Security administration

Josep Jorba Esteve

PID_00148474

Universitat Oberta de Catalunya
www.uoc.edu

Index

Introduction

The technological leap from isolated desktop systems to current systems integrated into local networks and Internet has added a new difficulty to the administrator's usual tasks: controlling system security.

Security is a complex field, which combines analysis techniques with techniques for detecting or preventing potential attacks. Such as the analysis of "psychological" factors, in relation to the behaviour of system users or attackers' possible intentions.

The attacks can come from many sources and affect from a single application or service or user to all of them or even the entire computer system.

Potential attacks can change the systems' behaviour and even make them crash (disabling them), or give a false impression of security, which can be difficult to detect. We can come across authentication attacks (obtaining access through previously disabled programs or users), interceptions (redirecting or intercepting communication channels and the data circulating within them) or substitution (replacing programs, machines or users for others, without the changes being noticed).

Note

Absolute security does not exist. A false impression of security can be as damaging as not having any security. Security is a very dynamic field on which we need to keep our knowledge constantly updated.

We must bear in mind that it is impossible to achieve 100% security.

Security techniques are a double-edged sword that can easily give us a false impression of controlling the problem. Currently, security is an extensive and complex problem and, more importantly, it is also dynamic. We can never expect or say that security is guaranteed, but rather it will probably be one of the areas that the administrator will have to spend most time on and on which knowledge will have to be kept updated.

In this unit we will examine some of the types of attacks we can encounter, how we can verify and prevent parts of local security and network environments from being attacked. Furthermore, we will examine techniques for detecting intrusions and some basic tools that can help us to control security.

We should also mention that in this unit we can only introduce some of the aspects related to security nowadays. For any real thorough learning, we advise consulting the available bibliography, as well as the manuals for the products and tools we have covered.

1. Types and methods of attack

Computer security in administration terms can be understood as the process that allows the system's administrator to prevent and detect unauthorised use of the system. Preventive measures help to prevent attempts by unauthorised users (known as intruders) to access any part of the system. Detection helps to discover when these attempts where made or, if they are effective, to establish barriers so that intrusions are not repeated and so that the system can be recovered if breached.

Intruders (known also colloquially as hackers, crackers, 'attackers' or 'pirates') normally wish to obtain control over the system, whether to cause its malfunctioning, to corrupt the system or its data, to make use of the machine's resources or simply to use it to launch attacks on other systems, thus helping them to protect their own identity and hide the real source of the attacks. It is also possible that they wish to examine (or steal) the system's information, straightforward espionage of the system's actions or to cause physical damage to the machine, by formatting the disk, changing data, deleting or modifying critical software etc.

With regard to intruders, we need to establish some differences that are not very clear in colloquial terms. Normally, we refer to a hacker [Him01], as a person with detailed knowledge of computing, more or less passionate about programming and security issues and that normally, for no malevolent purpose uses their knowledge to protect themselves or third parties by entering networks to detect security failures and, in some cases, to test their abilities.

An example would be the GNU/Linux community, which owes a lot to its hackers, since the term hacker has to be understood as an expert in certain issues (rather than an intruder on security).

At the same time, we have crackers. This is where the term is used more or less negatively, towards those who use their knowledge in order to corrupt (or destroy) systems, whether for their own fame, for financial reasons, with the intention of causing damage or simply inconvenience; for reasons of technological espionage, acts of cyber-terrorism etc. Likewise, we talk of hacking or cracking, when we refer to techniques for studying, detecting and protecting security, or, on the contrary, techniques designed to cause damage by breaching systems' security.

Unfortunately, obtaining access to a system (whether it is unprotected or partially safe) is much easier than it would seem. Intruders constantly discover new vulnerabilities (sometimes know as 'holes' or exploits), that allow them to enter different layers of software. The ever-increasing complexity of soft-

ware (and hardware) makes it more and more difficult to test the security of computer systems in a reasonable manner. The common use of GNU/Linux on networks, whether via the Internet or private networks with TCP/IP technology such as intranets, makes us expose our systems, as victims, to security attacks. [Bur02][Fen02][Line]

The first thing we have to do is to break the myth of computer security: it simply does not exist. What we can achieve is a certain level of security that makes us feel safe within certain parameters. But as such, it is merely a perception of security and, like all perceptions, can be false so that we may only become aware at the last minute once our systems have already been affected. The logical conclusion is that computer security requires an important effort in terms of consistency, realism and learning on a practically daily basis.

We need to be capable of establishing security policies for our systems that allow us to prevent, identify and react against potential attacks. And to be aware that the feeling of security that we may have, is precisely no more than that: a feeling. Therefore, we must not neglect any implemented policies and we need to keep them up to date, as well as our knowledge of the issue.

Possible attacks are a constant threat to our systems and can compromise their functioning, as well as the data that we handle; We will always have to define a certain policy of security requirements for our systems and data. The threats we may suffer could affect the following aspects:

Note

Threats affect confidentiality, or the integrity or accessibility of our systems.

- **Confidentiality**: the information must only be accessible to authorised persons; we are answering the question: who will be able to access it?

- **Integrity**: the information must only be modified by authorised persons: what can be done with it?

- **Accessibility**: the information must be available for those who need it when they need it, on condition that they are authorised: how and when can it be accessed?

Let's move on to a certain (non-exhaustive) classification of the usual types of attacks that we can suffer:

- **Authentication**: attacks that falsify the identity of the participant so that access is obtained to programs or services that were initially out of bounds.

- **Interception** (or tapping): mechanism whereby data is intercepted by third parties to whom the data was not directed.

- **Falsification** (or replacement): replacement of some participants – whether machines, software or data – by other false ones.

- **Theft** of resources: unauthorised use of our resources.

- Or, simply, **vandalism:** after all, the presence of mechanisms that allow interference with the correct functioning of the system or services to cause partial inconvenience or the shutdown or cancellation of resources is fairly common.

The methods and precise techniques employed can vary enormously (moreover, innovations arise everyday), obliging us, as administrators to be in constant contact with the field of security to know what we may have to face on a daily basis.

For each of these types attacks, normally one or more methods of attack may be used, which in turn can provoke one or more types of attack.

With regards to where an attack occurs, we need to be clear what can be done or what the objective of the methods will be:

Note

Attacks may have the purpose of destroying, disabling or spying our components, whether hardware, software or communication systems.

- **Hardware**: in this respect, the threat is directly on accessibility, what will someone who has acces to the hardware be able to do? In this case, we will normally need "physical" measures, such as security controls for access to the premises where the machines are located in order to prevent problems of theft or damage to the equipment designed to erase their service. Confidentiality and integrity may also be compromised if physical acess to the machines allows some of their devices, such as disk drives, to be used, or if it allows booting of the machines or access to user accounts that may be open.

- **Software**: if accessibility is compromised during an attack, programs may be deleted or disabled, denying access. In the case of confidentiality, it can give rise to unauthorised copies of the software. In the case of integrity, the default functioning of the program could be altered, so that it fails in certain situations or so that it performs tasks in the interest of the attacker, or may simply compromise the integrity of program data: making them public, altering them or simply stealing them.

- **Data:** whether structured, such as in database services, or version management (such as cvs) or simple files. Attacks that threaten accessibility can destroy or eliminate them, thus denying access to them. In the case of confidentiality, we could be allowing unauthorised reading and the integrity would be affected when modifications are made or new data is created.

- **Communication channel** (on the network, for example): for the methods that affect accessibility, it can cause the destruction or elimination of messages and prevent access to the network. In confidentiality, reading and observation of the traffic of messages to or from the machine. And

with regards to integrity, any modification, delay, reordering, duplication or falsification of the incoming and/or outgoing messages.

1.1. Techniques used in the attacks

The methods used are various and can depend on an element (hardware or software) or the version of the element. Therefore, we need to maintain the software updated for security corrections that arise and to follow the instructions of the manufacturer or distributor in order to protect the element.

Despite this, there are normally always "fashionable" techniques or methods at any particular time. Some brief notes on today's attack techniques are:

Note

The methods used by attackers are extremely varied and evolve constantly in terms of the technological details that they use.

- **Bug exploits**: or exploitation of errors or exploits [CERb] [Ins][San], whether of a hardware, software, service, protocol or of the operating system itself (for example, in the kernel), and normally in a specific version of these. Normally, any computer element is more or less prone to errors in its design, or simply to things that have not been foreseen or taken into account. Periodically, holes are discovered (sometimes known as exploits, or simply bugs), which may be taken advantage of for breaching system security. Normally either generic attack techniques are used, such as the one explained as follows, or particular techniques for the affected element. Every affected element will have someone responsible – whether the manufacturer, developer, distributor or the GNU/Linux community – for producing new versions or patches to handle these problems. As administrators, we are responsible for being informed and maintaining a responsible policy of updates to avoid potential risks of attack. If there are no solutions available, we can also study the possibility of using alternatives for the element or disabling it until we find a solution.

- **Virus**: program normally annexed to others and that uses mechanisms of autocopy and transmission. It is common to annex viruses to executable programs, electronic mails, or to incorporate them into documents or programs that allow macros (not verified). They are perhaps the greatest security plague of the moment.
  GNU/Linux systems are protected almost completely against these mechanisms for several reasons: in executable programs, they have very limited access to the system, in particular to the user account. With the exception of the root user, where we have to be very careful with what it executes. Mail does not tend to use non-verified macros (contrary to Outlook and Visual Basic Script in Windows, which is an exploit for the entry of viruses), and in the case of the documents, we are in a similar situation, since they do not support non-verified macros or scripting languages (such as Visual Basic for Applications (VBA) in Microsoft Office).
  In any case, we will have to pay attention to what may happen in the future, since specific viruses for GNU/Linux could be created taking advan-

tage of some bugs or exploits. We must also take a look at mail systems, since although we may not generate viruses, we can transmit them; for example, if our system functions as a mail router, messages with a virus could come in and could then be sent on to others. Here we can implement virus detection and filtering policies. Another plague that could enter the category of viruses are spam messages, which although not usually used as attacking elements, can be considered problematic due to the virulence with which they appear, and the financial cost that they can entail (in loss of time and resources).

- **Worm**: normally this is a type of program that takes advantage of a system bug in order to execute code without a permission. They tend to be used to take advantage of the machine's resources, such as the use of the CPU, when it detects that the system is not functioning or is not in use or, with malicious intent, with the objective of stealing resources or to use them to stop or block the system. Transmission and copying techniques are also commonly used.

- **Trojan horse** (or 'Trojans'): useful programs that incorporate some functionality but hide other functionalities, which are the ones used to obtain information from the system or in order to compromise it. A particular case could be the one of the mobile type codes of web applications such as Java, JavaScript or ActiveX; these normally ask for consent to be executed (ActiveX in Windows), or have limited models of what they can do (Java, JavaScript). But like all software, they also have bugs and are an ideal method for transmitting Trojans.

- **Back door** (or trap door): method for accessing a hidden program that could be used to give access to the system or processed data without our knowledge. Other effects could be changing the system's configuration, or allowing viruses to be introduced. The mechanism employed could come included in some type of common software or in a Trojan.

- **Logic bombs**: program embedded in another program which checks when specific conditions occur (temporary, user actions etc.) to activate itself and perform unauthorised activities.

- **Keyloggers**: special program dedicated to hijacking the interactions with the user's keyboard and/or mouse. They may be individual programs or Trojans incorporated into other programs.
  Normally, they would need to be introduced in an open system to which there was access (although more and more frequently they can come incorporated in Trojans that are installed). The idea is to capture any introduction of keys, in such a way as to capture passwords (for example, for bank accounts), interaction with applications, visited websites, completed forms etc.

- **Scanner** (port scanning): rather than an attack, it represents a prior step consisting of gathering potential targets. Basically, it consists of using tools that allow the network to be examined in order to find machines with open ports, whether TCP, UDP or other protocols, which indicate the presence of certain services. For example, scanning machines looking for port 80 TCP, indicates the presence of web servers, from which we can obtain information about the server and the version used in order to take advantage of its known vulnerabilities.

