithms designed by the National Security Agency (NSA) [168].

HMAC: Hash Message Authentication Code is a type of MAC calculated with a secret key [168]. The algorithm was first published by by Mihir Bellare, Ran Canetti,

and Hugo Krawczyk.

RC4: RC4 is the most widely-used software stream cipher and is used in popular protocols such as Secure Sockets Layer (SSL) and WEP. It was designed by Ron Rivest [84].

7.10 Protocols

There are several protocols to enhance security in different types of networks and different types of applications. Below are listed some of the most important of these:

WEP: Wired Equivalency Protocol is a scheme to secure IEEE 802.11 wireless networks. Several serious weaknesses were identified in WEP, so it was superseded by WPA [168].

WPA: Wi-Fi Protected Access is a protocol to secure wireless WPA and it implements the majority of the IEEE 802.11i standard [168]. It was created by the Wi-Fi Alliance.

IPSec: This is a suite of protocols for securing Internet Protocol (IP) communications by authenticating and/or encrypting each IP packet in a data stream [168].

Secure Shell: A network protocol that allows data to be exchanged over a secure channel between two computers [150].

7.11 Application

There are several other ways to enhance security in services and applications. Some of them are listed below:

SSL: Security protocol that provides privacy over the Internet [84] and it was originally developed by Netscape..

Kerberos: It is an authentication service based on session tickets, developed as part of Project Athena at MIT [168].

X.509 Authentication: Standard for public key infrastructure that provides public key certificates and a certification path validation algorithm [42].

7.12 email

Finally, to enhance the privacy and integrity of the now necessity email application, we have the following options:

PGP: PGP stands for Pretty Good Privacy and was developed by Philip Zimmermann. It is a widely used scheme that provides cryptographic privacy and authentication using public key cryptography [45].

S/MIME: The Secure/Multipurpose Internet Mail Extension is a security enhancement for public-key encryption and signing of email encapsulated in MIME [94].

Chapter 8

Security Metrics

William Thompson, Lord Kelvin, said:

> "When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the state of science."

Wat do we need to understand about software? What do we need to measure? How do we determine how much security is enough? It is obviously a wrong approach to skip the security considerations, but it is also definitely inconvenient to overdue it. The goal of developers is to produce software that does what it is supposed to do and nothing else, and to have such solution be economically and operationally feasible (to have optimal Return on Investment). It is important to reduce costs, increase revenue and productivity, and at the same time minimize risk. Being software development such a complex multistage process, there are many things to understand, and therefore measure, about it in order to meet the goal. Security mechanisms, need to be adequate to the characteristics of the system and they need to be justified. After all, no one is happy of spending time or money unless there is evidence that it is absolutely necessary to do so; The quantification of security trade-offs must be done.

Metrics aid in the evaluation and understanding of systems in many ways. In essence, quantifiers in software development projects should provide us with information relating the "before, during, and after" of a system:

- Before: What will we need? What can happen? Define the requirements and resources in terms of a level of quality and predict the outcome.
- During: How are we doing? How should we be doing? Define the current state and make adjustments.
- After: How did we do after all? What can we do better next time? How did it compare with other similar products? What metrics were reliable and which ones were not?

Metrics should be easy to gather, and clearly expressed. Number and percentages are the best way to define scales that meet our needs for evaluation, understanding and quantification. Mark Graff and Kenneth Van Wyk point out [75] that without reliable security metrics "consumers will lack the means to reward manufacturers who produce good code and punish those whose products reek with vulnerabilities."

8.1 At Inception Phase

The resources one can employ to enhance security are money, time, people, tools, and knowledge. When a software project is identified, how do we estimate how much of each resource are we to invest? The answer to this depends on the answer to another important question, what is at stake?

If we are thinking about security, it is because we want to protect something. It may be money, services, information or knowledge, reputation, health, time, market share and anything of value that may be put in danger with information systems. Prioritizing the assets according to their value to the organization is then very useful, but not always that easy to do, since there could be political and procedural difficulties in large enterprises [119]. If you wish to prioritize your testing based on the top areas of risk, how do you identify these top areas of risk?

8.1.1 Application Insecurity Index

Andrew Jaquith proposes the Application Insecurity Index [101] (AII) as a scoring method to identify critical business functions. AII is designed to be quickly obtained as a lightweight interview process or questionnaire.

AII covers the following areas:

- **Business importance** scores consider the application's importance to the organization.
- **Technology outlier** scores put a number on the degree to which the application follows prescribed organizational guidelines for certain security topics.
- **Assessed risk** scores highlight the application's relative riskiness based on whether the application might be considered subject to regulatory inspection or review. It also scores whether the application carries any risks associated with third-party code development or data storage, and whether the application has received a technical security assessment.

Table 8.1 displays the AII scoring questionnaire:

8.1.2 Legislation and Compliance

Should software developers be sued for negligence in malfunction of software? Excellent doctors often get in trouble for making mistakes, but when something goes wrong with

Business Importance Score	Technology Outlier Score	Assessment Risk Score
Business function (1-4 points) • 4 Customer account processing • 3 Transactional/core business or unknown processing • 2 Personnel, public-facing • 1 Departmental/back office **Access scope (1-4 points)** • 4 External public-facing • 3 External partner-facing • 2 Internal enterprise • 1 Internal departmental **Data sensitivity (1-4 points)** • 4 Customer data/subject to regulator fines • 3 Company proprietary & confidential • 2 Company non-public • 1 Public **Availability impact (1-4 points)** • 4 > $10m loss, serious damage to reputation • 3 > $2m loss, minor damage to reputation • 2 < $2m loss, minimal damage to reputation • 1 Limited or no losses	**Authentication (0-2 pts)** • 2 Does not meet requirements or unknown • 1 Partially meets baseline • 0 Fully meets baseline requirement **Data classification (0-2 pts)** **Input/output validation (0-2 pts)** **Role-based access control (0-2 pts)** **Security requirements documentation (0-2 pts)** **Sensitive data handling (0-2 pts)** **User identity management (0-2 pts)** **Network/firewall architecture (0-2 pts)**	**Technical assessment** • 8 Not assessed • 6 High-risk vulnerabilities found • 4 Medium-risk vulnerabilities found • 2 Low-risk vulnerabilities found **Regulatory exposure** • 4 Unknown/no regulatory review • 3 Subject to Sarbanes-Oxley, EU Privacy Directive, California On line Privacy Protection Act (SB 68) • 2 Subject to other regulations • 1 Not subject to regulation **Third-party risks** • 4 Code and data offshore • 3 Code offshore • 2 Outsourced development (US) • 1 In-house development
Total (4-16 points):	Total (0-16 points):	Total (4-16 points):

Table 8.1: Application Insecurity Index

software, who should we point the finger at? Compliance to certain regulations is a great approach to assuring specific security requirements for different types of systems. The difficult part is to decide what requirements to demand, and to enforce such compliance. There currently are no global agreements to any type of technological laws, so criminals the other side of the world often get away with their felonies.

How do we prove software is good? If we can guarantee compliance to official

regulation standards, security in software would be better valued and appreciated. Mandatory emphasis on security in the development at an early stage could be effective, but what specifically should we mandate? The authors of [79] describe approaches like considering all software insecure unless it is somehow proven that is secure, make software makers liable for damages, and define mandatory performance standards. In September 2007, after years of attorneys trying to discover the code behind the breathalyzer responsible for determining a suspect's guilt or innocence, judges finally ordered the release of it [174]. After analysis, the algorithm revealed a lack of compliance to any standard, lack of error detection and no quality check for calibration.

Below is a list of some of the most important and current regulation approaches.

Federal Information Security Management Act (FISMA).

This legislation was implemented in the United States and it establishes that federal agencies must maintain an incident response capability, periodic assessments of risk.

California SB 1386

This is a state law in California, US, which addresses how a company responds to a breach, and describes the requirement of cooperation with law enforcement and prompt notification to affected customers.

Sarbanes-Oxley Act (SOx)

Section 404 of the act requires publicly traded companies to assess the effectiveness of their internal controls for financial reporting in annual reports they submit at the end of each fiscal year. Chief information officers are responsible for the security, accuracy and the reliability of the systems that manage and report the financial data. The act also requires publicly traded companies to engage independent auditors who must attest to, and report on, the validity of their assessments.

It requires companies to implement extensive corporate governance policies, procedures, and tools to prevent, respond and report fraudulent activity within the company. Effective self-policing requires companies to have the ability to acquire, search and preserve electronic data relating to fraudulent activity within the organization.

Health Insurance Portability and Accountability Act (HIPAA)

Health Insurance Portability and Accountability Act (HIPAA) requires the adoption of national standards for electronic health care transactions and national identifiers for providers, health insurance plans, and employers. And, it requires health care providers, insurance providers and employers to safeguard the security and privacy of health data.

Gramm-Leach-Bliley Act

Gramm-Leach-Bliley Act of 1999(GLBA), also know as the Financial Services Modernization Act of 1999, protects the privacy and security of private financial information that financial institutions collect, hold, and process, as well as detect, prevent and respond to information security incidents.

UK Data Protection Act 1998

This legislation regulates processing of information relating to individuals, including the obtaining, holding, use or disclosure of such information.

The Computer Misuse Act 1990

This is an Act of the UK Parliament making computer crime an offence. The Act has become a model upon which several other countries have drafted their own information security laws.

The Family Educational Rights and Privacy Act (FERPA)

This is a USA Federal law that protects the privacy of student education records. The law applies to all schools that receive funds under an applicable program of the U.S. Department of Education. Generally, schools must have written permission from the parent or eligible student in order to release any information from a student's education record.

Payment Card Industry Data Security Standard (PCI DSS)

This standard establishes requirements for enhancing payment account data security. It was developed by the founding payment brands of the PCI Security Standards Council, including American Express, Discover Financial Services, JCB, MasterCard Worldwide and Visa International, to help facilitate the broad adoption of consistent data security measures on a global basis. The PCI DSS is a multifaceted security standard that includes requirements for security management, policies, procedures, network architecture, software design and other critical protective measures.

Personal Information Protection and Electronics Document Act (PIPEDA)

This act supports and promotes electronic commerce by protecting personal information that is collected, used or disclosed in certain circumstances, by providing for the use of electronic means to communicate or record information or transactions.

8.1.3 How much Security?

The resources employed to enhance security should be those estimated to cover the threats identified. Having well defined the security demands, helps to estimate the time,

knowledge, and people required to cover these needs. The risks have been identified, and scoring systems like the AII serve to prioritize in the attention of critical areas. The different mechanisms and tools are choice of development teams.

A key step of security metrics is the definition of a value scale with a corresponding threshold to indicate what constitutes "good enough" security.

In figure 8.1, a block diagram presented in [101] is shown, depicting the logical model of security controls. The diagram clarifies the role of metrics in the security scope.

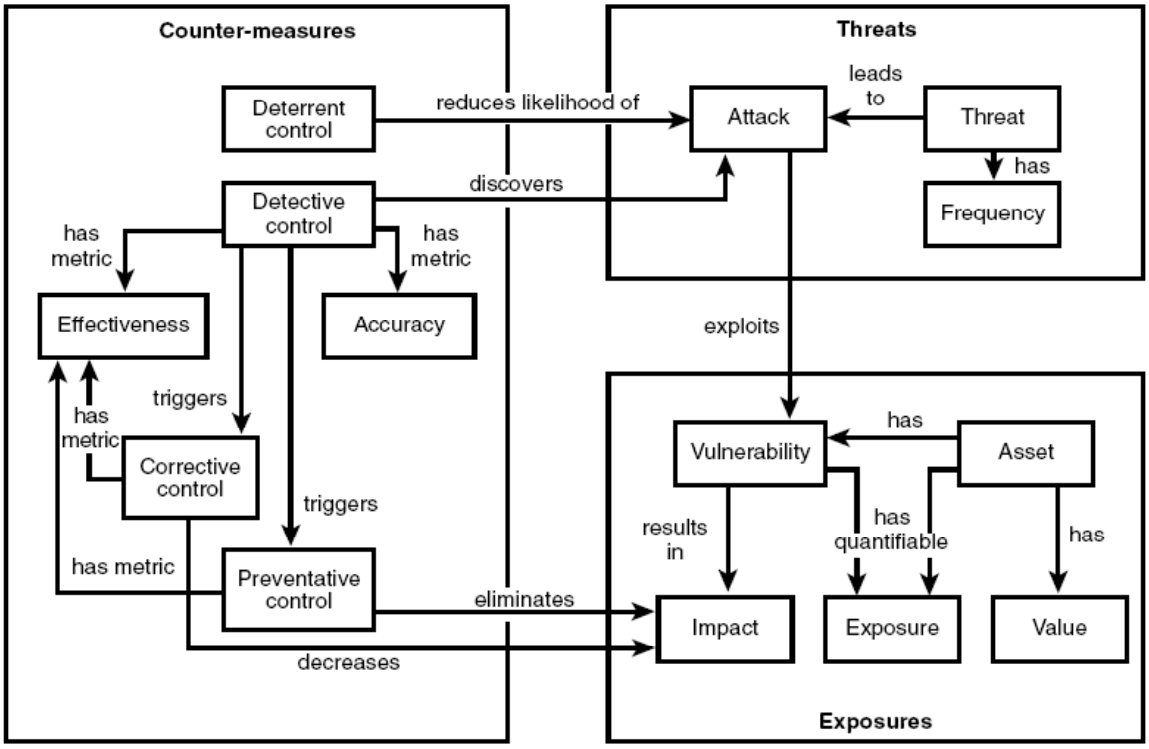

Figure 8.1: Logical Model of IT Security Controls [101]

8.2 At the Development

8.2.1 During Development

The important questions to answer during the development are:

How are we doing? If not very good, can we adjust to needs? Adjustment to the security plan may arise from the revisions during the development. Change can always happen, but it gets more expensive as time passes.

Having insider knowledge about the system provides an advantage to auditing. Table 8.2 presents some metrics to gain knowledge about the system and its current state.

Metric	Purpose
Assessment frequency for developed applications • % with design reviews • % with application assessments • % with code reviews of sensitive functions • % with go-live penetration tests	Measures how often security quality assurance "gates" are applied to the software development lifecycle for custom-developed applications.
Thousand lines of code (KLOC)	Shows the aggregate size of a developed application.
Defects per KLOC	Characterizes the incidence rate of security defects in developed code.
Vulnerability density (vulnerabilities per unit of code)	Characterizes the incidence rate of security defects in developed code.
Known vulnerability density (weighted sum of all known vulnerabilities per unit of code)	Characterizes the incidence rate of security defects in developed code, taking into account the seriousness of flaws.
Tool soundness	Estimates the degree of error intrinsic to code analysis tools.
Cyclomatic Complexity	Shows the relative complexity of developed code. Indicates potential maintainability issues and security trouble spots. Cyclomatic complexity for a code module is defined as the minimum number of paths that in linear combination generate all possible paths through the module.

Table 8.2: Code Security Metrics [101]

Attack Surface

The extent of the Attack surface is also a considered a good security metric. As mentioned in a previous chapter, it aids to quantify the number and kinds of vectors available to an attacker by modeling the potential targets and channels facilitating the attack [91].

8.2.2 After the Development

Is it good enough to release? If not, can we fix it? Measuring the extent of bugs and vulnerabilities through final tests provides understanding of the final product state. As the authors describe it in [6], it is common to find the attitude:

> "Ship it Tuesday and get it right by version 3"

It is important to evaluate that the system has met the security requirements, and there are different ways to find out. The metrics in table 8.3 and those that we saw in table 8.2 are some of the common approaches.

Metric	Purpose
Defect Counting	Shows externally identified defects due to implementation or design flaws.
Vulnerabilities per application • By business unit • By criticality • By proximity	Measures the number of vulnerabilities that a potential attacker without prior knowledge might find.

Table 8.3: Black-Box Defect Metrics [101]

8.3 At Operation -Maintenance & Support

Performance, response, time, capacity, availability and other quality factors could be affected but arising vulnerabilities and exploits. This is why once deployed, an evaluation and support system should be in place to always have the desired quality.

8.3.1 Availability

Security, availability, and reliability are closely related since security incidents commonly lead to downtime. The classic metrics used to measure these factors are uptime and downtime, where downtime can be thought of as being either planned (maintenance, backup, etc..) or unplanned.

8.3.2 Recovery

Table 8.5 lists important Recovery Metrics.

8.3.3 Patching

Many vulnerabilities could be reported in very short periods of time, but how many are really dangerous? Criticality, likelihood, and impact of the threats should be measured in order to prioritize the support effort. Patching is an essential part of system maintenance and keeping software up-to-date, which translates in security assurance [77]. However, the point is to minimize Maintenance costs, and like Mary Ann Pavison expressed, "an ounce of creative destruction is worth a pound of patching."

Business-Adjusted Risk

BAR is a technique presented in [101] invented by Andrew Jaquith and the people in @stake. It is a scoring formula that assigns an overall risk number to vulnerabilities, the higher the score, the higher the risk. BAR classifies security defects by vulnerability

Metric	Purpose
Host uptime (percent, hours) • For critical hosts • For all hosts	Availability measure for critical hosts and other systems.
Unplanned downtime (%)	Shows the amount of change control process variance. Larger numbers indicate a less "controlled" environment.
Unplanned downtime due to security incidents (Percent, hours)	Shows the amount of change control process variance that can be pinned on security issues.
Mean/median unplanned outage (time) • Due to security incidents	Characterizes the seriousness of a "typical" unplanned outage.
System revenue generation (cost per hour) • For critical hosts	Shows business value associated with systems. Can be co-graphed with downtime incidents to show the explicit relationship between incidents and revenue.
Unplanned downtime impact ($)	Quantifies foregone revenue due to the impact of incidents.
Mean time between failures (time)	Characterizes how long systems are "typically" up between failures.

Table 8.4: Uptime Metrics [101]

Metric	Purpose
Support response time (avg)	Average time from outage to response.
Mean time to recovery (time)	Characterizes how long it takes to recover from incidents.
Elapsed time since last disaster recovery walk-through (days). • For nominated business-critical systems.	Shows relative readiness of disaster recovery programs.

Table 8.5: System Recovery Metrics [101]

type, degree of risk, and potential business impact. When assessing an application, for each security defect we calculated a BAR score as follows:

> BAR (1 to 25) = business impact (1 to 5) x risk of exploit (1 to 5, depending on business context)

Risk of exploit indicates how easily an attacker can exploit a given defect. A score of 5 denotes high-risk, well-known defects an attacker can exploit with off-the-shelf

tools or canned attack scripts. A score of 3 indicates that exploiting the defect requires intermediate skills and knowledge, such as the ability to write simple scripts. Finally, only a professional-caliber malicious attacker can exploit certain classes of defects; these receive a score of 1.

Business impact indicates the damage that would result if the defect were exploited. An impact score of 5 represents a flaw that could cause significant financial impact, negative media exposure, and damage to reputation. A score of 3 indicates that a successful exploit could cause limited or quantifiable financial impact, and possible negative media exposure. Defects that would have no significant impact (monetary or otherwise) receive a score of 1.

V-Density

V-Density (vulnerability density) is the key metric of Ounce labs solutions. It is a numerical expression that enables a way to evaluate the vulnerability of your applications. V-Density is calculated by relating the number and criticality of vulnerabilities to the size of application or project being analyzed.

8.4 Looking Back

When maintaining the system requires too much effort and becomes too expensive, the time comes to cut the umbilical cord, and finish supporting it, and maybe make a new version. At the end of the software lifecycle there are several important aspects to evaluate, and lessons to be learned. How did the system compare with similar products? What can we do better next time to optimize ROI? Did we evaluate correctly? Were the metrics reliable? Were the thresholds reliable?.

Developing software is a maturity process guided by experience, although different projects behave differently. It is important to point out that existing and proposed software security metrics focus almost exclusively on counting and comparing vulnerabilities in implemented software, measuring the attack surface, or measuring complexity. No one has yet determined whether the comparison of numbers of vulnerabilities in earlier vs. later versions of software programs or the average number of vulnerabilities per x lines of code are, in fact, meaningful metrics in terms of indicating whether efforts to produce more secure software have succeeded, or for predicting the likelihood that software that appears to be secure in the development environment will, in fact, prove to be secure.

8.4.1 Scorecards

Metrics scorecards [104] require enterprises to adopt new processes for measurement. The goal of a security scorecard is to communicate two things: security effectiveness, and the ability to help the business understand and respond to new threats and opportunities in the future. They provide insight into whether a company is making money, using assets appropriately, or returning value to shareholders.

Four primary perspectives:

Financial

Traditional measures such as profit and loss, return on invested capital, earnings before interest and taxes (EBIT), earnings per share (EPS), and others.

Sample Measures:

- Order or transaction rate
- Number of orders or transactions (total, authorized, unauthorized)
- Number of revenue-generating sessions (total, authorized, unauthorized)
- System uptime
- Downtime cost associated with denial-of-service attacks
- Number of revenue- and cost-accounting events (total, authorized, unauthorized)
- Data flow (customers, vendors, partners)
- Cost of security for revenue-generating systems
- Cost of security for revenue-accounting systems
- Cost of security incidents
- Budget allocations for security (new programs, maintenance)
- Risk indices for revenue-generating systems
- Risk indices for revenue-accounting systems
- Risk indices for cost-accounting systems

Customer

Measures that indicate how effectively the organization serves its customer base, such as customer retention, market share, customer complaints, order fill rate, average deal size, and profit per customer. Sample Measures:

- Percentage of customer wins and losses
- Number of company deals won in which security played a contributing role
- Number (and percent) of customer losses due to security reasons
- Number (and percent) of integrity controls for data exchanged with customers/partners

- Number (and percent) of confidentiality controls for data exchanged with customers/partners
- Quantified losses from accidentally disclosed customer/partner data
- Customer/partner ratings of company security effectiveness
- Percentage of security incidents involving third-party personnel
- Number of data privacy escalations per thousand/million customers, and estimated time/cost to fix

Internal Process

Measures that indicate how effective the organization's internal processes are at satisfying customers and achieving financial objectives. Typical measures include order-to-cash ratios, product development cycle times, labor utilization, days of sales outstanding, and technology support metrics.

Sample Measures:

- Patch latency (mean)
- Password strength (time to break)
- Percentage of security incidents that did not cause damage beyond policy thresholds
- Estimated damage ($) from all security incidents
- Percentage of security compliance reviews with no violations
- Percentage of critical assets/functions with documented risk assessment
- Percentage of critical assets/functions with cost of compromise estimated
- Percentage of critical assets/functions with a documented risk mitigation plan
- Percentage of systems implementing approved configurations
- Percentage of systems in compliance with approved configurations
- Percentage of systems monitored for deviations against approved configurations
- Percentage of systems with the latest patches installed
- Percentage of downtime of critical services due to security incidents
- Percentage of systems affected by incidents exploiting known solutions/patches/workarounds

- Percentage of systems with critical assets assessed for vulnerabilities
- Mean time between failures (MTBF) due to security-related incidents
- Mean time to recover (MTTR) from failures due to security related incidents.

Learning and Growth

Measures that show how well the organization's people are equipped to succeed in the workplace, such as training investment per employee, staff turnover rates, knowledge management metrics, and participation in professional associations.

Sample Measures:

- Ratio of business unit (shadow) security teams to security team staff
- Percentage of staff with security responsibilities
- Percentage of new employees completing security awareness training
- Percentage of users who have undergone background checks
- Fulfillment rate of target external security training workshops and classroom seminars
- Percentage of security staff with professional security certifications
- Number of security skills mastered, average per employee and per security team member

Chapter 9

Proposed Questionnaire for Security Enhancement in the Software Life-Cycle

This chapter is the culmination of this work. After identifying and analyzing the available work in the software security field, the following important questions are proposed that describe the areas of focus for security enhancement. The closer we get to answering the following questions in both research and industry, the closer we are to understanding the security problem, thus be secure.

9.1 What is needed?

The answer to this question identifies the scope of the project. What are the necessities faced? Whether entertainment, business related, or other, the scope defines not only what is needed of a solution, but what is definitely not needed. The feasibility of a system is evaluated in terms of the available technology, budget, and knowledge. The problem to solve should be clear and it must be determined that a software system is the best way to go.