- **Sniffers**: allows to capture packages circulating on a network. With the right tools we can analyse machines' behaviours: which are servers, clients, what protocols are used, and in many case obtaining passwords for insecure services. Initially, they were used a lot for capturing passwords of telnet, rsh, rcp, ftp... insecure services that should not be used (use the secure versions instead: ssh, scp, sftp). Sniffers (and scanners) are not necessarily an attack tool, since they can also serve for analysing our networks and detecting failures, or simply for analysing our own traffic. Normally, the techniques of both scanners and sniffers tend to be used by an intruder looking for the system's vulnerabilities whether to learn the data of an unknown system (scanners), or to analyse its internal interaction (sniffer).

- **Hijacking**: these are techniques that try to place a machine in such a way that it intercepts or reproduces the functioning of a service in another machine from which it has intercepted the communication. They tend to be common in cases of electronic mail, file or web transfers. For example, in the web case, a session may be captured and it will be possible to reproduce what the user is doing, pages visited, interaction with forms etc.

- **Buffer overflows**: fairly complex technique that takes advantage of the programming errors in the applications. The basic idea is to take advantage of overflows in application buffers, whether queues, arrays etc. If the limits are not controlled, an attacking program can generate a bigger message or data than expected and cause failures. For example, many C applications with poorly written buffers, in arrays, if we surpass the limit we can cause the program's code to be overwritten causing a malfunctioning or breakdown of the service or machine. Moreover, a more complex variant allows parts of program to be incorporated in the attack (C compiled or shell scripts), that may allow the execution of any code that the attacker wishes to introduce.

- **Denial of Service** ('DoS attack'): this type of attack causes the machine to crash or overloads one or more services, rendering them unusable. Another technique is DDoS (Distributed DoS), which is based on using a set of distributed machines in order to produce the attack or service overload. This type of attack tends to be solved with software updates, since normally all of the services that were not designed for a specific workload are affected and saturation is not controlled. DoS and DDoS attacks are commonly used in attacks on websites or DNS servers, which are affected by server vulnerabilities, for example, specific versions of Apache or BIND. Another aspect that is worth taking into account is that our system could also be used for DDoS type attacks, through control from a backdoor or a Trojan.

  A fairly simple example of this attack (DoS) is known as the SYN flood, which tries to generate TCP packages that open a connection, but then do nothing else with it, simply leaving it open; this spends system resources on data structures of the kernel, and network connection resources. If this attack is repeated hundreds or thousands of times, all of the resources can become occupied without being used, in such a way that when users wish to make use of the service, it is denied because the resources are occupied. Another case is known as mail bombing, or simply resending (normally with a false sender) until mail accounts are saturated, causing the mail system to crash or to become so slow that it is unusable. To some extent these attacks are fairly simple to carry out with the right tools and have no easy solution, since they take advantage of the internal functioning of protocols and services; in these cases we need to take measures of detection and subsequent control.

Web sites

SYN flood, see: http://www.cert.org/advisories/CA-1996-21.html

Problems associated to e-mail bombing amb spamming: http://www.cert.org/tech tips/email_bombing_spamming.html

- **Spoofing**: the techniques of spoofing encompass various methods (normally, very complex) of falsifying both information or the participants in a transmission (origin and/or destination). Some spoofing examples include:
  - IP spoofing, falsification of a machine, allowing false traffic to be generated or intercepting traffic that was directed to another machine. In combination with other attacks, it can even breach firewall protection.
  - ARP spoofing, complex technique (uses a DDoS), which tries to falsify source addresses and network recipients by means of attacking the machines' ARP caches, in such a way that the real addresses are replaced by others in various points of a network. This technique can breach all type of protections, including firewalls, but is not a simple technique.
  - E-mail is perhaps the simplest. It consists of generating false emails, in terms of both content and source address. For this type, techniques of the type known as social engineering are fairly common; these basically trick the user in a reasonable manner, a classical example are false emails from the system administrator or, for example, from the bank where we have our current account, stating that there have been prob-

Web site

See the case of Microsoft in: http://www.computerworld.com/softwaretopics/os/windows/story/ 0,10801,59099,00.html

lems with the accounts and that we have to send confidential information or the previous password in order to solve them, or asking the password to be changed for a specific one. Surprisingly, this technique (also known as phising) manages to deceive a considerable number of users. Even with (social engineering of) simple methods: a famous cracker commented that his preferred method was by telephone. As an example, we describe the case of a certification company (*Verisign*), for which the crackers obtained the Microsoft private software signature by just making a call on behalf of a company that said a problem had arisen and that they needed their key again. In summary, high levels of computer security can be overcome by a simple telephone call or by an email badly interpreted by a user.

- **SQL injection:** it is a technique aimed at databases and web servers in particular, which generally takes advantage of the incorrect programming of web forms, where the information provided has not been correctly controlled. It does not determine that the input information is of the correct type (strongly typified in relation to what is expected) or the type or literal characters that are introduced are not controlled. The technique takes advantage of the fact that the literals obtained by the forms (for example web, although the attacks can be sustained from any API that allows access to a database, for example php or perl) are used directly for making consultations (in SQL), which will attack a specific database (to which in principle there is no direct access). Normally, if there are vulnerabilities and poor form control, SQL code can be injected into the form, in such a way that it can make SQL consultations which provide the searched information. In drastic cases, security information could be obtained (database users and passwords), or even entire database tables, or else loss of information or intentional deletion of data. This technique in web environments in particular can be serious, due to the laws on the protection of the privacy of personal data which an attack of this nature can threaten. In this case, rather than an issue of system security, we are dealing with a problem of programming and control with strong typing of the data expected by the application, in addition to the appropriate control of knowledge of vulnerabilities present in the used software (database, web server, API like php, perl...).

- **Cross-side scripting** (or XSS): another problem associated to web environments and, in particular, to alterations of html code and/or scripts that a user can obtain by visualising a particular website, which can be altered dynamically. Generally errors when it comes to validating HTML code are taken advantage of (all navigators have problems with this, due to the definition of HTML itself, which allows reading of practically any HTML code however incorrect it is). In some cases, the use of vulnerabilities can be direct through scripts in the web page, but normally the navigators have good control of these. At the same time, indirectly there are techniques that allow script code to be inserted, either through access to the user's

cookies from the navigator, or by altering the process of redirecting from one web page to another. There are also techniques using frames, that can redirect the HTML code that is being viewed or directly hang the browser. In particular, web sites' search engines can be vulnerable, for allowing script code to be executed. In general, they are attacks with complex techniques, but designed to capture information such as cookies, which can be used for sessions, and thus allow a determined person to be substituted by redirecting websites or obtaining their information. Once more from the system's perspective, it is a question of the software in use. We need to control and know about vulnerabilities detected in navigators (and make the most of the resources that they offer in order to avoid these techniques) and control the use of software (search engines used, versions of the web server, and APIs used in developments).

Some basic general recommendations for security, could be:

- Controlling a problematic factor: users. One of the factors that can most affect security is the confidentiality of passwords, which is affected by users' behaviour; this facilitates actions within the system itself on the part of potential attackers. Most attacks tend to come from within the system, in other words, once the attacker has obtained access to the system.

- Users include those who are forgetful (or indiscreet) and forget their password on a frequent basis, mention it in conversation, write it down on a piece of paper left somewhere or stuck next to the desk or computer, or that simply lend it to other users or acquaintances. Another type of user uses predictable passwords, whether the same as their user id, national identity number, name of girlfriend, mother, dog etc., which with a minimum amount of information can be easily discovered. Another case is normal users with a certain amount of knowledge, who have valid passwords but we should always bear in mind that there are mechanisms capable of discovering them (cracking of passwords, sniffing, spoofing...). We need to establish a "culture" of security among users and, through the use of techniques, oblige them to change their passwords, without using typical words, for long passwords (of more than 2 or 3 characters) etc. Lately, many companies and institutions are implementing the technique of making a user sign a contract obliging the user not to disclose the password or to commit acts of vandalism or attacks from their accounts (although of course this does not prevent others from doing so through the user).

- Not to use or run programs with no guarantee of origin. Normally, distributors use signature verification mechanisms in order to verify that software packages are what they say, like for example md5 sums (command md5sum) or the use of GPG signatures [Hatd] (gpg command). The seller or distributor provides an md5 sum of their file (or CD image) and we can check its authenticity. Lately, signatures for both individual packages and

for package repositories are used in distributions as a mechanism to ensure the supplier's reliability.

- Not to use privileged users (like the root user) for the normal working of the machine; any program (or application) would have the permissions to access anywhere.

- Not to access remotely with privileged users' privileges or to run programs that could have privileges. Especially if we do not know or have not checked the system's security levels.

- Not to use elements when we do not know how they behave or to try to discover how they behave through repeated executions.

These measures may not be very productive but if we have not protected the system, we have no control over what can happen and, even so, nobody can guarantee that a malicious program cannot sneak in and breach security if we execute it with the right permissions. In other words, in general we need to be very careful with all type of activities related to access and the execution of more or less privileged tasks.

1.2. Countermeasures

With regard to the measures that can be taken against the types of attacks that occur, we can find some preventive measures and some measures for detecting what is happening to our systems.

Let's look at some of the types of measures that we could take in the sphere of intrusion prevention and detection (useful tools are mentioned, some of which we will examine later):

- **Password cracking:** in attacks of brute force designed to crack passwords, it is common to try and obtain access through repeated logins; if entry is obtained, the user's security has been compromised and the door is left open to other types of attacks, such as backdoor attacks or simply the destruction of the user's account. In order to prevent this type of attack, we need to reinforce the passwords policy, asking for a minimum length and regular changes of password. One thing we need to avoid is the use of common words in the passwords: many of these attacks are made using brute force, with a dictionary file (containing words in the user's language, common terms, slang etc.). This type of password will be the first to be broken. It can also be easy to obtain information on the victim, such as name, national identity number or address, and to use this data for testing a password. For all of the above, it is also not recommended to have passwords with national identity numbers, names (own or of relatives etc.), addresses etc. A good choice tends to be a password of between 6 and 8

characters at minimum with alphabetic and numerical contents in addition to a special character.

Even if the password has been well chosen, it may be unsafe if used for unsafe services. Therefore, it is recommended to reinforce the services using encryption techniques that protect passwords and messages. And, on the other hand, to prevent (or not use) any service that does not support encryption, and consequently that is susceptible of attack using methods, such as sniffers; among these, we could include telnet, FTP, rsh, rlogin services.

- **Bug exploits**: avoid having programs available that are not used, are old or are not updated (because they are obsolete). Apply the latest patches and updates that are available for both applications and the operating system. Test tools that detect vulnerabilities. Keep up to date with vulnerabilities as they are discovered.

Web sites

See patches for the operating system at: http://www.debian.org/security http://www.redhat.com//security http://fedoraproject.org/wiki/Security

- **Virus**: use antivirus mechanisms or programs, systems for filtering suspicious messages; avoid the execution of macros (which cannot be verified). We should not minimise the potential effects of viruses, every day they are perfected and technically it is possible to make simple viruses that can deactivate networks in a matter of minutes (we just have to look at some of the recent viruses in the world of Windows).

Web site

For vulnerabilities, a good tool is Nessus. To discover new vulnerabilities, see CERT in: http://www.cert.org/advisories/ (old site) and http://www.us-cert.gov/cas/techalerts/index.html.

- **Worm:** control the use of our machines or users outside of normal hours and control incoming and/or outgoing traffic.

- **Trojan horse** (or Trojans): regularly check the integrity of programs using sum or signature mechanisms. Detection of anomalous incoming or outgoing system traffic. Use firewalls to block suspicious traffic. A fairly dangerous version of trojans consist of rootkits (discussed below), which perform more than one function thanks to a varied set of tools. In order to verify integrity, we can use sum mechanisms like md5 or gpg, or tools that automate this process like Tripwire or AIDE.

- **Backdoor** (or trap door): we need to obtain certification that programs do not contain any type of undocumented hidden backdoor from software sellers or suppliers and, of course, only accept software from places that offer guarantees. When the software belongs to third parties or comes from sources that could have modified the original software, many manufacturers (or distributors) will integrate some type of software verification based on sum codes or digital signatures (md5 or gpg type) [Hatd]. Whenever these are available, it is useful to verify them before proceeding to install the software. We can also test the system intensively, before installing it as a production system.

  Another problem consists of software alteration a posteriori. In this case, systems of signatures or sums can also be useful for creating codes over already installed software so as to control that no changes are made to

vital software. Or backup copies, which we can make comparisons with in order to detect changes.

- **Logic bombs**: in this case, they tend to be hidden after activations through time or through user actions. We can verify that there are no non-interactive jobs introduced on the system of the crontab or at type and other processes (of the nohup type for example), which are periodically executed, or executed in the background for a long time (w commands, jobs). In any case, we could use preventive measures to prevent non-interactive jobs for users, or only allow them for users that need them.

- **Keyloggers and rootkits**: in this case there would be some intermediary process that would try to capture our pressing of keys and try to store them somewhere. We will have to examine situations where a strange process appears belonging to our user, or to detect if we have any file open with which we are not working directly (for example, lsof could be helpful, see man), or network connections, if we were dealing with a keylogger with external sending. To test a very basic functioning of a simple keylogger, we can see the system *script* command (see *script* man). In the other case, the rootkit (which also tends to include a keylogger) is usually a package of several programs with various techniques that allow the attacker, once inside an account, to use various elements such as a keylogger, backdoors, Trojans (replacing system commands) etc. in order to obtain information and entrance doors to the system, often accompanied by programs that clean the logs, in order to eliminate evidence of the intrusion. A particularly dangerous case is that of rootkits, that are used or come in the form of kernel modules, which allows them to act at the level of the kernel. In order to detect them, we will need to control that there is no external traffic travelling to a specific address. A useful tool for verifying rootkits is *chrootkit*.

Web site

We can find the chkrootkit tool in: http://www.chkrootkit.org

- **Scanner** (port scanning): scanners tend to be launched over one or more loop systems for scanning known ports in order to detect those that are left open and what services are functioning (and to obtain information on the versions of the services) that could be susceptible to attacks.

- **Sniffers:** avoid tapping and thus prevent the possibility of interceptions being inserted. One technique is the network's hardware construction, which can be divided into segments so that the traffic can only circulate through the zone that will be used, placing firewalls to join these segments to be able to control incoming and outgoing traffic. Use encryption techniques so that the messages cannot be read and interpreted by someone intercepting the network. For the case of both scanners and sniffers, we can use tools such as Whireshark [Wir] (formerly Ethereal) and Snort to check our network or, for port scanning, Nmap. Sniffers can be detected on the network by searching for machines in promiscuous Ethernet mode

(intercepting any circulating package); the network card only usually captures the traffic that goes towards it (or of the broadcast or multicast type).

- **Hijacking**: implement mechanisms for services encryption, requiring authentication, and if possible, regularly renewing authentication. Control incoming or outgoing traffic through the use of firewalls. Monitor the network in order to detect suspicious flows of traffic.

- **Buffer overflows**: they tend to be common as bugs or holes in the system, and tend to be resolved through software updates. In any case, through logs, we can observe strange situations of crashed services that should be functioning. We can also maximise the control of processes and access to resources in order to isolate the problem when it occurs in environments of controlled access, such as the one offered by SELinux (see further on in the module).

- **Denial of Service** ('DoS attack') and others, such as SYN flood, or mail bombing: take measures to block unnecessary traffic on our network (through the use of firewalls for example). With the services where it is possible, we will have to control buffer sizes, the number of clients to be attended, connection timeouts, service capacities etc.

- **Spoofing: a)** IP spoofing, **b)** ARP spoofing, **c)** electronic mail. These cases require strong service encryption, control through the use of firewalls, authentication mechanisms based on various aspects (for example, not based on the IP, if it could be compromised), mechanisms can be implemented that control established sessions based on several machine parameters at the same time (operating system, processor, IP, Ethernet address etc.). Also monitor DNS systems, ARP cachés, mail spools etc. in order to detect changes in the information that invalidate preceding ones.

- **Social engineering**: this is not an IT issue really, but we have to make sure that users do not make security worse. Appropriate measures such as increasing information or educating users and technicians about security: controlling which personnel will have access to critical security information and in what conditions they may cede it to others. A company's help and maintenance services can be a critical point: controlling who has security information and how it is used.

- In relation to end users, improving the culture of passwords, avoiding leaving them noted down anywhere where third parties can see them or simply disclosing them.

2. System security

In the face of potential attacks, we need to have mechanisms for preventing, detecting and recovering our systems.

For local prevention, we need to examine the different mechanisms of authentication and permissions for accessing the resources in order to define them correctly and be able to guarantee the confidentiality and integrity of our information. In this case, we will be protecting ourselves against attackers that have obtained access to our system or against hostile users who wish to overcome the restrictions imposed on the system.

In relation to network security, we need to guarantee that the resources that we offer (if we provide certain services) have the necessary parameters of confidentiality and that the services cannot be used by unauthorised third parties, meaning that a first step will be to control which of the offered services are the ones we really want, and that we are not offering other services that are uncontrolled at the same time. In the case of services of which we are clients, we will also have to ensure the mechanisms of authentication, in the sense that we access the right servers and that there are no cases of substitution of services or servers (normally fairly difficult to detect).

With regards to the applications and the services themselves, in addition to guaranteeing the right configuration of access levels using permissions and authentication of authorised users, we need to monitor the possible exploitation of software bugs. Any application, however well designed and implemented may have a more or less high number of errors that can be taken advantage of in order to overcome imposed restrictions using certain techniques. In this case, we enforce a policy of prevention that includes keeping the system updated as much as possible, so that we either update whenever there is a new correction or if, we are conservative, we maintain those versions that are the most stable in security terms. Normally, this means periodically checking several security sites in order to learn about the latest failures detected in the software and the vulnerabilities that stem from them that could expose our systems to local or network security failures.

3. Local security

Local security [Peñ] [Hatb] is basic for protecting the system [Deb][Hatc], since normally following a first attempt from the network, it is the second protection barrier before an attack that manages to obtain partial control of the machine. Also, most attacks end up using the system's internal resources.

Note

Various attacks, although they may come from the outside, are designed to obtain local access.

3.1. Bootloaders

With regards to local security, problems can already start with booting with the physical access that an intruder could gain to a machine.

One of the problems starts when the system boots. If the system can be booted from disk or CD, an attacker could access the data of a GNU/Linux (or also Windows) partition just by mounting the file system and placing themselves as root users without needing to have any password. In this case, we need to protect the system's boot from the BIOS, for example, by protecting the access with a password, so that booting from a CD is not allowed (for example through a Live CD or diskette). It is also reasonable to update the BIOS, since it can also have security failures. Plus, we need to be careful because many BIOS manufacturers offer additional known passwords (a sort of backdoor), meaning that we cannot depend exclusively on these measures.

The following step is to protect the boot loader, whether lilo or grub, so that the attacker is not able to modify the start up options of the kernel or directly modify the boot (in the case of grub). Either of the two can also be protected using passwords.

In grub, the file /sbin/grub-md5-crypt asks for the password and generates an associated md5 sum. Then, the obtained value is entered into /boot/grub/grub.conf. Under the *timeout* line, we introduce:

```
password --md5 sum-md5-calculated
```

For lilo we place, either a global password with:

```
password = <selected password>
```

or one in the partition that we want:

```
image = /boot/vmlinuz-version
    password = <selected password>
    restricted
```

In this case *restricted* also indicates that we will not be able to change the parameters passed onto the kernel from the command line. We need to take care to set the file /etc/lilo.conf as protected so that only the root user has read/write privileges (chmod 600).

Another issue related to booting is the possibility that someone with access to the keyboard could reinitiate the system because if they press CTRL+ALT+DEL, (in some distributions it is now disabled by default) they will cause the machine to shutdown. This behaviour is defined in /etc/inittab, with a line like:

```
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
```

If commented, this possibility of reinitiating will become deactivated. Or on the other hand, we can create a file /etc/shutdown.allow, which allows certain users to reinitiate.

3.2. Passwords and shadows

The typical passwords of the initial UNIX systems (and of the first versions of GNU/Linux) were encrypted using DES algorithms (but with small keys and a system call that was responsible for encrypting and decrypting, specifically crypt, see the man).

Normally, they were in the file /etc/passwd, in the second field, for example:

```
user:sndb565sadsd:...
```

But the problem lies in the fact that this file is legible by any user, meaning that an attacker could obtain the file and use an attack of brute force, until decrypting the passwords that the file contained, or use an attack of brute force with dictionaries.

The first step is to use the new files /etc/shadow, where the passwords are now saved. This file is only legible by the root user and by nobody else. In this case, in /etc/passwd an asterisk (*) appears where previously the encrypted password was. By default, current GNU/Linux distributions use passwords of the shadow type unless told not to use them.

A second step is to change the system of encrypting the passwords for one that is more complex and difficult to break. Now, both Fedora and Debian offer passwords by md5; we are usually allowed to choose the system at the time of the installation. We need to take care with md5 passwords, because if we use NIS, we could have a problem; otherwise, all clients and servers will use md5 for their passwords. Passwords can be recognised in /etc/shadow because they have a "1" prefix.

Other possible actions include obliging users to change password frequently (the *change* command can be useful), imposing restrictions on the size and content of the passwords, and validating them with dictionaries of common terms .

Regarding the tools, it is interesting to have a password cracker (i.e. a program for obtaining passwords), in order to check the real security situation with our users' accounts, and thus forcing change in the ones we detect to be insecure. Two of the ones most commonly used by administrators are *John the Ripper* and *crack*. They can also work with a dictionary, so it will be interesting to have some ASCII dictionary in English (can be found on the web). Another tool is "Slurpie", which can test several machines at the same time.

An issue that we always need to take into account is to run these tests on our systems. We must not forget that the administrators of other systems (or the access or ISP provider) will have intrusion detection systems enabled and that we could be denounced for attempts at intrusion, either before the competent authorities (computer crime units) or before our ISP so that they close down our access. We need to be very careful with the use of security tools, which are always on the edge of security or intrusion.

3.3. Suid and sticky bits

Another important problem affects some special permissions used on files or script.

The sticky bit is used especially on temporary directories, where we want in some (sometimes unrelated) groups for any user to be able to write, but only the owner of the directory to be able to delete, or the owner of the file that is within the directory. A classical example of this bit is the temporary directory /tmp. We need to make sure that there are no directories of this type, since they can allow anyone to write on them, so that we must check that there are no more than those that are purely necessary as temporaries. The bit is placed using (chmod +t dir), and can be removed with -t. In an *ls* command it will appear as a directory with drwxrwxrwt permissions (take note of the last t).

The bit setuid allows a user to execute (whether an executable or a shell script) with another user's permissions. In some cases this can be useful, but it is also potentially dangerous. This is the case, for example, of programs with setuid as root: a user, although without root permissions, can execute a program with setuid that could have internal root user permissions. This is very dangerous in the case of scripts, since they could be edited and modified to do anything. Therefore, we need to keep these programs controlled, and if setuid is not necessary, we need to eliminate it. The bit is placed using chmod +s, whether applying it to the owner (then it is called suid) or to the group (then it is called

bit sgid); it can be removed with -s. In the case of viewing with *ls* command, the file will appear with -rwSrw-rw (take note of the S), if it is only suid, in sgid the S would appear after the second w.