9.2 Who are the stakeholders?

It is of high importance to be clear of who is benefited by the implementation of the systems, since they are the ones who could also be negatively affected by them. When developing the systems, the worries of the stakeholders should be contemplated, as well as the many threats that they might not know about. Some examples of stakeholders are: Owners, government, bank clients, teenagers, medical patients, economists, employees, children, etc.

9.3 What can be lost, how can the stakeholders be affected, or what needs protection?

This question aims to find the Assets relevant to the system and to estimate the value of these. The system might sell products, provide a service, control finance, serve as medical instrumentation and endless more possibilities, so vulnerabilities can put at risk things like health, money, time, reputation, market-share and confidential information. It could be very difficult to identify the impact of security vulnerabilities. As Anderson explains it in [5], the value and revenue of a technological product can vary a great deal and be complicated to estimate, since it depends in user adoption, marketing, and many other factors. How much will a new on-line store loose for an hour of unavailability?

The environment of deployment and the end users of the system can also be of great influence to the general risk of getting attacked. It is not the same to develop a system for banks or military organizations, as it it for non-profit organizations. Different security requirements should be identified accordingly.

9.4 From Whom Should I Protect?

The type of system and end users gives much information of who could cause harm. While government entities could probably be more threatened by cyber terrorists, corporations are more likely to be attacked by competitors or unethical hackers, and so on. The attackers should be identified according to the system to be developed. The attacker profile is useful to determine their technical expertise and probability of being successful in an attack. In chapter 3, a classification and description of the attackers has been presented.

9.5 What do I have to be protected against, what are my threats?

After identifying the assets to protect, it is important to know what to protect the assets from. In chapter 1, section 1.1 several vulnerability databases are listed. It is important to consult them to learn of new vulnerabilities and to gain knowledge of common threats for similar systems to that being developed. It is also important to consult this pages to learn of any bugs or insecurity holes that the platforms, tools and technologies to be employed in the development might have, and to find the fixes and patches for them. The vulnerabilities presented in chapter 4 make up a great percentage of the vulnerability population, and are relatively common in many different types of systems, thus a knowledge of these represents a minimum essential set.

9.6 How will I get protection from the threats at the different stages of the project life?

The mitigation strategies to be used in analysis, design, implementation, testing, configuration, operation should correspond to the vulnerabilities identified and to the type of system to be developed. Chapter 5 presented the different ways to integrate security in the development of the project, and chapter 6 described many mitigation practices and standards for security. Chapter 7 presented a catalog of useful free and commercial tools of security to be used in different stages of the lifecycle. Configuration instructions should be documented in order to avoid insecure defaults. The response plan for operation should also be defined, have patching plans, support line, and a prioritization process for emerging vulnerabilities should be in place.

9.7 How much security is needed?

As is already obvious, no security at all can never be good, and overdoing it would be too expensive, time consuming, and unlikely to ever be 100% secure. The effort should be focused in maximizing ROI. The security effort takes money, time, people and many other resources, but the designation of these for security enhancement should balance the cost of finding a vulnerability VS cost of not finding. Cullinane exposes that there are consequences beyond the economic loss, a damage to the image of the organization. "The reputation risk can literally put you out of business. Twenty percent to 45% of your customers will leave you if you report a system breach"[76]. Chapter 8 presents important metrics that help to evaluate the need for security.

9.8 Is the development secure?

We should have established earlier the security strategies to use in the analysis, design, implementation, and testing phases; but, how do we evaluate if it is in fact secure? Chapter 8 describes several metrics to evaluate the state of the development compared to what was expected of the mitigation strategies adopted. Such measurements should provide insight as to how secure the system is so far, this helps to adjust to adopt different design, tools, or strategies to keep the expected level of security, or in case the project is too far off track, abort it as soon as possible.

9.9 Is the Final Product Secure Enough to Release?

The system is done, but, is it as secure as we expected? Will it cost us more to respond to vulnerabilities? Developers are responsible for the security of the assets, so a final thorough evaluation should be implemented. Different approaches to testing systems are presented in chapter 6, and the metrics to evaluate them were discussed in chapter 8.

9.10 Was the configuration secure?

In systems such as COTS, it is impossible be next to the end users to ensure a secure configuration. This is why the system should provide the adequate mechanisms to make sure the system is deployed securely. Developers should provide manuals, support, and alert the client of the risks of keeping secure defaults.

9.11 Are there Security Incidents in the Operation? Can we do something about it?

Are our security response Plans working? An evaluation of the support provided should be constantly monitored to adjust to needs, as long as it is possible. Chapter 8 describes some metrics to evaluate the final product during operation, in order to prioritize the assistance to clients and correction of emerging bugs in order of criticality.

9.12 How did we do after all?

Time comes to stop supporting the system and look back to the overall success or failure of it. There are several things to take into consideration such as revenue, reputation, comparison with similar products, success of competitors, stakeholder satisfaction, client retention, client complaints, market-share and many more. Chapter 8 lists several metrics that can be used to evaluate the overall success of the system.

9.13 What can we do better next time?

Experience plays an important role in software development organizations. Not all practices are for everyone, and different systems have different threats, attackers, and mitigation strategies. Every project, successful or not, has its learned lessons of successful metrics and strategies, and those that failed. The goal is to be evolutionary and set up a security maturity process.

Chapter 10

Conclusion

This dissertation has exposed the complexities and difficulties behind securing software, which is the Genesis of the other technological securities. The most relevant and important topics of software security have been mapped to show the relationship between each other, and their place has been identified within the whole scope of the problem.

According to [55], Science is "a body of knowledge and a set of processes for advancing that knowledge." The definition continues describing that "It is mankind's interconnected, internally consistent, growing body of knowledge about natural and man-made objects and phenomena of the past, present, and future; a body of knowledge that is based on repeatable experimentation with, or observation of, these natural and man-made objects and phenomena, that is organized and extended using logic and mathematics, and that is validated by the testing of hypotheses."

Unlike Computer Science, and despite of more than 4 decades of studying the problem, software security has not yet become a science. As argued by the authors in [77], better experimental techniques, metrics of security, and predictive models should be established, and security research should be placed on a foundation of science in order to make real progress in the field. While Knowledge management serves to walk towards a software security science, there is still a long way to go.

The proposed Questionnaire for Security Enhancement in the Life-Cycle in the last chapter, identifies areas of opportunity for researchers and walked roads for developers. The better we can answer the questions, the closer we get to Software Security Utopia; making software behave and protect users from systems, other users, and themselves.

Appendices

Appendix A

Vulnerability Taxonomies, Classifications and Lists

This appendix presents different taxonomies and classifications proposed by different researchers and organizations. The tables represent hierarchical classifications, being the highest level the leftmost columns, and the lower levels the rightmost.

Nature of Flaw	Improper Protection	Improper choice of initial protection domain
		Improper isolation of implementation detail
		Improper change
		Improper naming
		Improper deallocation or deletion
	Improper Validation	
	Improper Synchronization	Improper Indivisibility
		Improper Sequencing
	Improper choice of operand or operation	
Time of Induction	During Development	Requirements /Specification /Design
		Source Code
		Object Code
	During Maintenance	
	During Operation	
Exploitation Domain		
Effect Domain		
Minimum Number of components needed to exploit		
Source of Vulnerability Identification		

Table A.1: UNIX System and Network Vulnerabilities [21]

Buffer overruns
Format string problems
Integer overflows
SQL Injection
Command injection
Failure to handle errors
Cross-site scripting
Failure to protect network traffic
Use of magic URLs and hidden forms
Use of weak password-based systems
Improper use of SSL
Failure to store and protect data securely
Information leakage
Trusting network address resolution
Improper file accesses
Race conditions
Unauthenticated key exchange
Failure to use cryptographically strong random numbers
Poor usability

Table A.2: 19 Deadly Sins of Software Security[89]

Operational Fault	Configuration Error		
	Object installed with incorrect permissions		
	Utility installed in the wrong place		
	Utility installed with incorrect setup parameters		
Environment Fault			
Coding Fault	Condition Validation Error	Failure to Handle Exceptions	
		Input Validation Error	Field value Correlation Error
			Syntax Error
			Type and Number of Input Fields
			Missing Input
			Extraneous Input
		Origin Validation Error	
		Access Rights Validation Error	
		Boundary Condition Error	
	Synchronization Error	Improper or Inadequate Serialization Error	
		Race Condition Error	

Table A.3: Security Faults in UNIX [9]

Range and Type Errors	Buffer overflow
	Write-what-where condition
	Stack overflow
	Heap overflow
	Buffer underwrite
	Wrap-around error
	Integer overflow
	Integer coerdon error
	Truncation error
	Sign extension error
	Signed to unsigned conversion error
	Unsigned to signed conversion error
	Unchecked array indexing
	Miscalculated null termination
	Improper string length checking
	Covert storage channel
	Failure to account for default case in switch
	Null-pointer deference
	Using freed memory
	Doubly freeing memory
	Invoking untrusted mobile code
	Cross-site scripting
	Format string problem
	Injection problem (data used as)
	Command injection
	SQL Injection
	Deserlalization of untrusted data

Table A.4: CLASP Classification, attribute 1, Range and Type Errors [164]

Environmental Problems	Rellance on data layout
	Rellance on data layout
	Relative path library search
	Relying on package-level scope
	Insufficient entropy in PRING
	Failure of PRING
	Publidzing of private data when using inner classes
	Trust of system event data
	Resource extraustion(file descriptor, disk space, sockets...)
	Information leak through class cloning
	Information leak through serialization
	Overflow of static internal buffer

Table A.5: CLASP Classification, attribute 2, Environmental Problems [164]

Synchronization and Timing Errors	State synchronization error
	Covert timing channel
	Symbolic name not mapping to correct object
	Time of check, time of use race condition
	Comparing classes by name
	Race condition in switch
	Race condition in signal handler
	Unsafe function call from a signal handler
	Failure to drop privileges when reasonable
	Race condition in checking for certificate revocation
	Mutable objects passed by reference
	Passing mutable objects to an untrusted method
	Accidental leaking of sensitive information through error messages
	Accidental leaking of sensitive information through sent data
	Accidental leaking of sensitive information through data queries
	Race condition within a thread
	Reflection attack in an auth protocol
	Capture-replay

Table A.6: CLASP Classification, attribute 3, Synchronization and Timing Errors [164]

Protocol Errors	Failure to follow chain of trust in certificate validation
	Key exchange without entity authentication
	Failure to validate host-specific certificate data
	Failure to validate certificate expiration
	Failure to check for certificate revocation
	Failure to encrypt data
	Failure to add integrity check value
	Failure to check integrity check value
	Use of hard-coded password
	Use of hard-coded cryptographic key
	Storing passwords in a recoverable format
	Trusting self-reported IP address
	Trusting self-reported DNS name
	Using referrer field for authentication
	Using a broken or risky cryptographic algorithm
	Using password systems
	Using single-factor authentication
	Not allowing password aging
	Allowing password aging
	Reusing a nonce key pair in encryption
	Using a key past its expiration date
	Not using a random IV with CBC mode
	Failure to protect stored data from modification
	Failure to provide confidentiality for stored data

Table A.7: CLASP Classification, attribute 4, Protocol Errors [164]

General Logic Errors	Ignored function return value
	Ignored function return value
	Missing parameter
	Misinterpreted function return value
	Uninitialized variable
	Duplicate key in associative list (alist)
	Deletion of data-structure sentinel
	Addition of data-structure sentinel
	Use of sizeof() on a pointer type
	Unintentional pointer scaling
	Improper pointer subtraction
	Using the wrong operator
	Assigning instead of comparing
	Comparing instead of assigning
	Incorrect black delimitation
	Omitted break statement
	Improper cleanup on thrown exception
	Uncaught exception
	Improper error handing
	Improper temp file opening
	Guessed or visible temporary file
	Failure to deallocate data
	Non-cryptographic PRING
	Failure to check whether privileges were dropped successfully

Table A.8: CLASP Classification, attribute 5, General Logic Errors [164]

Genesis	Intentional
	Inadvertent
Time of Introduction	During Development
	During Maintenance
	During Operation
Location	Software
	Hardware

Table A.9: Taxonomy of Computer Program Security Flaws [111]

Improper protection (initialization and enforcement)	Improper choice of initial protection domain
	Improper isolation of implementation detail
	Improper change
	Improper naming
	Improper deallocation or deletion
Improper validation	
Improper synchronization	Improper indivisibility
	Improper sequencing
Improper choice of operand or operation	

Table A.10: PLOVER Taxonomy (Preliminary List of Vulnerability Examples for Researchers) [36]

Incomplete Parameter Validation
Inconsistent Parameter Validation
Implicit Sharing of Privileged/Confidential Data
Asynchronous Validation/Inadequate Serialization
Inadequate Identification/Authentication/Authorization
Violable Prohibition/Limit
Exploitable Logic Error

Table A.11: RISOS Taxonomy [1]

Input Validation and representation	Buffer Overflow
	Command Injection
	Cross-site Scripting
	Format String
	HTTP Response Splitting
	Illegal Pointer Value
	Integer Overflow
	Log Forging
	Path Manipulation
	Process Control
	Resource Injection
	Setting Manipulation
	SQL Injection
	String Termination Error
	Struts:Duplicate Validation Forms
	Struts:Erroneous validate() Method
	Struts:Form Bean Does Not Extend Validation Class
	Struts:Form Field Without Validator
	Struts:Plug-in Framework Not In Use
	Struts:Unused Validation Form
	Struts:Unvalidated Action Form
	Struts:Validator Turned Off
	Struts:Validator Without Form Field
	Unsafe JNI
	Unsafe Reflection
	XML Validation

Table A.12: Seven Pernicious Kingdoms, attribute 1, input validation and representation [181]

API abuse	Often Misused:Privilege Management	CWEC:Often Misused:Privilege Management
	Often Misused:String Management	CWEC:Often Misused:String Management
	Unchecked Return Value	CWEC:Unchecked Return Value
	CWEC:API Abuse	
	Dangerous Functions	CWEC:Dangerous Functions
	Directory Restriction	CWEC:Directory Restriction
	Heap Inspection	CWEC:Heap Inspection
	J2EE Bad Practices:getConnection()	CWEC:J2EE Bad Practices:getConnection()
	J2EE Bad Practices:Sockets	CWEC:J2EE Bad Practices:Sockets
	Often Misused:Authentication	CWEC:Often Misused:Authentication
	Often Misused:Exception Handling	CWEC:Often Misused:Exception Handling
	Often Misused:Path Manipulation	CWEC:Often Misused:Path Manipulation

Table A.13: Seven Pernicious Kingdoms, attribute 2, API abuse [181]

Security features	CWEC:Security Features	
	OWASP:Insecure Storage	
	Insecure Randomness	CWEC :(RAND) Randomness and Predictability
	Least privilege Violation	CWEC:Least Privilege Violation
	Missing Access Control	OWASP:Broken Access Control
		CWEC:Missing Access Control
	Password Management	CWEC:Plaintext Storage
	Password Management:Empty Password in Configuration File	CWEC:Empty Password in Configuration File
	Password Management:Hard-Coded Password	CWEC:Hard-Coded Password
	Password Management:Password in Configuration File	CWEC:Password in Configuration File
	Password Management:Weak Cryptography	CWEC:Weak Cryptography for Passwords
	Privacy Violation	CWEC:Privacy Violation

Table A.14: Seven Pernicious Kingdoms, attribute 3, Security features [181]

Time and State	J2EE Bad Practices:System exit()
	J2EE Bad Practices:Threads
	Signal Handling Race Conditions
	CWEC:Time and State
	Deadlock
	Failure to Begin a New Session upon Authentication
	File Access Race Conditions:TOCTOU
	Insecure Temporary File

Table A.15: Seven Pernicious Kingdoms, attribute 4, Time and State [181]

Error Handling	Empty Catch Block	CWEC:UNCH-Unchecked Error Condition
	Overly-Broad Catch Block	CWEC:Overly-Broad Catch Block
	Overly-Broad Throws Declaration	CWEC:Overly-Broad Throws Declaration
	CWEC:Error Handling	
	OWASP:Improper Error Handling	
	Catch Null Pointer Exception	CWEC:Catch Null Pointer Exception

Table A.16: Seven Pernicious Kingdoms, attribute 5, Error Handling [181]

Code Quality	CWEC:Code Quality	
	OWASP:Denial of Service	
	Double Free	CWEC:Double Free
	Inconsistent Implementations	CWEC:Inconsistent Implementations
	Memory Leak	CWEC:MEMLEAK-Memory Leak
	Null Dereference	CWEC:Null Dereference
	Obsolete	CWEC:Obsolete
	Undefined Behavior	CWEC:Undefined Behavior
	Uninitialized Variable	CWEC:Uninitialized Variable
	Unreleased Resource	CWEC:RELEASE-Improper resource shudown or release
	Use After Free	CWEC:Use After Free

Table A.17: Seven Pernicious Kingdoms, attribute 6, Code Quality [181]

Encapsulation	CWEC:Encapsulation	
	Comparing Classes by Name	CWEC:Comparing Classes by Name
	Data Leaking Between Users	CWEC Data Leaking Between Users
	Leftover Debug Code	CWEC:Leftover Debug Code
	Mobile Code:Object Hijack	CWEC:Mobile Code:Object Hijack
	Mobile Code:Use of Inner Class	CWEC:Mobile Code:Use of Inner Class
	Mobile Code:Non-Final Public Field	CWEC:Mobile Code:Non-Final Public Field
	Private Array-Type Field Returned from a Public Method	CWEC:Private Array-Type Field Returned from a Public Method
	Public Data Assigned to Private Array-Typed Field	CWEC:Public Data Assigned to Private Array-Typed Field
	System Information Leak	CWEC:System Information Leak
	Trust Boundary Violation	CWEC:Trust Boundary Violation

Table A.18: Seven Pernicious Kingdoms, attribute 7,Encapsulation [181]

Environment	CWEC:Environment	
	OWASP:Insecure Configuration Management	
	ASP.NET Misconfiguration:Creating Debug Binary	CWEC:ASP.NET Misconfiguration:Creating Debug Binary
	ASP.NET Misconfiguration:Missing Custom Error Handling	CWEC:ASP.NET Misconfiguration:Missing Custom Error Handling
	ASP.NET Misconfiguration:Password in Configuration File	CWEC:ASP.NET Misconfiguration:Password in Configuration File
	Insecure Compiler Optimization	CWEC:Insecure Compiler Optimization
	J2EE Misconfiguration:Insecure Transport	CWEC:J2EE Misconfiguration:Insecure Transport
	J2EE Misconfiguration: Insufficient Session-ID Length	CWEC:J2EE Misconfiguration: Insufficient Session-ID Length
	J2EE Misconfiguration:Missing Error Handling	CWEC:J2EE Misconfiguration:Missing Error Handling
	J2EE Misconfiguration:Unsafe Bean Declaration	CWEC:J2EE Misconfiguration:Unsafe Bean Declaration
	J2EE Misconfiguration:Weak Access Permissions	CWEC:J2EE Misconfiguration:Weak Access Permissions

Table A.19: Seven Pernicious Kingdoms, attribute 7,Environment [181]

Authentication	Brute Force
	Insufficient Authentication
	Weak Password Recovery Validation
Authorization	Credential/Session Prediction
	Insufficient Authorization
	Insufficient Session Expiration
	Session Fixation
Client-side Attacks	Content Spoofing
	Cross-site Scripting
Command Execution	Buffer Overflow
	Format String Attack
	LDAP Injection
	OS Commanding
	SQL Injection
	SSI Injection
	XPath Injection
Information Disclosure	Directory Indexing
	Information Leakage
	Path Traversal
	Predictable Resource Location
Logical Attacks	Abuse of Functionality
	Denial of Service
	Insufficient Anti-automation
	Insufficient Process Validation

Table A.20: WASC Threat Classification [189]

Intentional	Malicious	Trapdoor	
		Logic/Time Bomb	
	Non-malicious	Covert Channel	Storage
			Timing
		Inconsistent access paths	
Inadvertent	Validation Error	Addressing Error	
		Poor parameter value check	
		Incorrect check positioning	
		Identification/Authentication Inadequate	
	Abstraction Error	Object reuse	
		Exposed Internal Representation	
	Asynchronous Flaws	Concurrency(including TOCTOU)	
		Aliasing	
	Subcomponent misuse/failure	Resource Leak	
		Responsibility Misunderstanding	
	Functionality Error	Error handling failure	
		Other security flaw	

Table A.21: Software Flaw Taxonomy [190]

Appendix B

Buffer Overflow Taxonomy

Write/Read

This attribute poses the question "Is the buffer access an illegal write or an illegal read?" While detecting illegal writes is probably of more interest in preventing buffer overflow exploits, it is possible that illegal reads could allow eavesdropping of information or could constitute one operation in a multi-step exploit. The possible values for the Write/Read attribute and examples are shown below:

Value	Description	Example
0	write	buf[10] = 'A'
1	read	c = buf[10]

Table B.1: Write/Read Attribute Values

Upper/Lower Bound

This attribute describes which buffer bound gets violated, the upper or the lower. While the term "buffer overflow" leads one to envision accessing beyond the upper bound of a buffer, it is equally possible to underflow a buffer, or access below its lower bound. Bellow, the values for the Upper/Lower Bound attribute and examples assuming a ten-byte buffer are listed.

Value	Description	Example
0	upper	buf[10]
1	lower	buf[-1]

Table B.2: Upper/Lower bound attribute values.

Data Type

The Data Type attribute, whose possible values and examples of which are shown below, describes the type of data stored in the buffer. Character buffers are often manipulated with unsafe string functions in C, and some tools may focus on detecting overflows of those buffers; buffers of all types may be overflowed, however, and should be analyzed.

Value	Description	Example
0	character	char buf[10];
1	integer	int buf[10];
2	floating point	float buf[10];
3	wide character	wchar_t buf[10];
4	pointer	char * buf[10];
5	unsigned integer	unsigned int buf[10];
6	unsigned character	unsigned char buf[10];

Table B.3: Data Type Attribute Values

Memory Location

The Memory Location attribute describes where the overflowed buffer resides. Non-static variables defined locally to a function are on the stack, while dynamically allocated buffers (e.g., those allocated by calling a malloc function) are on the heap. The data region holds initialized global or static variables, while the BSS region contains uninitialized global or static variables. Shared memory is typically allocated, mapped into and out of a program's address space, and released via operating system specific functions (e.g., shmget, shmat, shmdt, and shmctl on Linux). While a typical buffer overflow exploit may strive to overwrite a function return value on the stack, buffers in other locations have been exploited and should be considered as well. The stack is used to store local, fixed-size buffers, other local variables, function arguments, as well as the return addresses of functions, and some other state, including environment variables. The function calls rely on the stack. The heap stores dynammically allocated buffers (malloc(), calloc(), or realloc()).

Scope

The Scope attribute describes the difference between where the buffer is allocated and where is it overrun. The scope is the same if the buffer is allocated and overrun within the same function. Inter-procedural scope describes a buffer that is allocated in one

Value	Description	Example
0	on the stack	void function1() { char buf[10]; ...}

1	on the heap	void function1() {
		char * buf;
		buf = (char *)malloc(10*sizeof(char));
		...}

2	in data region	void function1() { static char buf[10] = "0123456789"; ...}
3	in BSS data	void function1() { static char buf[10]; ...}
4	in shared memory	-

Table B.4: Memory Location Attribute Values

function and overrun in another function within the same file. Global scope indicates that the buffer is allocated as a global variable, and is overrun in a function within the same file. The scope is inter-file/inter-procedural if the buffer is allocated in a function in one file, and overrun in a function in another file. Inter-file/global scope describes a buffer that is allocated as a global in one file, and overrun in a function in another file. Any scope other than the same may involve passing the buffer address as an argument to another function; in this case, the Alias of Buffer Address attribute must also be set accordingly.

Container

The Container attribute asks, "Is the buffer inside of a container?". Buffers may stand alone, or may be contained in arrays, structures, or unions. The buffer-containing structures and unions may be further contained in arrays. The ability of static analysis tools to detect overflows within containers (e.g., overrunning one array element into the next, or one structure field into the next) and beyond container boundaries (i.e., beyond the memory allocated for the container as a whole) may vary according to how the tools model these containers and their contents.