In the case of using chmod with octal notation, four numbers are used, where the last three are the classical rwxrwxrwx permissions (remember that we have to add in the number 4 for r, 2 w, and 1 for x), and the first has a value for every special permission that we want (which are added): 4 (for suid), 2 (sgid), and 1 (for sticky).

3.4. Enabling hosts

The system has several special configuration files that make it possible to enable the access of a number of hosts to some network services, but whose errors could later allow local security to come under attack. We can find:

- user .rhosts: allows a user to specify a number of machines (and users) that can use their account through "r" commands (rsh, rcp...) without having to enter the account's password. This is potentially dangerous, since a poor user configuration could allow entry to unwanted users or could allow an attacker (with access to the user account) to change the addresses in .rhosts in order to enter comfortably without any type of control. Normally, we should not allow these files to be created and we should even delete them completely and disable the "r" commands.

- /etc/hosts.equiv: this is exactly the same as with the .rhosts files but at the level of the machine, specifying what services, what users and what groups can access "r" commands without the need for password control. Also, an error such as putting a "+" on a line of that file allows access to "any" machine. Nowadays, this file does not usually exist either and there is always the alternative of the ssh service to "r".

- /etc/hosts.lpd: in the LPD printing system it was used to put the machines that could access the printing system. We need to be very careful, if we are not serving, to completely disable access to the system, and if we are serving, to restrict to a maximum the machines that really make use of it. Or try to change to a CUPS or LPRng system, which has far more control over the services. The LPD system is a common target of worm-type or buffer overflow attacks and several important bugs are documented. We need to be on the lookout if we use this system and the hosts.lpd file.

3.5. PAM modules

PAM modules [Peñ][Mor03] are a method that allows the administrator to control how the user authentication process is performed for certain applications. The applications need to have been created and linked to the PAM libraries.

Basically, PAM modules are a set of shared libraries that can be incorporated into applications as a method for controlling their user authentication. Also, the authentication method can be changed (by means of the PAM modules configuration), without having to change the application.

The PAM modules (libraries) tend to be in the /lib/security directory (in the form of dynamically loadable file objects). And the PAM configuration is present in the /etc/pam.d directory, where a PAM configuration file appears for every application that is using PAM modules. We find the authentication configuration of applications and services such as ssh, graphic login of X Window System, like xdm, gdm, kdm, xscreensaver... or, for example, the system login (entrance with username and password). Old PAM versions used a file (typically in /etc/pam.conf), where the PAM configuration was read if the */etc/pam.d* directory did not exist.

Web site

For further information, see "The Linux-PAM System Administrators' Guide": http://www.kernel.org/pub/linux/libs/pam/Linux-PAM-html

The typical line of these files (in /etc/pam.d) would have this format (if using /etc/pam.conf we would have to add the service to which it belongs as a first field):

```
module-type control-flag module-path arguments
```

which specifies:

a) module type: if it is a module that requires user authentication (*auth*), or has restricted access (*account*); things we need to do when the user enters or leaves (*session*); we the user has to update the password.

b) control flags: they specify whether it is *required*, a *requisite*, whether it is *sufficient* or whether it is *optional*. This is a syntax. There is another more up to date one that works in pairs of value and action.

c) the module path.

d) arguments passed onto the module (they depend on each module).

Because some services need several common configuration lines, it is possible to have operations for including common definitions for other services, we just have to add a line with:

```
@include service
```

A small example of the use of PAM modules (in a Debian distribution), could be their use in the login process (we have also listed the lines included from other services):

`auth`		`requisite pam_securetty.so`

auth		requisitepam_nologin.so
auth		requiredpam_env.so
auth		requiredpam_unix.so nullok
account	required	pam_unix.so
session	required	pam_unix.so
session	optional	pam_lastlog.so
session	optional	pam_motd.so
session	optional	pam_mail.so standard noenv
password	required	pam_unix.so nullok obscure min = 4 max = 8 md5

This specifies the PAM modules required to control user authentication during login. One of the modules, pam_unix.so, is the one that really verifies the user's password (looking at files password, shadow...).

Others control the session to see when the latest entry was or save when the user enters and leaves (for the lastlog command), there is also a module responsible for verifying whether the user has mail to read (authentication is also required) and another that controls that the password changes (if the user is obliged to do so with the first login) and that it has 4 to 8 letters, and that md5 can be used for encryption.

In this example we could improve user security: the *auth* and the passwords allow passwords of length nil: this is the module's nullok argument. This would allow having users with empty passwords (potential source of attacks). If we remove this argument, we no longer allow empty passwords for the login process. The same can be done in the password configuration file (in this case, the passwords change command), which also presents nullok. Another possible action is to increase the maximum size of the passwords in both files, for example, with max = 16.

3.6. System alterations

Another problem can be the alteration of basic system commands or configurations, through the introduction of Trojans, or backdoors, by merely introducing software that replaces or slightly modifies the behaviour of the system's software.

Web site

AusCert UNIX checklist:
http://www.auscert.org.au/5816

A typical case is the possibility of forcing the root to execute false system commands; for example, if the root were to include the "." in its variable PATH, this would allow commands to be executed from its current directory, which would enable the placing of files that replaced system commands and that would be executed as a priority before the system's commands. The same process can be done with a user, although because a user's permissions are more

limited, it may not affect the system as much, rather the security of the user itself. Another typical case is the one of false login screens, replacing the typical login process, password, with a false program that would store the entered passwords.

In the case of these alterations, it will be vital to enforce a policy of auditing changes, whether through a calculation of signatures or sums (gpg or md5), or using some type of control software such as Tripwire or AIDE. For Trojans we can have different types of detections or use tools such as *chkrootkit*, if these come from the installation of some known rootkit.

Web site

chkrootkit, see: http://www.chkrootkit.org

4. SELinux

Traditional security within the system has been based on discretionary access control (DAC) techniques, whereby normally each program has full control over the access to its resources. If a specific program (or the user permitting) decides to make an incorrect access (for example, leaving confidential data open, whether through negligence or malfunctioning). Therefore in DAC, a user has full control over the objects that belong to him or her and the programs he or she executes. The executed program will have the same permissions as the user who is executing it. Therefore, the system's security will depend on the applications that are being executed and on the vulnerabilities that these may have or on the malicious software that these may include,and will especially affect the objects (other programs, files or resources) to which the user has access. In the case of the root user, this would compromise the global security of the system.

On a separate note, mandatory access control (MAC) techniques, develop security policies (defined by the administrator) where the system has full control over the rights of access granted over each resource. For example, we can give access to files with permissions (of the Unix type), but, with MAC policies, we have extra control to determine explicity what files a process is allowed to access and what level of access we wish to grant. Contexts that specify in what situations an object can access another object are established.

SELinux [NSAb] is a MAC type component recently included in branch 2.6.x of the kernel, which the distributions are progressively incorporating: Fedora/ Red Hat have it enabled by default (although it is possible to change it during the installation) and it is an optional component in Debian.

SELinux implements MAC-type security policies, which allow more refined access permissions than traditional UNIX file permissions. For example, the administrator could allow data to be added to a log file, but not to rewrite or truncate it (techniques commonly used by attackers to erase their tracks). In another example, we could allow network programs to link to the port (or ports) they require, but deny access to other ports (for example, it could be a technique that helps to control certain Trojans or backdoors).

SELinux was developed by the US NSA agency with direct contributions from various companies for UNIX and free systems, such as Linux and BSD. It was freed in the year 2000 and since then it has been integrated in different GNU/Linux distributions.

Web sites

Some resources on SELinux:
http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/selinux-guide/
http://www.nsa.gov/research/selinux/index.shtml
http://fedoraproject.org/wiki/SELinux

In SELinux we have a domain-type model, where each process runs in a so-called security context and any resource (file, directory, socket etc.) has a type associated to it. There is a set of rules that indicates what actions can be performed in each context on each type. One of the advantages of this context-type model is that the policies defined can be analysed (there are tools) for determining what flows of information are allowed, for example, to detect various routes of attack, or whether the policy is sufficiently complete so as to cover all potential accesses.

It has what is known as the *SELinux policy database* which controls all aspects of SELinux. It determines what contexts each program can use to run and specifies what types of each context can be accessed.

In SELlinux, every system process has a context consisting of three parts: an identity, a role and a domain. The identity is the name of the user account or system_u for system processes or user_u if the user has no defined policies. The role determines what the associated contexts are. For example user_r is not allowed to have the context sysadm_t (main domain for the system administrator). Therefore a user_r with identity user_u cannot obtain a sysadm_t context in anyway. A security context is always specified by this set of values like:

root:sysadm_r:sysadm_t

is the context for the system administrator, defines its identity, role and security context.

For example, in a machine with SELinux activated (in this case a Fedora) we can see the -Z option of the ps of contexts associated to the processes:

 # ps ax –Z

LABEL	PID	TTY	STAT	TIME	COMMAND
system_u:system_r:init_t	1	?	Ss	0:00	init
system_u:system_r:kernel_t	2	?	S	0:00	[migration/0]
system_u:system_r:kernel_t	3	?	S	0:00	[ksoftirqd/0]
system_u:system_r:kernel_t	4	?	S	0:00	[watchdog/0]
system_u:system_r:kernel_t	5	?	S	0:00	[migration/1]
system_u:system_r:kernel_t	6	?	SN	0:00	[migration/1]

LABEL	PID	TTY	STAT	TIME	COMMAND
system_u:system_r:kernel_t	7	?	S	0:00	[watchdog/1]
system_u:system_r:syslogd_t	2564	?	Ss	0:00	syslogd -m 0
system_u:system_r:klogd_t	2567	?	Ss	0:00	klogd -x
system_u:system_r:irqbalance_t	2579	?	Ss	0:00	irqbalance
system_u:system_r:portmap_t	2608	?	Ss	0:00	portmap
system_u:system_r:rpcd_t	2629	?	Ss	0:00	rpc.statd
user_u:system_r:unconfined_t	4812	?	Ss	0:00	/usr/libexec/gconfd-2 5
user_u:system_r:unconfined_t	4858	?	Sl	0:00	gnome-terminal
user_u:system_r:unconfined_t	4861	?	S	0:00	gnome-pty-helper
user_u:system_r:unconfined_t	4862	pts/0	Ss	0:00	bash
user_u:system_r:unconfined_t	4920	pts/0	S	0:00	gedit
system_u:system_r:rpcd_t	4984	?	Ss	0:00	rpc.idmapd
system_u:system_r:gpm_t	5029	?	Ss	0:00	gpm -m /dev/input/mice -t exps2
user_u:system_r:unconfined_t	5184	pts/0	R+	0:00	ps ax -Z
user_u:system_r:unconfined_t	5185	pts/0	D+	0:00	Bash

and with ls using the -Z option we can see the contexts associated to files and directories:

```
# ls -Z
drwxr-xr-x josep josep user_u:object_r:user_home_t Desktop
drwxrwxr-x josep josep user_u:object_r:user_home_t proves
-rw-r--r-- josep josep user_u:object_r:user_home_t yum.conf
```

and from the console we can find out our current context with:

```
$ id -Z
user_u:system_r:unconfined_t
```

In relation to the functioning mode, SELinux presents two modes known as: *permissive* and *enforcing*. In *permissive*, unauthorised access is allowed, but is audited in the corresponding logs (normally directly over /var/log/messages or, depending on the distribution, with the use of audit in /var/log/audit/audit.log). In *enforcing* mode no type of access that is not allowed by the defined policies is permitted. We can also deactivate SELinux through its configuration file (normally in /etc/selinux/config), by setting SELINUX=disabled.

We need to take care with activating and deactivating SELinux, especially with the labelling of contexts in the files, since during periods of activation/deactivation labels could be lost or simply not made. Likewise, when backing up the file system, we need to make sure that the SELinux labels are preserved.