Pointer

The Pointer attribute indicates whether or not the buffer access uses a pointer dereference. Note that it is possible to use a pointer dereference with or without an array index; the Index Complexity attribute must be set accordingly. In order to know if

Value	Description	Example
0	same	void function1() { char buf[10]; buf[10] = 'A'; }

1	inter-
	procedural
		void function1() {
		char buf[10];
		function2(buf); }
		void function2(char * arg1)
		{
		arg1[10] = 'A'; }

2	global	static char buf[10]; void function1() { buf[10] = 'A'; }

3	inter-
	file/inter-
	procedural
		File 1:
		void function1() {
		char buf[10];
		function2(buf); }
		File 2:
		void function2(char * arg1)
		{
		arg1[10] = 'A'; }

4	inter- file/global	File1: static char buf[10]; File 2: extern char buf[]; void function1() { buf[10] = 'A'; }

Table B.5: Scope Attribute Values

the memory location referred to by a dereferenced pointer is within buffer bounds, a code analysis tool must keep track of what pointers point to; this points-to analysis is a significant challenge (Landi, 1992).

Index Complexity

This attribute describes the complexity of the array index, if any, of the buffer access causing the overflow. Note that this attribute applies only to the user program, and is not used to describe how buffer accesses are performed inside C library functions (for which the source may not be readily available). Handling the variety of expressions that may be used for array indices is yet another challenge faced by code analysis tools.

Value	Description	Example
0	no	char buf[10];
1	array	char buf[5][10];
2	struct	typedef struct { char buf[10]; } my_struct;
3	union	typedef union { char buf[10]; int intval; } my_union;
4	array of structs	my_struct array_buf[5];
5	array of unions	my_union array_buf[5];

Table B.6: Container Attribute Values

Value	Description	Example
0	no	buf[10]
1	yes	*pBuf or (*pBuf)[10]

Table B.7: Pointer Attribute Values

Value	Description	Example
0	constant	buf[10]
1	variable	buf[i]
2	linear expression	buf[5*i + 2]
3	non-linear expression	buf[i%3] or buf[i*i]
4	function return value	buf[strlen(buf)]
5	array contents	buf[array[i]]
6	not applicable	*pbuf = 'A'

Table B.8: Index Complexity Attribute Values

Address Complexity

The Address Complexity attribute poses the question, "How complex is the address or pointer computation, if any, of the buffer being overflowed?" Again, this attribute

is used to describe the user program only, and is not applied to C library function internals. Just as with array indices, code analysis tools must be able to handle a wide variety of expressions with varying degrees of complexity in order to accurately determine if the address accessed is beyond allocated buffer boundaries.

Value	Description	Example
0	constant	buf[x], (buf+2)[x], (0x80097E34)[x], *(pBuf+2), strcpy(buf+2, src)
1	variable	(buf+i)[x], (bufAddrVar)[x], *(pBuf+i), strcpy(buf+i, src)
2	linear expression	(buf+(5*i + 2))[x], *(pBuf+(5*i + 2)), strcpy(buf+(5*i + 2), src)
3	non-linear expression	(buf+(i%3))[x], *(pBuf+(i*i)), strcpy(buf+(i%3), src)
4	function return value	(buf+f())[x], (getBufAddr())[x], *(pBuf+f()), *(getBufPtr()), strcpy(buf+f(), src), strcpy(getBufAddr(), src)
5	array contents	(buf+array[i])[x], (array[i])[x], *(pBuf+ array[i]), strcpy(buf+ array[i], src)

Table B.9: Address Complexity Attribute Values

Length Complexity

The Length Complexity attribute describes the complexity of the length or limit passed to the C library function, if any, that overflows the buffer. Note that if a C library function overflows the buffer, the overflow is by definition inter-file/inter-procedural in scope, and involves at least one alias of the buffer address. In this case, the Scope and Alias of Buffer Address attributes must be set accordingly. As with array index and buffer addresses, C programs may contain arbitrarily complex expressions for the lengths or limits passed in C library function calls. Code analysis tools must be able to handle these in order to accurately detect buffer overflows. In addition, the code analysis tools may need to provide their own wrappers for or models of C library functions in order to perform a complete analysis .

Alias of Buffer Address

This attribute indicates if the buffer is accessed directly or through one or two levels of aliasing. Assigning the original buffer address to a second variable and subsequently using the second variable to access the buffer constitutes one level of aliasing, as does passing the original buffer address to a second function. Similarly, assigning the second variable to a third and accessing the buffer through the third variable would be classified

Value	Description	Example
0	not applicable	buf[10] (no library function called)
1	none	strcpy(buf, src)
2	constant	strncpy(buf, src, 10)
3	variable	strncpy(buf, src, i)
4	linear expression	strncpy(buf, src, 5*i + 2)
5	non-linear expression	strncpy(buf, src, i%3)
6	function return value	strncpy(buf, src, getSize())
7	array contents	strncpy(buf, src, array[i])

Table B.10: Length Complexity Attribute Values

as two levels of aliasing, as would passing the buffer address to a third function from the second.

Value	Description	Example
0	no	char buf[10]; buf[10] = 'A';
1	one alias	char buf[10]; char * alias_one; alias_one = buf; alias_one[10] = 'A';

2	two aliases	char buf[10];
		char * alias_one;
		char * alias_two;
		alias_one = buf;
		alias_two = alias_one;
		alias_two[10] = 'A';

Table B.11: Alias of Buffer Address Attribute Values

Alias of Buffer Index

The Alias of Index attribute is similar to the Alias of Address attribute, but applies to the index, if any, used in the buffer access rather than the buffer address itself. This attribute indicates whether or not the index is aliased. If the index is a constant or the results of a computation or function call, or if the index is a variable to which is directly assigned a constant value or the results of a computation or function call, then there is no aliasing of the index. If, however, the index is a variable to which the value of a second variable is assigned, then there is one level of aliasing. Adding a third variable

assignment increases the level of aliasing to two. If no index is used in the buffer access, then this attribute is not applicable.

Value	Description	Example
0	no	int i = 10; buf[10] = 'A'; buf[i] = 'A';
1	one alias	int i, j; i = 10; j = i; buf[j] = 'A';
2	two aliases	int i, j, k; i = 10; j = i; k = j; buf[k] = 'A';
3	not applicable	char * ptr; ptr = buf + 10; *ptr = 'A';

Table B.12: Alias of Buffer Index

Local Control Flow

This attribute describes what kind of program control flow, if any, most immediately surrounds or affects the overflow. For the values "if", "switch", and "cond", the buffer overflow is located within the conditional construct. "Goto/label" signifies that the overflow occurs at or after the target label of a goto statement. Similarly, "setjmp/longjmp" means that the overflow is at or after a longjmp address. Buffer overflows that occur within functions reached via function pointers are assigned the "function pointer" value, and those within recursive functions receive the value "recursion". The values "function pointer" and "recursion" necessarily imply a global or interprocedural scope, and may involve an address alias. The Scope and Alias of Address attributes should be set accordingly. Control flow involves either branching or jumping to another context within the program; hence, only path-sensitive code analysis can determine whether or not the overflow is actually reachable (Xie et al., 2003). A code analysis tool must be able to follow function pointers and have techniques for handling recursive functions in order to detect buffer overflows with the last two values for this attribute.

Secondary Control Flow

The types of control flow described by the Secondary Control Flow attribute are the same as the types described by the Local Control Flow; the difference is the location of the buffer overflow with respect to the control flow construct. Secondary control flow either precedes the overflow, or contains nested local control flow that affects the

Value	Description	Example
0	none	buf[10] = ‘A’;
1	if	if (flag) { buf[10] = ‘A’; }
2	switch	switch (value) { case 1: buf[10] = ‘A’; break; ...}
3	cond	flag ? buf[10] = ‘A’ : 0;
4	goto/label	goto my_label; my_label: buf[10] = ‘A’;

5	setjmp/longjmp	if (setjmp(env) != 0) {
		...}
		buf[4105] = ’A’;
		longjmp(env, 1);

6	function pointer	In main:
		void (*fptr)(char *);
		char buf[10];
		fptr = function1;
		fptr(buf);
		void function1(char * arg1)
		{
		arg1[10] = ’A’; }

7	recursion	In main:
		function1(buf);
		void function1(char *arg1,
		int counter) {
		if (counter > 0) {
		function1(arg1, counter - 1);
		}
		arg1[10] = ‘A’;
		...}

Table B.13: Local Control Flow Attribute Values

overflow (e.g., nested if statements, or an if statement within a case of a switch). Only control flow that affects whether or not the overflow occurs is classified. In other words, a preceding if statement that has no bearing on the overflow is not labeled as any kind of secondary control flow. Some types of secondary control flow may occur without any local control flow, but some may not. If not, the Local Control Flow attribute should be set accordingly. Some code analysis tools perform path-sensitive analyses, and some

do not. Even those that do often must make simplifying approximations in order to keep the problem tractable and the solution scalable. This may mean throwing away some information, and thereby sacrificing precision, at points in the program where previous branches rejoin.

Loop Structure

The Loop Structure attribute describes the type of loop construct, if any, within which the overflow occurs. This taxonomy defines a "standard" loop as one that has an initialization, a loop exit test, and an increment or decrement of a loop variable, all in typical format and locations. A "non-standard" loop deviates from the standard loop in one or more of these three areas. Omitting the initialization and/or the increment when they're not applicable does not constitute a deviation from the standard. Non-standard loops may necessitate the introduction of secondary control flow (such as additional if statements). In these cases, the Secondary Control Flow attribute should be set accordingly. Any value other than "none" for this attribute requires that the Loop Complexity attribute be set to something other than "not applicable." Loops may execute for a large number or even an infinite number of iterations, or may have exit criteria that depend on runtime conditions; therefore, it may be impossible or impractical for static analysis tools to simulate or analyze loops to completion. Different tools have different methods for handling loops; for example, some may attempt to simulate a loop for a fixed number of iterations, while others may employ heuristics to recognize and handle common loop constructs. The approach taken will likely affect a tool's capabilities to detect overflows that occur within various loop structures.

Loop Complexity

The Loop Complexity attribute, indicates how many of the three loop components described under Loop Structure (i.e., init, test, and increment) are more complex than the following baseline: init: initializes to a constant test: tests against a constant increment: increments or decrements by one If the overflow does not occur within a loop, this attribute is not applicable. If none of the three loop components exceeds baseline complexity, the value assigned is "none." If one of the components is more complex, the appropriate value is "one," and so on. Of interest here is whether or not the tools handle loops with varying complexity in general, rather than which particular loop components are handled or not. For any value other than "not applicable," the Loop Structure attribute must be set to one of the standard or non-standard loop values.

Asynchrony

The Asynchrony attribute asks if the buffer overflow is potentially obfuscated by an asynchronous program construct. These functions are often operating system specific. A code analysis tool may need detailed, embedded knowledge of these constructs and

Value	Description	Example

0	none	if (feel_like_it) {
		do_something_unrelated();
		}
		buf[10] = 'A';

1	if	if (flag1) { flag2 ? buf[10] = 'A' : 0; }

2	switch	switch (value) {
		case 1:
		flag ? buf[10] = 'A' : 0;
		break;
		...}

3	cond	i = (j > 10) ? 10 : j; buf[i] = 'A';
4	goto/label	goto my_label; my_label: flag ? buf[10] = 'A' : 0;

5	setjmp/longmp	if (setjmp(env) != 0) {
		...}
		flag ? buf[10] = 'A' : 0;
		longjmp(env, 1);

6	function pointer	In main:
		void (*fptr)(char *);
		char buf[10];
		fptr = function1;
		fptr(buf);
		void function1(char * arg1)
		{
		flag ? arg1[10] = 'A' : 0; }

7	recursion	In main:
		function1(buf);
		void function1(char *arg1,
		int counter) {
		if (counter > 0) {
		function1(arg1, counter - 1);
		}
		flag ? arg1[10] = 'A' : 0;
		...}

Table B.14: Secondary Control Flow Attribute Values

the O/S-specific functions in order to properly detect overflows that occur only under these special circumstances.

Value	Description	Example
0	none	buf[10] = 'A';
1	standard for	for (i=0; i<11; i++) { buf[i] = 'A'; }

2	standard do-while	i=0;
		do {
		buf[i] = 'A';
		i++;
		} while (i<11);

3	standard while	i=0;
		while (i<11) {
		buf[i] = 'A';
		i++; }

4	non-standard for	i=0 for (; i<11; i++) { buf[i] = 'A'; }

5	non-standard do-while	i=0;
		do {
		buf[i] = 'A';
		i++;
		if (i>10) break;
		} while (1);

6	non-standard while	i=0;
		while (++i) {
		buf[i] = 'A';
		if (i>10) break; }

Table B.15: Loop Structure Attribute Values

Taint

The Taint attribute describes how a buffer overflow may be influenced externally. These functions may be operating system specific. The occurrence of a buffer overflow may depend on command line or stdin input from a user, the value of environment variables, file contents, data received through a socket or service, or properties of the process environment, such as the current working directory. All of these can be influenced by users external to the program, and are therefore considered "taintable." These may be the most crucial overflows to detect, as it is ultimately the ability of the external user to influence program operation that makes exploits possible. As with asynchronous constructs, code analysis tools may require detailed modeling of these functions in order to properly detect related overflows.

Value	Description	Example
0	not applicable	buf[10] = 'A';
1	none	for (i=0; i<11; i++) { buf[i] = 'A'; }

2	one	init = 0;
		for (i=init; i<11; i++) {
		buf[i] = 'A'; }

3	two	init = 0;
		test = 11;
		for (i=init; i<test; i++) {
		buf[i] = 'A'; }

4	three	init = 0;
		test = 11;
		inc = k - 10;
		for (i=init; i<test; i += inc)
		{
		buf[i] = 'A'; }

Table B.16: Loop Complexity Attribute Values

Value	Description	Example
0	no	n/a
1	threads	pthread_create, pthread_exit
2	forked process	fork, wait, exit
3	signal handler	signal

Table B.17: Asynchrony Attribute Values

Run-time Environment Dependence

This attribute indicates whether or not the occurrence of the overrun depends on something determined at runtime. If the overrun is certain to occur on every execution of the program, it is not dependent on the runtime environment; otherwise, it is. The examples discussed under the Taint attribute are examples of overflows that may be runtime dependent. Another example would be an overflow that may or may not occur, depending on the value of randomly generated number. Intuition suggests that it should be easier to detect overflows and avoid false alarms when runtime behavior is guaranteed to be the same for each execution.

Value	Description	Example
0	no	n/a
1	argc/argv	using values from argv
2	environment variables	getenv
3	file read (or stdin)	fgets, fread, read
4	socket/service	recv
5	process environment	getcwd

Table B.18: Taint Attribute Values

Value	Description
0	no
1	yes

Table B.19: Runtime Environment Dependence Attribute Values

Magnitude

The Magnitude attribute indicates the size of the overflow. "None" indicates that there is no overflow; it is used when classifying patched programs that correspond to bad programs containing overflows. The remaining values indicate how many bytes of memory the program will attempt to write outside the allocated buffer. One would expect static analysis tools to detect buffer overflows without regard to the size of the overflow, unless they contain an off-by-one error in their modeling of library functions. The same is not true of dynamic analysis tools that use runtime instrumentation to detect memory violations; different methods may be sensitive to different sizes of overflows, which may or may not breach page boundaries, etc. The various overflow sizes were chosen with future dynamic tool evaluations in mind. Overflows of one byte test both the accuracy of static analysis modeling, and the sensitivity of dynamic instrumentation. Eight and 4096 byte overflows are aimed more exclusively at dynamic tool testing, and are designed to cross word-aligned and page boundaries.

Value	Description	Example
0	none	buf[9] = 'A';
1	1 byte	buf[10] = 'A';
2	8 bytes	buf[17] = 'A';
3	4096 bytes	buf[4105] = 'A';

Table B.20: Magnitude Attribute Values

Continuous/Discrete

This attribute indicates whether the buffer overflow is continuous or discrete. A continuous overflow accesses consecutive elements within the buffer before overflowing past the bounds, whereas a discrete overflow jumps directly out of the buffer. Loop constructs are likely candidates for containing continuous overflows. C library functions that overflow a buffer while copying memory or string contents into it demonstrate continuous overflows. An overflow labeled as continuous should have the loop-related attributes or the Length Complexity attribute (indicating the complexity of the length or limit passed to a C library function) set accordingly. Some dynamic techniques rely on "canaries" at buffer boundaries to detect continuous overflows; tools that rely on such techniques may miss discrete overflows.

Value	Description	Example
0	discrete	buf[10] = 'A';
1	continuous	for (i=0; i<11; i++) { buf[i] = 'A'; }

Table B.21: Continuous/Discrete Attribute Values

Signed/Unsigned Mismatch

This attribute indicates if the buffer overflow is caused by a signed vs. unsigned type mismatch. Typically, a signed value is used where an unsigned value is expected, and gets interpreted as a very large unsigned or positive value. For instance, the second example below shows a size being calculated and passed to the memcpy function. Since the buffer is ten bytes long, the size calculated and passed to memcpy is negative one. The memcpy function, however, expects an unsigned value for the size parameter, and interprets the negative one as a huge positive number, causing an enormous buffer overflow. Several real exploits have been based on this type of overflow.

Value	Description	Example
0	no	memcpy(buf, src, 11);
1	yes	signed int size = 9 - sizeof(buf); memcpy(buf, src, size);

Table B.22: Signed/Unsigned Mismatch Attribute Values

Appendix C

Structure of the Standard of Good Practice

The complete structure of the Standard of Good Practice [69].

- NW1 Network management
  - NW1.1 Roles and responsibilities
  - NW1.2 Network design
  - NW1.3 Network resilience
  - NW1.4 Network documentation
  - NW1.5 Service providers
- NW2 Traffic management
  - NW2.1 Configuring network devices
  - NW2.2 Firewalls
  - NW2.3 External access
  - NW2.4 Wireless access
- NW3 Network operations
  - NW3.1 Network monitoring
  - NW3.2 Change management
  - NW3.3 Incident management
  - NW3.4 Physical security
  - NW3.5 Back-up
  - NW3.6 Service continuity
  - NW3.7 Remote maintenance

- NW4 Local security management
  - NW4.1 Local security co-ordination
  - NW4.2 Security awareness
  - NW4.3 Security classification
  - NW4.4 Information risk analysis
  - NW4.5 Security audit/review
- NW5 Voice networks
  - NW5.1 Voice network documentation
  - NW5.2 Resilience of voice networks
  - NW5.3 Special voice network controls
- SD1 Development management
  - SD1.1 Roles and responsibilities
  - SD1.2 Development methodology
  - SD1.3 Quality assurance
  - SD1.4 Development environments
- SD2 Local security management
  - SD2.1 Local security co-ordination
  - SD2.2 Security awareness
  - SD2.3 Security audit/review
- SD3 Business requirements
  - SD3.1 Specification of requirements
  - SD3.2 Confidentiality requirements
  - SD3.3 Integrity requirements
  - SD3.4 Availability requirements
  - SD3.5 Information risk analysis
- SD4 Design and build
  - SD4.1 System design
  - SD4.2 Application controls
  - SD4.3 General security controls
  - SD4.4 Acquisition

  - SD4.5 System build
  - SD4.6 Web-enabled development
- SD5 Testing
  - SD5.1 Testing process
  - SD5.2 Acceptance testing
- SD6 Implementation
  - SD6.1 System promotion criteria
  - SD6.2 Installation process
  - SD6.3 Post-implementation review

Bibliography

[1] R. Abbott, J. Chin, J. Donnelley, W. Konigsford, S. Tokubo, and D. Webb. Security analysis and enhancements of computer operating systems. Technical Report 20050809 335, April 1976 1976.

[2] Dennis M. Ahern, Aaron Clouse, and Richard Turner. *CMMI distilled : a practical introduction to integrated process improvement*. Addison-Wesley, Boston, 2001.

[3] One Aleph. Smashing the stack for fun and profit. http://www.phrack.org/archives/49/P49-14.

[4] Ian Alexander. Misuse cases: Use cases with hostile intent., 2003.

[5] Ross Anderson. Why information security is hard - an economic perspective. http://www.acsac.org/2001/abstracts/thu-1530-b-anderson.html; http://www.acsac.org/2001/papers/110.pdf, December 2001 2001.

[6] Ross Anderson and Tyler Moore. The economics of information security: A survey and open questions. http://www.cl.cam.ac.uk/ rja14/Papers/toulouse-summary.pdf, January 2007 2007.

[7] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *Security & Privacy Magazine, IEEE*, 3(1):84–87, 2005.

[8] Ashampoo. Ashampoo product categories. http://www2.ashampoo.com/webcache/html/1/prod_overview_2.htm, 2007.

[9] Taimur Aslam. A taxonomy of security faults in the unix operating system. http://ftp.cerias.purdue.edu/pub/papers/taimur-aslam/aslam-taxonomy-msthesis.pdf;, August 1995 1995.

[10] Daniel B. Welcome to the home to ossec. http://www.ossec.net/, August 2007 2007.

[11] Steve Ballmer, Craig Mundie, and Kevin Turner. Msft financial analyst meeting:executive. http://www.microsoft.com/msft/speech/FY06/ExecQAFAM2006.mspx, July 2006 2006.

[12] Arash Baratloo, Navjot Singh, and Timothy Tsai. Libsafe and libverify: Transparent run-time defense against stack smashing attacks. http://www.mirrors.au.wiretapped.net/security/host-security/libsafe/paper.html.

[13] BASE. Basic analysis and security engine (base). http://base.secureideas.net/, May 2007 2007.

[14] Terrorism Knowledge Base. Group profile, internet black tigers. http://www.tkb.org/Group.jsp?groupID=4062, 2007.

[15] BBC. Y2k: Overhyped and oversold? http://news.bbc.co.uk/2/hi/talking_point/586938.stm, January, 2000 2000.

[16] Labs Bell. Uno tool synopsis. http://spinroot.com/uno/, August 2007 2007.

[17] Fabrice Bellard. Tiny c compiler 0.9.23. http://webscripts.softpedia.com/script/Development-Scripts-js/Compilers/Tiny-C-Compiler-26892.html, 2007.

[18] benzedrine. Openbsd packet filter. http://www.benzedrine.cx/pf.html, May 2007 2007.

[19] Dirk Beyer, Thomas Hezinger, Rupak Majundar, and Ranjit Jhala. Mtc (models and theory of computation):blast project. http://mtc.epfl.ch/software-tools/blast/, August 2007 2007.

[20] Bindshell. Odysseus. http://www.bindshell.net/tools/odysseus, December 2006 2006.

[21] Matt Bishop. A taxonomy of unix system and network vulnerabilities. http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/ucd-ecs-95-10.pdf, May 1995 1995.

[22] Jakub Brecka and David Matousek. Sunbelt kerio personal firewall. http://www.sunbelt-software.com/Home-Home-Office/Sunbelt-Personal-Firewall/, 2007.

[23] Brightsight. List of evaluated products. http://www.commoncriteriaportal.org/public/consumer/index.php?menu=5, 2007.

[24] Pete Broadwell and Emil Ong. A comparison of static analysis and fault injection techniques for developing robust system services. www.cs.berkeley.edu/ emilong/research/saswifi.pdf.

[25] Frederick P. Brooks. No silver bullet-essence and accidents of software engineering.
http://www.virtualschool.edu/mon/SoftwareEngineering/BrooksNoSilverBullet.html, April 1987 1987.

[26] William Bush, Jonathan Pincus, and David Sielaff. A static analyzer for finding dynamic programming errors (prefix).
http://www.cs.umd.edu/class/fall2002/cmsc631/notes/Pincus.txt, November 2002 2002.

[27] Robert Cardona. Achilles-maven security consulting inc.
http://www.mavensecurity.com/achilles, 2006.

[28] Bruno Castro da Silva. Tuxguardian-an application based firewall.
http://tuxguardian.sourceforge.net/documentation.php, April 2006 2006.

[29] CERT. Overview of attack trends.
http://www.cert.org/archive/pdf/attack_trends.pdf;, 2002.

[30] Pravir Chandra, Jeremy Feragamo, Dan Graham, John Viega, Jeff Williams, and Alex Newman. Owasp clasp project v1.2.
http://www.owasp.org/index.php/OWASP_CLASP_Project.

[31] Hao Chen and David Wagner. Mops.
http://www.cs.berkeley.edu/ daw/mops/.

[32] B. Chess and G. McGraw. Static analysis for security. *Security & Privacy Magazine, IEEE*, 2:76–79, 2004. 6.

[33] Brian Chess. Eau claire.
http://www.vantuyl.com/chess/EauClaire/;
http://www.vantuyl.com/chess/;.