Another possible problem to be taken into account is the large number of security policy rules that can exist and that can cause limitations in terms of controlling the services. In the face of a specific type of malfunctioning, it is worth determining first that it is not precisely SELinux that is preventing functioning due to a too strict security limitation (see the section on SELinux criticism) or options that we did not expect to have activated (can require a change in the configuration of the Booleans as we will see).

In relation to the policy applied, SELinux supports two different types: *targered* and *strict*. In targered policy type, most processes operate without restrictions and only specific services (some daemons) are put into different security contexts that are confined to security policy. In strict policy type, all processes are assigned to security contexts and confined to defined policies, in such a way that any action is controlled by the defined policies. In principle, these are the two types of policies defined in general, but the specification is open to include more.

A special case of policy is the multilevel security (MLS), which is a multilevel policy of the strict type. The idea is to define different levels of security within the same policy, with security contexts having an additional field of access level associated to them. This type of security policy (like MLS) tends to be used in governmental and military organisations, where there are hierarchical structures with different levels of privileged information, levels of general access and different capabilities of action at each level. In order to obtain some security certifications, we need to have this type of security policy.

We can define what type of policy will be used in /etc/selinux/config, variable SELINUXTYPE. The corresponding policy and its configuration will normally be installed in the directories /etc/selinux/SELINUXTYPE/, for example, in the policy subdirectory we tend to find the binary file of the policy compiled (which is what is loaded in the kernel, when SELinux is initiated).

4.1. Architecture

SELinux architecture consists of the following components:

- Code at kernel level
- Shared SELinux library
- The security policy (the database)
- Tools

Let us take a look at a few considerations with regards to each component:

- The kernel code monitors the system's activity and ensures that the requested operations are authorised under the current SELinux security policy configuration, not allowing unauthorised operations and normally generating log entries of denied operations. The code is currently integrated in kernels 2.6.x and, in previous ones, is offered as a series of patches.

- Most SELinux utilities and components not directly related to the kernel use the shared library called libselinux1.so, which provides an API for interacting with SELinux.

- The security policy is what is integrated in the SELinux rules database. When the system starts up (with SELinux activated), it loads the binary policy file, which normally resides in /etc/security/selinux (although it can vary according to the distribution).
  The binary policy file is created on the basis of a compilation (via *make*) of the policy source files and some configuration files.
  Some distributions (such as Fedora) do not install the sources by default, which we can normally find in /etc/security/selinux/src/policy or in /etc/selinux.
  Normally, these sources consist of various groups of information:
  – The files related to the compilation, *makefile* and associated scripts.

  – Initial configuration files, associated users and roles.

  – Type-enforcement files, which contain most sentences of the policy language associated to a particular context. We need to take into account that these files are enormous, typically tens of thousands of lines, which means that we can encounter the problem of finding bugs or defining changes in policies.

  – And files that serve to label the contexts of the files and directories during loading or at specific moments.

- Tools: include commands used to administrate and use SELinux. Modified versions of standard Linux commands. And Tools for the analysis of policies and for development.

Let's look from this last section at the typical tools that we can generally find:

Some of the main commands:

Name	Use
chcon	Labels a specific file, or set of files with a specific context.

Name	Use
checkpolicy	Performs various actions related to policies, including compilation of policies to binary, typically invoking from the makefile operations.
getenforce	Generates a message with the SELinux mode (*permissive* or *enforcing*). Or deactivated if this is the case.
getsebool	Obtains the list of Booleans, in other words, the list of on/off options for every context associated to a service or general option of the system.
newrole	Allows the transition of a user from one role to another.
runn_init	Used in order to activate a service (*start, stop*), ensuring that it is performed in the same context as when it is started up automatically (with init).
setenforce	Changes the mode of SELinux, `0 permissive`, `1 enforcing`.
setfiles	Labels directories and subdirectories with the appropriate contexts, it is typically used in the initial SELinux configuration.
setstatus	Obtains the system status with SELinux.

Also, other common programs are modified to support SELinux such as:

- cron: modified to include the contexts for jobs in progress by cron.
- login: modified to place the initial security context for users when they log in the system.
- logrotate: modified to preserve the context of the logs when they have been compiled.
- pam: modified to place the user's initial context and to use the SELinux API and obtain privileged access to password information.
- ssh: modified to place the user's initial context when the user logs in the system.
- And various additional programs that modify /etc/passwd or /etc/shadow.

Also some distributions include tools for managing SELinux, such as the setools(-gui), which carry several tools for managing and analysing policies. As well as specific tools for controlling the contexts associated to the different services supported by SELinux in the distribution, for example the system-config-security level tool in Fedora has a section for configuring SELinux as we can see in the following figure:

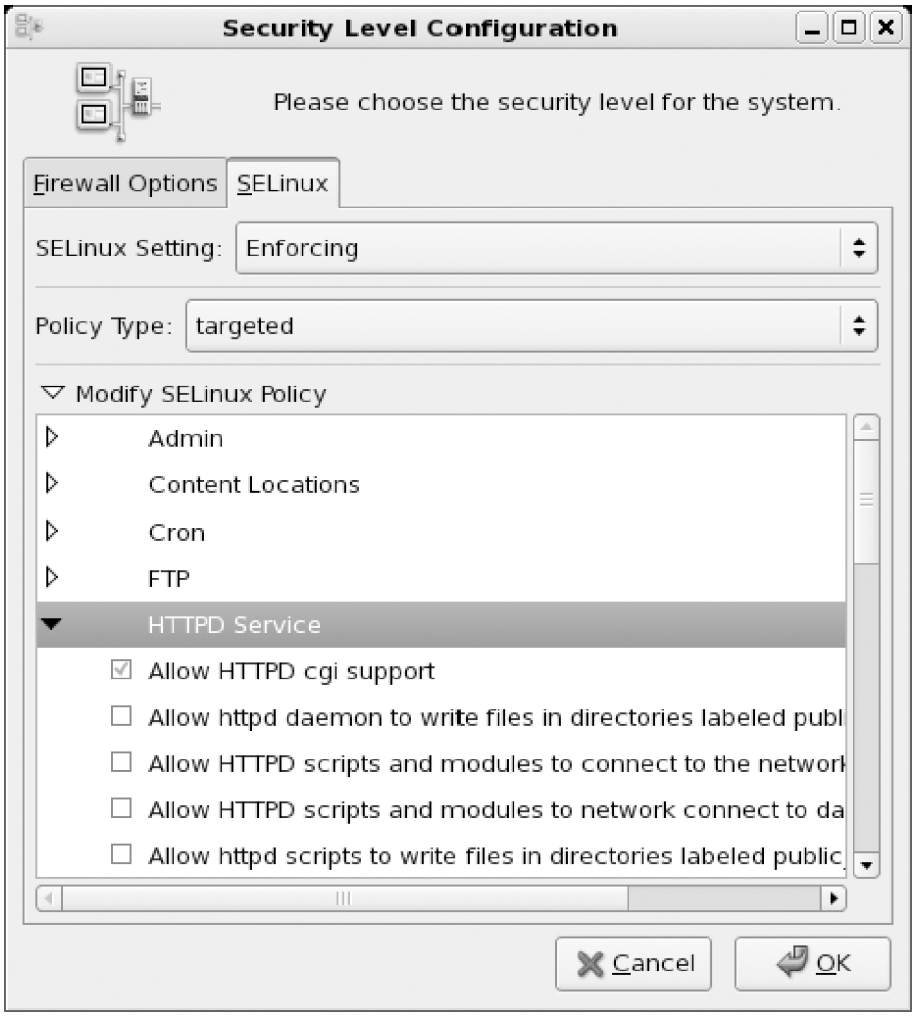

Figure 1. Interface in Fedora for configuring SELinux

In the figure, we can see the Booleans configuration for different services and generic options, including the web server. We can also obtain this list with the getsebool -a command and, with the setsebool/togglesebool command, we can activate/deactivate the options.

In Fedora, for example, we find Booleans support for (among others): Cron, FTP, httpd (apache), dns, grub, lilo, nfs, nis, cups, pam, ppd, samba, protections against undue access to the process memory etc.

The configuration of Booleans allows the SELinux policy to be tailored during running time. Booleans are used as conditional values of the applied policy rules, which allow policy modifications without having to load a new policy.

4.2. Criticism

Some administrators and security experts have criticised SELinux in particular for being too complex to configure and administer. It is argued that due to its intrinsic complexity, even experienced users can commit errors, leaving the SELinux configuration insecure or unusable and the system vulnerable. Although to a certain extent this is debatable, since even if we have SELinux

badly configured, the UNIX permissions would remain active, SELinux will not allow an operation that the original permissions already did not allow, in fact, we can see this as another stricter level of security.

Performance factors may also be affected, due to the enormous size of the policies, which use up a lot of memory and take a lot of time to load, and, in some cases, due to the processing of rules. We need to bear in mind that we are dealing with a system of practically 10,000 policy rules. And that this number can be even greater if we select the *strict* type policy where we need to specify absolutely all the options to be controlled. Normally, the processing of policy in binary format and the use of Booleans in order to disable rules allows the system to be used more efficiently.

Another aspect that tends to bother administrators is the additional problem of determining, in the event of a malfunction, what the origin or initial cause is. Because it is common for us to find in the end that the problem has stemmed from an excessively restrictive configuration (perhaps due to unawareness on the part of the administrator) of SELinux for a particular service.

In the last instance, we need to point out the extensive support that SELinux offers for security and that, as administrators, we need to be aware of the capabilities and dangers of any new technique that we employ.

5. Network security

5.1. Service client

As service clients, we basically need to make sure that we do not put our users in danger (or they put themselves in danger) by using insecure services. Avoid the use of services that do not use data encryption and passwords (FTP, telnet, non-secure mail). Use encrypted connection techniques, such as SSH and SSL.

Note

As service clients, we will need to avoid using insecure services.

Another important point concerns the potential substitution of servers for other false ones or session hijacking techniques. In these cases, we will need to have powerful authentication mechanisms that allow us to verify the servers' authenticity (for example, SSH and SSL have some of these mechanisms). And we will also have to verify the network searching for intruders who try to replace servers, as well as to apply correct package filtering services using firewalls, which allow us to remove our packages from a request and use the right servers, controlling the incoming packages that we receive as a response.

5.2. Server: inetd and xinetd

As we have seen, network services [Mou01] are configured from various places [Ano99][Hat01][Peñ]:

- In /etc/inetd.conf or the equivalent directory in /etc/xinetd.d: these systems are sort of "superservers" since they control subsidiary services and start up conditions. The inetd service is used in Debian and xinetd in Fedora (in Debian it can be installed as an option to replace inetd).

- Servers initiated during start up: depending on the runlevel we will have a number of servers initiated. The start up will originate in the directory associated to the runlevel. For example, in Debian, the default runlevel is 2, the services will be started up from the /etc/rc2.d directory, certainly with links to the scripts contained in /etc/init.d, which will run with the parameter start, stop, restart, as applicable.

- Other RPC type services: associated to remote calls between machines are used, for example in NIS and NFS. We can examine which ones with the rpcinfo -p command.

Other support files (with useful information) include: /etc/services, which consists of a list of known local or network services together with the protocol name, (tcp, udp or others), used for the service and the port that it uses; /etc/protocols is a list of known protocols; and /etc/rpc is a list of RPC servers

together with the used ports. These files come with the distribution and are a sort of database used by the network tools and commands in order to determine the name of services and their associated protocols or rpc and ports. We should mention that they are more or less historical files, which do not necessarily contain all the definitions of protocols and services; likewise we can search different Internet lists of known ports.

One of the administrator's first actions will be to disable all services that are not being used or that are not scheduled to be used, reading up on the use of services [Mou01] and what software may need them. [Neu]

In the case of /etc/inetd.conf, we just have to comment the service line that has to be disabled, by placing a number symbol (#) as the first character on the line.

In the other model of services, used by default in Fedora (and optionally in Debian), xinetd, the configuration lies in the /etc/xinetd.conf file, where some of the default values of log, control are configured and then the configuration of each subsidiary service is done through a file within the /etc/xinetd.d directory. In each file, the service information is defined, equivalent to what appears in the inetd.conf, in this case, to disable a service, we just have to enter the line "disable = yes" within the service file. Xinetd has a more flexible configuration than inetd, since it separates the configuration of the different services into different files and has a fair number of options for limiting connections to a service, their number or capabilities; all of which allows for a better control of the service and with the right configuration we can avoid some of the attacks by denying the service (DoS o DDoS).

With regard to the handling of runlevel services from the distribution's commands, we have already mentioned several tools that allow services to be enabled or disabled in the unit on local administration. There are also graphic tools such as *ksysv* of KDE, or the system-config-services and *ntsysv* in Fedora (in Debian, we recommend *sysv-rc-conf*, *rcconf* or *bum*). And at a lower level, we can go to the runlevel that we want (/etc/rcx.d) and deactivate the services we wish by changing the initial S or K of the script for other text: for example, one method would be: changing S20ssh, for STOP_S20ssh, and it will no longer start up; the next time we need it, we can remove the prefix and it will be active again. Or perhaps the recommended use of simple utilities to place, remove or activate a specific service (like service and *chkconfig* in Fedora or similar ones in Debian, such as *update-rc.d* and *invoke-rc.d*).

Another aspect is closing down insecure services. Traditionally, in the world of UNIX file transfer systems such as FTP were used with remote connection, such as telnet, and remote run commands (login or copy), many of which started with the letter "r" (for example, rsh, rcp, rexec...). Other potential dangers are *finger* and *rwhod* services, which allowed information to be obtained from the network of the machine users; here the danger lay in the information

that an attacker could obtain that would make the attacker's job easier. All of these services should not be used currently due to the potential dangers that they entail. In relation to the first group:

a) in network transmissions, ftp and telnet do not encrypt passwords and anyone can obtain pde service passwords or the associated accounts (for example, by using a sniffer).

b) rsh, rexec, rcp also have the problem that, under certain conditions, passwords are not even necessary (for example, if run from places validated in the .rhosts file), which means that once again they are insecure and leave the doors wide open to attacks.

The alternative is to use secure clients and servers that support message encryption and the authentication of participants. There are secure alternatives to the classical servers, but currently the most commonly used solution is the OpenSSH package (which can also be combined with OpenSSL for web environments). OpenSSH offers solutions based on the ssh, scp and sftp commands, allowing old clients and servers to be replaced (using a daemon called sshd). The ssh command allows the old functionalities of telnet, rlogin and rsh among others, scp would be the secure equivalent of rcp and sftp the equivalent of ftp.

With regards to SSH, we also have make sure we use ssh version 2. The first version has some known exploits; we need to take care when we install OpenSSH and, if we do not need the first version, install only the support for ssh2 (see the option Protocol in the /etc/ssh/ssh_config configuration file).

Besides, most services that we leave active on our machines would have to be filtered afterwards by a firewall to make sure that they are not used or attacked by people to whom they are not directed.

6. Intrusion detection

With intrusion detection systems [Hat01] (IDS) the aim is to take a step forward. Once we have been able to configure our security correctly, the next step will be to detect and actively prevent intrusions.

Note

IDS systems allow us to detect on time intruders using our resources or exploring our systems in search of security failures.

IDS systems create listening procedures and generate alerts when they detect suspicious situations, in other words, they look for the symptoms of potential security incidents.

We have systems based on local information, for example, gathering information from the system logs, monitoring changes in the file system or in the configurations of typical services. Other systems are based on the network and verify that there is no strange behaviour, such as spoofing, with the falsification of known addresses; controlling suspicious traffic, potential service denial attacks, detecting excessive traffic towards particular services, controlling that there are no network interfaces in promiscuous mode (a symptom of sniffers or package capturers).

Examples

Some examples of IDS tools: Logcheck (log verification), TripWire (system status through md5 sums applied to the files), AIDE (a free version of TripWire), Snort (IDS for verifying the status of an entire network).

7. Filter protection through wrappers and firewalls

TCP wrappers [Mou01] are programs that act as intermediaries between the requests of the users of a service and the daemons of the servers that provide the service. Most distributions already come with the *wrappers* activated and we configure the levels of access. The *wrappers* tend to be used in combination with inetd or xinetd, so as to protect the services that they offer.

The *wrapper* basically replaces the service's daemon for another that acts as an intermediary (called tcpd, normally in /usr/sbin/tcpd). When this receives a request, it verifies the user and the origin of the request, in order to determine whether the configuration of the service's *wrapper* allows it to be used or not. Also, it includes the ability to generate logs, or to inform via email possible attempts at access and then runs the appropriate daemon assigned to the service.

For example, let's assume the following entry in inetd:

```
finger stream tcp nowait nobody /usr/etc/in.fingerd
in.fingerd
```

We change it for:

```
finger stream tcp nowait nobody /usr/sbin/tcpd in.fingerd
```

so that when a request arrives, it is handled by the tcpd daemon which will be responsible for verifying the access (for more detailed information, see the tcpd man pages).

There is also an alternative method of TCP wrapper that consists of compiling the original application with the wrappers library. This way the application does not have to be in inetd and we can control it like in the first case with the configuration that we will discuss next.

Note

Wrappers allow us to control security through access lists to levels of services.

The wrappers system is controlled from the /etc/hosts.deny file, where we specify which services we deny to whom, using options, like a small shell to save the information on the attempt, and the /etc/hosts.allow file, where we place the service we intend to use, followed by the list of who is allowed to use the service (later, in the workshop, we will look at a small example). We also have the tcpdchk commands, which test the configuration of the hosts files (see man hosts_access and hosts_options) to check that they are

correct, in other words, it tests the configuration. The other useful command is tcpdmatch, to which we give the name of a service and a potential client (user, and/or *host*), and it tells us what the system will do in this situation.

7.1. Firewalls

A firewall is a system or group of systems that reinforces policies of access control between networks. The firewall can be implemented in software as a specialised application running on an individual computer or could be a special device designed to protect one or more computers.

In general, we will have either a firewall application to protect a specific machine directly connected to Internet (directly or through a provider), or we can place one or several machines designed for this function on our network in order to protect our internal network.

Technically, the best solution is to have one computer with two or more network cards that isolate the different connected networks (or network segments), in such a way that the firewall software on the machine (or if it is a special hardware) is responsible for connecting network packages and determining which can pass or not and to which network.

This type of firewall is normally combined with a router to link the packages of the different networks. Another typical configuration is the firewall towards the Internet, for example with two network cards: on one we obtain/provide traffic to the Internet and on the other we send or provide traffic to our internal network, thus eliminating traffic that is not addressed to us and also controlling traffic moving out towards the Internet, in case we do not wish to allow access to certain protocols or if we suspect that there are potential information leaks due to some attack. A third possibility is the individual machine connected with a single card towards the Internet, either directly or through a provider. In this case, we just want to protect our machine from intruders, unwanted traffic or traffic that is susceptible to data robbery.

In other words, in all these cases we can see that a firewall can have different configurations and uses depending on whether it is software or not, on whether the machine has one or several network cards or on whether it protects an individual machine or a network.

In general, the firewall allows the user to define a series of access policies (which machines can be connected to do or which machines can receive information and what type of information) by means of controlling the allowed incoming or outgoing TCP/UDP ports. Some firewalls come with preconfigured policies; in some cases they just ask whether we want a high, medium or low level of security; others allow all options to be tailored (machines, protocols, ports etc.).

Note

Firewalls make it possible to establish security at the level of packages and communication connections.

Another related technique is network address translation (NAT). This technique provides a route for hiding IP addresses used on the private network and hides them from the Internet, but maintains the access from the machines. One of the typical methods is the one known as masquerading. Using NAT masquerading, one or several network devices can appear as a single IP address seen from the outside. This allows several computers to be connected to a single external connection device; for example, the case of an ADSL router at home that allows several machines to be connected without the need for the provider to give us various IP addresses. ADSL routers often offer some form of NAT masquerading, and also firewall possibilities. It is fairly common to use a combination of both techniques. In this case, as well as the configuration of the firewall machine (in the cases we have seen above), the configuration of the internal private network that we want to protect also comes into play.

7.2. Netfilter: IPtables

The Linux kernel (as of versions 2.4.x) offers a filtering subsystem called Netfilter [Net], which offers package filtering features as well as NAT. This system allows different filter interfaces to be used, the most commonly used one is called IPtables. The main control command is *iptables*. Previously [Hata], it provided another filter called *ipchains* in kernels 2.2 [Gre], the system had a different (although similar) syntax. The 2.0 kernels used a different system called *ipfwadm*. Here (and in later examples) we will only deal with Netfilter/IPTables (in other words with the kernel versions 2.4/2.6).

Web site

Netfilter, ver: http://www.netfilter.org

Ipchains, see: http://www.netfilter.org/ipchains/

The interface of the IPtables command allows the different tasks to be perfomed for configuring the rules that affect the filter system: whether the generation of logs, pre and post package routing actions, NAT, and port forwarding.

Note

IPTables provides different elements such as the tables, chains and the rules themselves.

Service start up with: /etc/init.d/iptables start, if not already configured in the runlevel.

The iptables -L command lists the active rules at that time in each of the chains. If not previously configured, by default they tend to accept all the packages of the chains of input output and forward.

The IPTables system has the tables as a superior level. Each one contains different chains, which in turn contain different rules. The three tables that we have are: Filter, NAT and Mangled. The first is for the filtering norms themselves, the second is to translate addresses within a system that uses NAT and the third, less frequently used, serves to specify some package control options and how to manage them. Specifically, if we have a system directly connected to the Internet, we will generally only use the Filter table. If the system is on a private network that has to pass through a router, gateway or proxy (or a combination of them), we will almost certainly have a NAT or IP masquerading system; if we are configuring the machine to allow external access,

we will have to edit the NAT table and the Filter table. If the machine is on a private network system, but is one of the internal machines, it will be enough to edit the Filter table, unless it is a server that translates network addresses to another network segment.

If a package reaches the system, the firewall will first look at whether there are rules in the NAT table, in case addresses towards the internal network need to be translated (addresses are not normally visible outwards); then it will look at the rules in the Filter table in order to decide whether the packages will be allowed to pass or whether they are not for us and we have forward rules to know where to redirect them. On the contrary, when our processes generate packages, the output rules of the Filter table will control whether we allow them out or not, and if there is a NAT system the rules will translate the addresses in order to masquerade them. In the NAT table there are usually two chains: prerouting and postrouting. In the first, the rules have to decide if the package has to be routed and, if so, what the destination address will be. In the second, it is finally decided whether the package is allowed inside or not (to the private network, for example). And there is also an output chain for locally generated outgoing traffic to the private network, since prerouting does not control this (for more details, see iptables man page).

Next we will comment on some aspects and examples of configuring the Filter table (for the other tables, we can consult the associated bibliography).

The Filter table is typically configured as a series of rules that specify what is done inside a particular chain, like the three preceding ones (input, output and forward). Normally, we will specify:

```
iptables -A chain -j target
```

where *chain* is the input, output or forward and *target* is the destination that will be assigned to the packet which corresponds to the rule. Option -A adds the rule to the existing ones. We have to be careful here, because the order does matter. We have to put the least restrictive rules at the beginning, given that, if we put a rule that eliminates the packets at the beginning, even if there is another rule, this will not be taken into account. Option -j can be used to decide what we will do with the packets, typically *accept*, *reject* or *drop*. It is important to note the difference between *reject* and *drop*. With the first, we reject the packet and we will normally inform the sender that we have rejected the connection attempt (normally for an ICMP-type packet). With the second, *drop*, we simply "lose" the package as though it had never existed and we will not send any form of response. Another *target* that is used is *log*, to send the packet to the log system. Normally, in this case, there are two rules, one with the *log* and another identical one with *accept*, *drop* and *reject*, so that the information on the accepted, rejected or dropped packets can be sent to the log.