[34] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet security : repelling the wily hacker*. Addison-Wesley, Reading, Mass., 1994. William R. Cheswick, Steven M. Bellovin.; Includes bibliographical references (p. 257-276) and index.

[35] Andy C. Chou and Stanford University. Computer Science Dept. *Static analysis for bug finding in systems software*. 2003. Andy C. Chou.; xviii,169 leaves, bound; Submitted to the Department of Computer Science.; Copyright by the author.; Thesis (Ph. D.)–Stanford University, 2003.; Engler, Dawson.

[36] Steve Christey. Plover-preliminary list of vulnerability examples for researchers.
http://cve.mitre.org/docs/plover/plover.html, March 2006 2006.

[37] Steve Christey and Robert A. Martin. Vulnerability type distributions in cve, May 22, 2007 2007.

[38] Cigital. Its4:software security tool.
http://www.cigital.com/its4/, 2007.

[39] CIRT. Nikto.
http://www.cirt.net/code/nikto.shtml, February 2007 2007.

[40] Aleksandar Colovic. Pscan - linux process monitoring tool.
http://developer.novell.com/wiki/index.php/PScan_-
_Linux_process_monitoring_tool.

[41] IEEE Computer Society. Software Engineering Technical Committee. *IEEE standard glossary of software engineering terminology*. Institute of Electrical and Electronics Engineers, 1983.

[42] Condor. Using x.509 certificates for authentication.
http://www.cs.wisc.edu/condor/manual/v6.2/3_9Using_X_509.html, 2007.

[43] Microsoft Corporation. Prefast for drivers.
http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx, 2007.

[44] MITRE Corporation. Time and state.
http://cwe.mitre.org/data/definitions/361.html, May 2007 2007.

[45] PGP Corporation. Pgp corporation - hard disk encription.
http://www.pgp.com/, 2007.

[46] Coverity. Coverity incorporated: Products: Coverirty prevent sqs.
http://www.coverity.com/html/prod_prevent.html, 2007.

[47] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows:attacks and defenses for the vulnerability of the decade.
http://www.ece.cmu.edu/ adrian/630-f04/readings/cowan-vulnerability.pdf;
http://www.cse.ogi.edu/DISC/projects/immunix;, 2000.

[48] Martin Croxford and Roderick Chapman. Correctness by construction: A manifesto for high-integrity software.
http://www.stsc.hill.af.mil/crossTalk/2005/12/0512CroxfordChapman.html, December 2005.

[49] CyberCop. Cybercop.
http://www.cybercopportal.org/.

[50] Darknet. Pwdump.
http://www.darknet.org.uk/2006/10/download-pwdump-142-and-fgdump-134-windows-password-dumping/, 2007.

[51] N. Davis, W. Humphrey, S. T. Redwine Jr., G. Zibulski, and G. McGraw. Processes for producing secure software. *Security & Privacy Magazine, IEEE*, 02(3):18–25, 2004.

[52] Noopur Davis. Secure software development life cycle processes: A technology scouting report. Technical Report CMU/SEI-2005-TN-024, December 2005 2005. sponsored by the U.S. Department of Defense.

[53] DOROTHY E. DENNING. Cyberterrorism. In *Global Dialogue*. Global Dialogue, August 24, 2000 2000.

[54] Jack Dennon. Exploring diffie-hellman encryption. http://www.linuxjournal.com/article/6131, August 2002 2002.

[55] Public Education Department. Lites-science standars glossary. http://www.nmlites.org/standards/science/glossary_5.htm, 2007.

[56] descargar.mp3. Kaspersky internet security 6.0. http://descargar.mp3.es/lv/group/view/kl32371/Kaspersky_Internet_Security.htm.

[57] Information Technology Laboratory Software Diagnostics and Conformance Testing Division. Tool taxonomy, 25 Apr 2007. 2007.

[58] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities.* Addison Wesley Professional, 2006.

[59] SPI Dynamics. Web application and website security from spi dynamics. http://www.spidynamics.com/products/index.html, 2007.

[60] Freeware edition. Ida pro - freeware edition. http://www.securityfocus.com/tools/1923; http://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/IDA-PRO.shtml, 2007.

[61] eEye Digital. Retina- vulnerability assessment, vulnerability scanner, security assessment, network security scann. http://www.eeye.com/html/Products/Retina/index.html, 2007.

[62] Dawson Engler. Meta-level compilation. http://metacomp.stanford.edu/.

[63] David Ensor. Biggest u.s. spy agency choking on too much information. http://www.cnn.com/US/9911/25/nsa.woes/index.htmlB, November 25, 1999 1999.

[64] J. Epstein, S. Matsumoto, and G. McGraw. Software security and soa: danger, will robinson! *Security & Privacy Magazine, IEEE*, 4(1):80–83, 2006.

[65] eset. Nod32-antivirus software. http://www.eset.com/, 2007.

[66] Hiroaki Etoh. Propolice-gcc extension for protecting applications from stack-smashing attacks.
http://www.research.ibm.com/trl/projects/security/ssp/, August 2005 2005.

[67] Dan Farber. Sap acknowledges inappropriate downloads by tomorrownow in response to oracle suit.
http://blogs.zdnet.com/BTL/?p=5569&tag=nl.e622, July 2nd, 2007 2007.

[68] International Organization for Standardization (ISO). Iso/iec 27002:2005 information technology – security techniques – code of practice for information security management.
http://www.iso27001security.com/html/27002.html, 2007.

[69] Information Security Forum. About the standard. Technical report, January 2005 2005.

[70] James Foster, Vitaly Osipov, Nish Bhalla, and Niels Heinen. *Buffer Overflow Attacks. Detect, Exploit, Prevent.* Syngress Publishing, Inc., Hingham Street Rockland, MA 02370 United States of America., 2005.

[71] Jeff Foster. Cqual.
http://www.cs.umd.edu/ jfoster/cqual/;
http://sourceforge.net/projects/cqual/;, November 2004 2004.

[72] Ariel Futoransky, Luciano Notarfrancesco, Gerardo Richarte, and Carlos Sarraute. Building computer network attacks, March 31st, 2003 2003.

[73] Steve Gibson and Leo Laporte. Cross-site scripting.
http://www.grc.com/sn/SN-086.pdf, 2006.

[74] GoldSofts. Outpost firewall pro.
http://www.goldsofts.com/soft/821/41739/Outpost_Firewall_Pro.html.

[75] Mark G. Graff and Kenneth R. Van Wyk. *Secure Coding. Principles and Practices.* O'REILLY, 2003.

[76] Tim Greene. Businesses should pay more attention to software security.
http://www.networkworld.com/news/2006/022006-rsa-secure.html, February 2006 2006.

[77] Michael Greenwald, Carl A. Gunter, Bjorn Knutsson, Andre Scedrov, Jonathan M. Smith, and Steve Zdancewic. Computer security is not a science (but it should be).

[78] Roger Gustavsson. Buffer overflows. Technical Report 1, 2006-04-02 2006.

[79] Robert W. Hahn and Anne Layne-Farrar. The law and economics of software security.
http://aei-brookings.org/admin/authorpdfs/page.php?id=1266, 2006.

[80] Anthony Hall and Rod Chapman. Software engineering correctness by construction, 13th January 2004 2004.

[81] Welfe. Harald. Netfilter/iptables project. http://www.netfilter.org/, 2007.

[82] Van Hauser. Thc-hydra-fast and flexible network login hacker. http://freeworld.thc.org/thc-hydra/, May 2006 2006.

[83] Compuware Corporation Corporate Headquarters. Compuware devpartner securitychecker, an expert application security advisor. http://www.compuware.com/; www.compuware.com/products/devpartner/securitychecker.htm.

[84] Kipp E. B. Hickman. Ssl 2.0 protocol specification. http://wp.netscape.com/eng/security/SSL_2.html, 1995.

[85] Greg Hoglund and Gary McGraw. *Exploiting Software How to Break Code.* Addison-Wesley, 2004.

[86] P. Hope, G. McGraw, and A. I. Anton. Misuse and abuse cases: getting past the positive. *Security & Privacy Magazine, IEEE*, 02(3):90–92, 2004.

[87] John D. Howard and Thomas A. Longstaff. A common language for computer security incidents. Technical Report SAND98-8667, 1998.

[88] Michael Howard and David LeBlanc. *Writing Secure Code.* Microsoft Corporation, 2002.

[89] Michael Howard, David LeBlanc, and John Viega. *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them.* McGraw-Hill/Osborne, 2005.

[90] Michael Howard, Steve Lipner, and Inc Books24x7. The security development lifecycle, 2006.

[91] Michael Howard, Jon Pincus, and Jeanette M. Wing. *Measuring Relative Attacks Surfaces.*, chapter 8, page 110. 2003.

[92] Linda Ibrahim, Joe Jarzombek, and Matt Ashford. Integrity assurance: Extending the cmmi sm & icmm for safety and security. In *2nd Annual CMMI Technology Conference and User Group*, November 2002 2002.

[93] Linda Ibrahim, Joe Jarzombek, Matt Ashford, Roger Bate, Paul Croll, Mary Horn, Larry LaBruyere, and Curt Wells. Safety and security extensions for integrated capability maturity models. September 2004 2004.

[94] IETF. S/mime mail security (smime). http://www.ietf.org/html.charters/smime-charter.html, May 2007 2007.

[95] Immunity. Spike.
http://www.immunitysec.com/resources-freesoftware.shtml, 2004.

[96] Security Innovation. Static analysis tools. *December 2004*, 2004.

[97] Insecure. Nmap-free security scanner for network exploration & security audits.
http://insecure.org/nmap/.

[98] Insecure.org. Top 100 network security tools.
http://sectools.org/index.html;, 2006.

[99] ISO. Freely available standards.
http://isotc.iso.org/livelink/livelink/fetch/2000/2489/Ittf_Home/
PubliclyAvailableStandards.htm, 09-14-2007 2007.

[100] ISO/IEC. Common criteria, common methodology for information technology security evaluation, iso/iec 15408, September 2006 2006.

[101] Andrew Jaquith. *Security Metrics: Replacing Fear, Uncertainty, and Doubt.* March 2007 2007.

[102] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities.
http://www.auto.tuwien.ac.at/ chris/research/doc/oakland06_pixy.pdf, 2006.

[103] Mariam Kamkar and John Wilander. *A Comparison of Publicly Available Tools for Static Intrusion Prevention*, pages 68–84. Proceedings of the 7th Nordic Workshop on Secure IT Systems. Karlstad, Sweden, 2002.

[104] Robert Kaplan and David Norton. The balanced scorecard- measures that drive performance, 1992.

[105] Michelle Keeneym, Dawn Cappelli, Eileen Kowalski, Andrew Moore, Timothy Shimeall, and Stephanie Rogers. Secret service and cert release report analyzing acts of insider sabotage via computer systems in critical infrastructure sectors. Technical report, United States Secret Service, May 16, 2005 2005.

[106] T. J. Klevinsky, Scott Laliberte, and Ajay Gupta. *Hack IT Security through penetration testing.* Addison Wesley, February 2002.

[107] Charles J. Kolodgy. Penetration testing: Taking the guesswork out of vulnerability management, Jan 2007 2007.

[108] KP-Tools. Kp-tools-security group.
http://skeet.kp-tools.com/english.htm, 2007.

[109] Kendra June Kratkiewicz. Evaluating static analysis tools for detecting buffer overflows in c code.
www.ll.mit.edu/IST/pubs/KratkiewiczThesis.pdf, 2005.

[110] Ounce Labs. Software risk analysis: Know where your software is vulnerable. http://www.ouncelabs.com/solutions/solutions-software-portfolio-security.asp.

[111] Carl Landwehr, Alan Bull, Jhon Mcdermott, and Williams Choi. A taxonomy of computer program security flaws, with examples. http://chacs.nrl.navy.mil/publications/CHACS/1994/1994landwehr-acmcs.pdf;, September 1994 1994.

[112] Secure Programming Lint. Splint home page. http://splint.org/.

[113] Cheers Loni. Securityforest exploitation framework beta has been released! http://osdir.com/ml/security.penetration/2005-03/msg00115.html.

[114] Salvador Mandujano. A multiagent approach to outbound intrusion detection., December 2004 2004.