When entering the rule, we can also use the option -I (insert) to indicate a position, for example:

```
iptables -I INPUT 3 -s 10.0.0.0/8 -j ACCEPT
```

which tells us that the rule should be put in the third position in the *input* chain; and that packets (-j) that come from (with *source*, -s) from the subnet 10.0.0.0 with netmask 255.0.0.0 will be accepted. With -D, similarly, we can delete either a rule number or the exact rule, as specified below, deleting the first rule of the chain or the rule that we mention:

```
iptables -D INPUT 1
iptables -D INPUT -s 10.0.0.0/8 -j ACCEPT
```

There are also rules that can be used to define a default "policy" for the packets (option -P); the same thing will be done with all the packets. For example, we would usually decide to drop all the packets by default and then enable the ones that we require; likewise, we would often avoid forwarding packets if it is not necessary (if we do not act from the router), this could be declared as follows:

```
iptables -P INPUT DENY
iptables -P OUTPUT REJECT
iptables -P FORWARD REJECT
```

This establishes default policies that consist of rejecting any incoming packets and not permitting the sending or resending of packets. Now we will be able to add rules that affect the packets that we wish to use, stating which protocols, ports and origins or destinations we wish to permit or avoid. This can be difficult as we have to know all the ports and protocols that our software or services use. Another strategy would be to only leave active the services that are essential and to enable access to the services for the desired machine through the firewall.

Some examples of these rules on the Filter table could be:

```
1)iptables -A INPUT -s 10.0.0.0/8 -d 192.168.1.2 -j DROP
2)iptables -A INPUT -p tcp --dport 113 -j REJECT --reject-with
tcp-reset
3)iptables -I INPUT -p tcp --dport 113 -s 10.0.0.0/8 -j ACCEPT
```

where:

1) We drop the packets that come from 10.x.x.x sent to 192.168.1.2.

2) We reject the tcp packets sent to port 113, issuing a tcp-reset type response.

3) The same packets as in 2) but that come from 10.x.x.x will be accepted.

With regard to the names of the protocols and ports, the iptables system uses the information provided by the files /etc/services and /etc/protocols, and we can specify the information (port or protocol) either with numbers or with the names (we must make sure, in this case, that the information on the files is correct and that it has not been modified, for example, by an attacker).

The configuration of the iptables is usually established through consecutive calls to the iptables command with the rules. This creates a state of active rules that can be consulted with iptables -L; if we wish to save them so that they are permanent, we can do this in Fedora with:

```
/etc/init.d/iptables save
```

And they are saved in:

```
/etc/sysconfig/iptables
```

In Debian, we can execute:

```
/etc/init.d/iptables save name-rules
```

We have to be careful and ensure that the directory /var/log/iptables already exists, as this is where the files will be saved; name-rules will be a file in the directory.

With (/etc/init.d/iptables load) we can load the rules (in Debian, we have to provide the name of the rules file), although Debian supports some default file names, which are active for the normal rules (the ones that will be used when the service starts) and inactive for the ones that will remain when the service is deactivated (or stopped). Another similar method that is commonly used is that of putting the rules in a script file with the iptables calls that are necessary and calling them, for example, by putting them in the necessary runlevel, or with a link to the script in /etc/init.d.

7.3. Packets of firewalls in the distributions

With regard to the configuration tools that are more or less automatic in the firewall, there are various possibilities, but we should remember that they do not usually offer the same features as the manual configuration of iptables (which in most cases, is the recommended process). Some tools are:

- **lokkit:** in Fedora/Red Hat, on a very basic level, the user can only choose the desired security level (high, medium or low). Afterwards, the services that would be affected are shown and we can leave it, or not, so that we pass on to the service changing the default configuration. The mechanism

used beneath is the iptables. The final configuration of the rules that is made can be seen at /etc/sysconfig/iptables which, in turn, is read by the iptables service, which is loaded on boot up or when stopping or booting using /etc/init.d/iptables with the start or stop options. It is also possible to install it in Debian, but the rules configuration should be left in /etc/defaults/lokkit-l and a script in /etc/init.d/lokkit-l. There is also a graphic version called gnome-lokkit.

- **Bastille** [Proa]: this is a fairly complete and educational security program that explains different recommended security settings and how we can apply them step by step; it also explains the configuration of the firewall (the program is interactive). It works in various distributions, including in both Fedora and Debian.

- **fwbuilder**: a tool that can be used to build the rules of the firewall using a graphical interface. It can be used in various operating systems (GNU/Linux, both Fedora and Debian, OpenBSD, MacOS), with different types of firewalls (including iptables).

Web site

See: http://www.fwbuilder.org/

- **firestarter**: a graphical tool (Gnome) for creating a firewall. It is very complete, practically managing all the possibilities of the iptables, but, likewise, it has assistants that make it easy to set up a firewall intuitively. Likewise, there is a real-time monitor for detecting any intrusions.

Normally, each of these packets uses a rules system that is saved in its own configuration file and that usually starts up as a service or as a script execution in the default runlevel.

7.4. Final considerations

Even if we have well-configured firewalls, we have to remember that they are not an absolute security protection, as there are complex attacks that can pass over the firewalls or falsify the data to create confusion. In addition, modern connectivity sometimes needs to force us to create software that will bypass the firewalls:

Note

We should never rely on one single mechanism or security system. The security of the system must be established at all the different levels.

- Technologies, such as IPP, the printing protocol used by CUPS, or WebDAV, the authoring and versioning protocol for websites, make it possible to bypass (or make it necessary to bypass) the configurations of the firewalls.

- A technique called tunnelling is often used (for example, with the above-mentioned protocols and others). This technique basically encapsulates the non-permitted protocols, on the basis of others that are permitted; for example, if a firewall only permits HTTP traffic to pass (port 80 by default), it is possible to write a client and server (each one on one side of the firewall) that can speak in any protocol known to both, but in which

the network is transformed into a standard HTTP, which means that the traffic can bypass the firewall.

- The mobile codes by web (ActiveX, Java, y JavaScript) bypass the firewalls, and it is therefore difficult to protect the systems if these are vulnerable to attacks against any open holes that are discovered.

Therefore, although firewalls are a very good solution for most security-related aspects, they can always have vulnerabilities and let traffic that is considered valid through, which then includes other possible sources of attack or vulnerabilities. With regard to security, we should never consider (and rely on) only one single solution and expect it to protect us from everything; it is necessary to examine the various problems, to propose solutions that will detect any problems on time and to establish prevention policies that will protect the system before any harm is done.

8. Security tools

Some of these tools can also be considered tools for attacking other machines. Therefore, it is advisable to test these tools on machines in our own local or private network; we should never do this with third party IPs, as these could interpret the tests as intrusions and we or our ISP may be held responsible for them and the corresponding authorities may be notified to investigate us and remove our access.

We will now briefly discuss some tools and the ways in which they can be used:

a) TripWire: this tool maintains a database of sums for checking the important files in the system.

It may serve as a preventive IDS system. We can use it to "take" a snapshot of the system, so that we can subsequently check any modification made and that it has not been corrupted by an attacker. The aim here is to protect the files in the machine itself and to avoid any changes occurring, such as those that, for example, the rootkit might have caused. Therefore, when we execute the tool again, we can check all the changes compared to the previous execution. We have to choose a subset of files that are important in the system or possible sources of attack. TripWire is proprietary, but there is a free open-source tool that is the equivalent called AIDE.

b) Nmap [Insb]: this is a tool that scans ports in large networks. It can scan from individual machines to network segments. It provides various scanning modes, depending on the system's protections. It also provides techniques with which we can determine the operating system used by remote machines. Different TCP and UDP packets may be used to test the connections. There is a graphical interface known as xnmap.

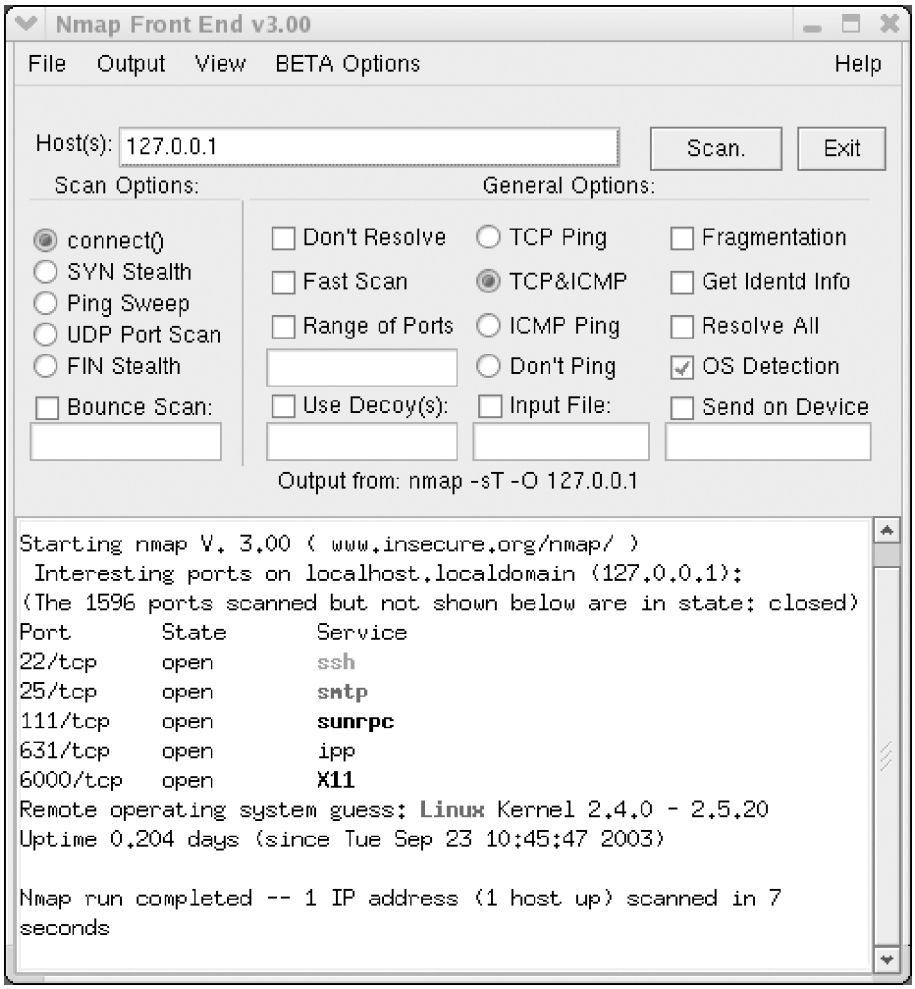

Figure 2. xnmap analysing the local services

c) Wireshark [Wir] (previously called **Ethereal**): is a protocol analyser that captures the traffic in the network (it acts as a sniffer). It can be used to visualise the captured traffic, see the statistics and data of the individual packets and group the packets, either by origin, destination, ports or protocol. It can even reconstruct the traffic from a whole session from a Transmission Control Protocol (TCP).

d) Snort [Sno]: is an IDS system that makes it possible to analyse the traffic in real time and save logs of the messages. It can be used to analyse the protocols and search by patterns (protocol, origin, destination etc.). It can be used to detect various types of attack. Basically, it analyses the traffic in the network to detect patterns that might correspond to an attack. The system uses a series of rules to either produce a log of the situation (log) or warn the user (alert) or reject the information (drop).

e) Nessus [Nes]: detects any known vulnerabilities (by testing different intrusion techniques) and assesses the best security options for those discovered. It is a modular program that includes a series of plugins (more than 11,000) for performing the different analyses. It uses a client-server architecture, with a graphic client to show the results and the server, which carries out different tests on the machines. It has the capacity to examine whole networks. It

generates reports on the results, which can be exported to different formats (HTML, for example). Up until 2005, Nessus 2 was a free tool, but the company decided to make it proprietary, in version Nessus 3. In GNU/Linux, Nessus 2 is still used, as it continues to have a GPL license and a series of plugins, which are gradually updated. Nessus 3, as a proprietary tool for GNU/Linux, is more powerful and widely used, as it is one of the most popular security tools and there is normally a free version available with plugins that are less updated than the ones in the version that is not free.