[115] McAfee. Mcafee:software antivirus. http://mcafee.com/, 2007.

[116] G. McGraw. Will openish source really improve security? pages 128–129, 2000.

[117] Nancy R. Mead. The common criteria. https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements/239.html?layoutType=plain, 2005.

[118] Phoram Mehta. Hailstorm - application security. http://searchsecurity.techtarget.com/magazineFeature/0,296894,sid14_gci1257211,00.html, 2007.

[119] P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. *Security & Privacy Magazine, IEEE*, 4:85–89, 2006. 6.

[120] C. C. Michael and Will Radosevich. Black box security testing tools, December 2005 2005.

[121] Christoph Michael and Steven R. Lavenhar. Source code analysis tools - overview, January 2006 2006.

[122] Microsoft. Introduction to windows firewall with advanced security. http://www.microsoft.com/downloads/details.aspx?familyid=DF192E1B-A92A-4075-9F69-C12B7C54B52B&displaylang=en, August 2006 2006.

[123] mister_x. Main[aircrack-ng]. http://www.aircrack-ng.org/doku.php, October 2007 2007.

[124] Massimiliano Montoro. Cain & abel. http://www.oxid.it/cain.html, 2006.

[125] msfdev. The metasploit project.
http://www.metasploit.com/, 2007.

[126] Steven Musil. Paris hilton's cell phone hacked?
http://news.com.com/Paris+Hiltons+cell+phone+hacked/2100-7349_3-5584691.html, February 21, 2005 2005.

[127] George Necula, Scott McPeak, Westley Weimer, Matthew Harren, and Jeremy Condit. Ccured documentation.
http://manju.cs.berkeley.edu/ccured/, January 2007 2007.

[128] netfilter. netfilter/iptables project homepage.
http://www.netfilter.org/projects/iptables/index.html, 2007.

[129] CNET Networks. Lavasoft personal firewall 2 - security and spyware.
http://www.cnet.com.au/downloads/0,239030384,10620347s,00.htm, 2007.

[130] CBS News. National security meltdown the largest spy agency falls behind., August 8, 2007 9:37am 2007.

[131] InfoSec News. Airsnort decryption tool.
http://seclists.org/isn/2001/Aug/0129.html, August 2001 2001.

[132] Stephen Northcutt, Jerry Shenk, Dave Shackleford, Tim Rosenberg, Raul Siles, and Steve Mancini. Penetration testing: Assessing your overall security before attackers do.
http://whitepapers.zdnet.com/whitepaper.aspx?&docid=278661&promo=100511, Jun 2006 2006.

[133] NT Objectives. Ntoinsight 2.0 - application security software.
http://www.ntobjectives.com/freeware/index.php.

[134] NT Objectives. Ntospider - application security software;.
http://www.ntobjectives.com/products/ntospider.php.

[135] Commuter Rail Division of the Regional Transportation Authority. Metra, northern league baseball, 2003.

[136] Openwall. John the ripper password cracker.
http://www.openwall.com/john/.

[137] Parasoft. C++ unit testing & code compliance: C++test-parasoft.
http://www.parasoft.com/jsp/products/home.jsp?product=CppTest&itemId=40, 2007.

[138] OWASP Project. Owasp webscarab project.
http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project, October 2007 2007.

[139] Samuel T. Redwine, Rusty O. Baldwin, Mary L. Polydys, Daniel P. Shoemaker, Jeffrey A. Ingalsbe, and Larry D. Wagoner. Software assurance: A guide to the common body of knowledge to produce, acquire, and sustain secure software. https://buildsecurityin.us-cert.gov/daisy/bsi/96/version/1/part/4/data/Secure, July 2006 2006.

[140] Samuel T. Redwine, Rusty O. Baldwin, Mary L. Polydys, Daniel P. Shoemaker, Jeffrey A. Ingalsbe, and Larry D. Wagoner. *Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software Version 1.0*. US Department of Homeland Security, Harrisonburg, Va., May 2006 2006.

[141] Darren Reed. Ip filter - tcp/ip firewall/nat software. http://coombs.anu.edu.au/ avalon/.

[142] Amy Rindskopf. Juvenile computer hacker cuts off faa tower at regional airport, first federal charges brought against a juvenile for computer crime. http://www.cybercrime.gov/juvenilepld.htm, March 18, 1998 1998.

[143] Paula Rooney. Is windows safer? http://www.crn.com/software/179103240, Feb. 10, 2006 2006.

[144] Joseph Roth and Victor Garza. Product guide: Internet security systems internet scanner 7.0. http://akamai.infoworld.com/Internet_Security_Systems_Internet_Scanner_7.0 /product_46438.html?view=1&curNodeId=0, 2007.

[145] Corporation Sandisk. Gramma tech: Products: Codesonar. http://www.grammatech.com/products/codesonar/overview.html, 2007.

[146] SecTools. Sectools-fragroute/fragrouter. http://secure2s.net/tools/2006/06/23/fragroutefragrouter/, June 2006 2006.

[147] SecuriTeam. L0phtcrack, the integrated password cracker for nt. http://www.securiteam.com/tools/2PUPRR5Q0K.html, January 1999 1999.

[148] Core Security. Core security. http://www.coresecurity.com/index.php5, 2007.

[149] RSA Security. Home-rsa, the security division of emc. http://www.rsa.com/, 2007.

[150] SSH Communications Security. Ssh comunications security. http://www.ssh.com/, 2007.

[151] Tenable Network Security. Nessus vulnerability scanner. http://www.nessus.org/nessus/;, 2007.

[152] Julian Seward. Valgrind 3.2.3.
http://webscripts.softpedia.com/script/Development-Scripts-js/Valgrind-26957.html, 2007.

[153] R. Shirey. Internet security glossary.
http://www.ietf.org/rfc/rfc2828.txt, May 2000 2000.

[154] Zhu Shuanglei. Project rainbowcrack.
http://www.antsight.com/zsl/rainbowcrack/, 2007.

[155] Peter Silberman and Richard Johnson. A comparison of buffer overflow prevention implementations and weaknesses.
www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf ;www.idefense.com, 2004.

[156] Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements by misuse cases. In *Proc. 37th Int. Conf. Technology of Object-Oriented Languages and Systems*, 2000.

[157] SNORT. Snort-the facto standard for intrusion detection/prevention.
http://www.snort.org/, 2007.

[158] Sofpedia. Comodo firewall pro 3.0.9.229 beta.
http://www.softpedia.com/get/Security/Firewall/Comodo-Personal-Firewall.shtml, 2007.

[159] SOFTgo. Routix netcom 1.8-misc networking tools.
http://www.soft-go.com/view/Routix-NetCom_20490.html, 2006.

[160] Softonic. Panda internet security.
http://panda-internet-security.softonic.com/, September 2006 2006.

[161] Softpedia. Look n stop firewall download.
http://www.softpedia.com/progDownload/Look-n-Stop-Firewall-Download-1252.html, 2007.

[162] Fortify Software. Rats-rough auditing tool for security.
http://www.fortifysoftware.com/security-resources/rats.jsp, 2007.

[163] GFI Software. Gfi languard-vulnerability.
http://www.gfi.com/lannetscan/, 2007.

[164] Secure Software. Clasp-comprehensive lightweight application security process.
searchsoftwarequality.techtarget.com/searchAppSecurity/downloads/clasp_v20.pdf ;, 2006.

[165] Sunbelt Software. Showpass.
http://research.sunbelt-software.com/threatdisplay.aspx?name=ShowPass&threatid=29362, 2007.

[166] Tiny Software. Tiny personal firewall 2.14.
http://www.brothersoft.com/Utilities_Security_Tiny_Personal_Firewall_81.html.

[167] SourceForge. Pmd.
http://pmd.sourceforge.net/.

[168] William Stallings. *Cryptography and Network Security : principles and practice*. Prentice Hall, Upper Saddle River, N.J., 2006.

[169] Mark Stamp. *Information security : principles and practice*. Wiley, Hoboken, N.J., 2005. Mark Stamp.

[170] James Michael Steward. Ten ways hackers breach security.
http://images.globalknowledge.com/wwwimages/whitepaperpdf/WP_Steward_Hackers.pdf, 2007.

[171] Richard Stiennon. Lessons learned from biggest bank heist in history.
http://www.cioupdate.com/trends/article.php/3600126, 2006.

[172] SWiK. sguil-swik.
http://swik.net/sguil.

[173] Symactec. Norton-best computer protection.
http://www.symantec.com/norton/products/index.jsp, 2007.

[174] Lawrance Taylor. Dui blog: Bad drunk driving laws, false evidence and a fading constitution.
http://www.duiblog.com/2007/09/04/secret-breathalyzer-software-finally-revealed/, September 2007 2007.

[175] Indian Computer Emergency Response Team. Empanelment of it security auditing organisations, terms and conditions for empanelment.
www.cert-in.org.in/emp-terms-conditions.pdf, March 26, 2006 2006.

[176] Armorize Technologies. Vulnerability database, 2007.

[177] Chinotec Technologies. Parosproxy.org - web application security.
http://www.parosproxy.org/index.shtml, 2004.

[178] Kerio Technologies. Kerio technologies.
http://www.kerio.com/, 2007.

[179] Jay-Evan J. Tevis and John A. Hamilton Jr. Methods for the prevention, detection and removal of software security vulnerabilities, Apr 2004 2004.

[180] PC Tools. Pc tools-essential tools for your pc.
http://www.pctools.com, 2007.

[181] K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *Security & Privacy Magazine, IEEE*, 3(6):81–84, 2005.

[182] American Civil Liberties Union. Eavesdropping 101: What can the nsa do? www.aclu.org, 1/31/2006 2006.

[183] Vendicator. Stack shield. http://www.angelfire.com/sk/stackshield/.

[184] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: a static vulnerability scanner for c and c++ code. pages 257–267, 2000.

[185] John Viega. Security in the software development lifecycle, 15 Oct 2004 2004.

[186] John Viega and Gary McGraw. *Building secure software how to avoid security problems the right way*. Addison-Wesley, Boston, 2002. John Viega, Gary McGraw.; Includes bibliographical references and index.

[187] David Wagner. Boon-buffer overrun detection. http://www.cs.berkeley.edu/ daw/boon/;.

[188] watchfire. Appscan suite for web application security testing. http://www.watchfire.com/products/appscan/default.aspx, 2007.

[189] webappsec. Web application security consortium: Threat classification. http://www.webappsec.org/projects/threat/v1/WASC-TC-v1_0.pdf;, 2004.

[190] Sam Weber, Paul Karger, and Amit Paradkar. A software flaw taxonomy: Aiming tools at security. http://cwe.mitre.org/documents/sources/ASoftwareFlawTaxonomy-AimingToolsatSecurity

[191] WebmasterFree. Ca personal firewall. http://www.webmasterfree.com/CA_Personal_Firewall_d7637.html, 2007.

[192] WebSnapr. Codeassure. http://javatoolbox.com/tools/codeassure, 2007.

[193] K. Wei, M. Muthuprasanna, and Suraj Kothari. Preventing sql injection attacks in stored procedures. page 8, 2006. IS:.

[194] David Wheeler. Flawinder home page. http://www.dwheeler.com/flawfinder/.

[195] Brian Wichmann. Tool assurance for predictable execution. www.aitcnet.org/isai/_NextMeeting/_Last22 November 2006 2006.

[196] Wiki-Based. Chaperon - iterating.
http://www.iterating.com/products/Chaperon#datasheet.

[197] John Wilander and Mariam Kamkar. *A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention*, pages 149–162. Proceedings of the 10th Network and Distributed System Security Symposium. San Diego, CA, 2003.

[198] Mariusz Woloszyn. Immunix stakguard: Emsi's vulnerability.
http://community.corest.com/ juliano/emsi_vuln.html.

[199] Chris Wysopal, Lucas Nelson, Dino Dai Zovi, and Elfriede Dustin. The art of software security testing. 2007.

[200] Michal Zalewski. Delivering signals for fun and profit.
http://www.zone-h.org/files/22/signals.txt, 2001.

[201] Misha Zitser. Securing software: An evaluation of static source code analyzers.
www.dspace.mit.edu/bitstream/1721.1/18025/1/57225430.pdf, 2003.

[202] Zonelabs. Zonealarm.
http://zonealarm-pro-7.softonic.com/, March 2007 2007.