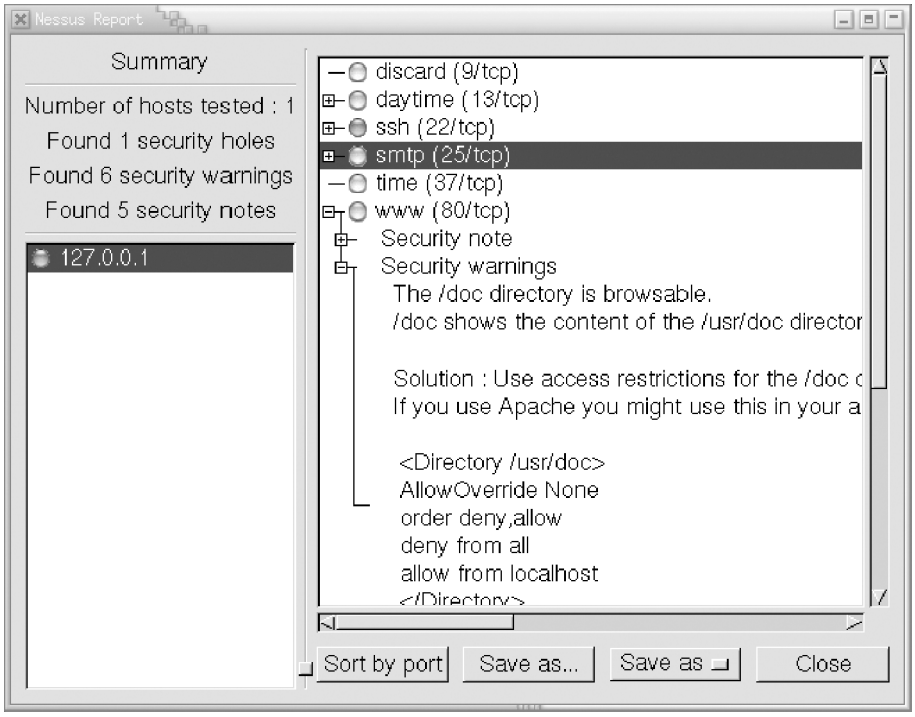

Figure 3. Nessus client showing the vulnerabilities report and the possible solutions

We can find many other security tools that are available. A good place to start is http://sectools.org, where the designers of Nmap maintain a list of popular tools, as voted by the users (now, a bit older list, but useful tools).

9. Logs analysis

By observing the log files [Ano99][Fri02], we can quickly get an idea of the global state of the system, as well as the latest events, and detect any irregular intrusions (or intrusion attempts). But it should also be remembered that, if there really has been an intrusion, the logs may have been cleaned or falsified. Most of the log files will be in the /var/log directory.

Many of the services may have their own logs, which are normally established during configuration (through the corresponding configuration file). Most of them usually use the log facilities incorporated in the Syslog through the Syslogd daemon. The configuration will be in /etc/syslog.conf. This configuration is usually established according to the message levels: there are different types of message according to their importance. Normally, levels such as debug, info, err, notice, warning, err, crit, alert, emerg, appear, in which the order of importance of the messages would be more or less as follows (from least to most important). Normally, most of the messages are sent to the /var/log/messages log, but the system can be set so that each message type goes to different files and it is also possible to identify who has created them; typically, the kernel, mail, news, the authentication system etc.

Consequently, it is appropriate to examine (or in any case adapt) the configuration of Syslog so as to determine the logs in which we can find / generate the information. Another important point is to control its growth, as, depending on which are active and the operations (and services) that are performed in the system, the logs can grow very quickly. In Debian and Fedora, this can be controlled through logrotated, a daemon that regularly makes copies and compresses the oldest logs; it is possible to find the general configuration in /etc/logrotate.conf, although some applications set specific configurations that can be found in the /etc/logrotate.d directory.

In the following points, we will discuss some of the log files that should be taken into account (perhaps the most frequently used):

a) /var/log/messages: is the default log file of the Syslogd daemon, but we would have to check its configuration, in case it has been moved to another place or there are several of them. This file contains a wide range of messages from various origins (different daemons, services or the same kernel); anything that looks irregular must be verified. If there has been an intrusion, the date of the intrusion and related files should be checked.

b) /var/log/utmp: this file contains binary information for each user that is currently active. It is useful to determine who is logged in the system. The who command uses this file to provide this information.

c) /var/log/wtmp: each time that a user logs in or out of the system, or the machine reboots, an entry is saved in this file. This is a binary file from which the last command obtains the information; the file records which users logged in or out of the system and when and where the connection was made. It can be useful for finding out where (in which accounts) the intrusion started and detect the use of suspicious accounts. There is also a variation in the command called lastb, which lists the login attempts that were not correctly validated and the /var/log/btmp file is used (you may have to create it if it doesn't exist). These same authentication faults can also be sent to log auth.log. In a similar manner, the lastlog command uses another file, /var/loglastlog, to verify which was the last connection of each of the users.

d) /var/log/secure: they are usually used in Fedora for sending the tcp *wrapper* messages (or firewalls). Each time that a connection is established to an inetd service, or, in the case of Red Hat 9, to the xinetd service (with its own security), a log message is added to this file. We can search for intrusion attempts in services that are not usually used or in unfamiliar machines that try to connect.

In the logs system, another thing that should be checked is that the directory logs in /var/log can only be writable by the root (or the daemons associated to the services). Otherwise, any attacker could falsify the information in the logs. Nevertheless, if attackers manage to access the root, they may often delete all their tracks.

10. Tutorial: tools for security analysis

We will now perform some of the processes described above on a Debian system, to improve the security configuration.

First we will examine what our machine offers the network. In order to do this, we will use the nmap tool as a port scanner. With the command (from the root):

```
nmap -sTU -O localhost
```

we obtain:

```
root@machine:~# nmap -sUT -O localhost
starting nmap 3.27 (www.insecure.org/nmap/) at 2003-09-17
11:31 CEST Interesting ports on localhost (127.0.0.1):
```

(The 3079 ports scanned but not shown below are in state: closed)

Port	State	Service
9/tcp	open	discard
9/udp	open	discard
13/tcp	Open	daytime
22/tcp	Open	smtp
25/tcp	open	time
37/tcp	open	time
37/udp	open	http
80/tcp	open	sunrpc
111/tcp	open	sunrpc
111/udp	open	auth
113/tcp	open	ipp
631/tcp	open	unknown
728/udp	open	
731/udp	open	netviewdm3
734/tcp	open	unknown

```
Remote operating system guess: Linux kernel 2.4.0-2.5.20
```

```
Uptime 2.011 days (since Mon Sep 15 11:14:57 2003)
Nmap run completed --1 IP address (1 host up) scanned in
9.404 seconds
```

We can see that a high number of open services have been detected (depending on the machine, there may be more: telnet, FTP, finger...), in both transmission control protocol (TCP) and user datagram protocol (UDP). Some services, such as discard, daytime, time may be useful on occasion, but they should not normally be open to the network, as they are considered non-secure. SMTP is the resending and routing service, for mail; if we are acting as the host or mail server, this would have to be active; but if we are only reading and writing emails through POP3 or IMAP accounts, this doesn't necessarily have to be active.

Another method for detecting active services would be by searching active listening ports, which can be achieved with *netstat -lut* command.

The *nmap* command can also be applied with the DNS or IP name of the machine; this shows us how the system looks from the exterior (with localhost, we see what the actual machine can see), or, better still, we could even use a machine of an external network (for example, any PC connected to the Internet) to examine what could be seen in our machine from outside.

We will now go to /etc/inetd.conf to deactivate these services. We should look for lines such as:

```
discard stream tcp nowait root internal
smtp stream tcp nowait mail /usr/sbin/exim exim –bs
```

and we type a number symbol (#) at the beginning of the line (only in the services that we wish to deactivate and when we know what they are really doing (check pages of man as deactivating them has been recommended). Another case of recommended deactivation would be that of the ftp, telnet, finger services and we should use ssh to replace them.

Now we have to reboot inetd so that it rereads the configuration that we have changed: /etc/init.d/inetd restart.

We return to nmap:

22/tcp	open	ssh
80/tcp	open	http
111/tcp	open	sunrpc
111/udp	open	sunrpc

113/tcp	open	auth
631/tcp	open	ipp
728/udp	open	unknown
734/tcp	open	unknown

From what is left, we have the ssh service, which we wish to leave active, and the web server, which we will stop for the moment:

```
/etc/init.d/apache stop
```

ipp is the printing service associated to CUPS. In the local administration section we saw that there was a CUPS web interface that connected to port 631. If we wish to have an idea of what a specific port is doing, we can look in /etc/services:

```
root@machine:# grep 631 /etc/services
ipp 631/tcp              # Internet Printing Protocol
ipp 631/udp              # Internet Printing Protocol
```

If we are not acting as the printing server to the exterior, we have to go to the CUPS configuration and eliminate this feature (for example, by placing a *listen* 127.0.0.1:631, so that only the local machine listens), or limit the access to the permitted machines.

Some other ports also appear as unknown, in this case, ports 728 and 734; this indicates that the system has not been able to determine which nmap is associated to the port. We will try to see it ourselves. For this, we can execute the netstat command on the system, which offers different statistics on the network system, from the packets sent and received and errors to the elements in which we are interested, which are the open connections and who is using them. We will try to find out who is using the unknown ports:

```
root@machine:~# netstat -anp | grep 728
udp 0 0 0.0.0.0:728 0.0.0.0:* 552/rpc.statd
```

And if we do the same with port 734, we can see that it was rpc.statd that opened the port; rpc.statd is a daemon associated to NFS (in this case, the system has an NFS server). If we repeat this process with ports 111, which appeared as sunrpc, we will see that the daemon that is behind is portmap, which is used in the remote procedure call system (RPC). The RPC system permits users to use the remote calls between two processes that are on different machines. portmap is a daemon that converts the calls that arrive at the port to the internal RPC services numbers and it is used by different servers such as NFS, NIS, NIS+.

The RPC services offered can be seen with the rpcinfo command:

```
root@machine:~# rpcinfo -p
```

programme vers	proto	Port
100000 2 tcp	111	portmapper
100000 2 udp	111	portmapper
100024 1 udp	731	status
100024 1 tcp	734	status
391002 1 tcp	39797	sgi_fam
391002 2 tcp	39797	sgi_fam

where we see the RPC services with some of the ports that had already been detected. Another command that may be useful is lsof, which, among other functions, makes it possible to relate ports with the services that have opened them (for example: lsof -i | grep 731).

The portmap daemon is somewhat critical with regard to security, as, in principle, it does not offer the client authentication mechanisms, as this is supposedly delegated to the service (NFS, NIS...). Consequently, portmap could be subjected to DoS attacks that could cause faults in the services or cause downtime. We usually protect portmap using some kind of wrapper and/or firewall. If we do not use these and we do not intend to use the NFS and NIS services, the best thing to do is to completely deactivate portmap, removing it from the runlevel on which it activates. We can also stop them momentarily with the following scripts (in Debian):

```
/etc/init.d/nfs-common
/etc/init.d/nfs-kernel-server
/etc/init.d/portmap
```

by entering the *stop* parameter to stop the RPC services (in this case NFS).

We will then control the security in the base using a simple wrapper. Let us suppose that we wish to let a specific machine pass through ssh, which we will call 1.2.3.4 (IP address). We will close portmap to the exterior, as we do not have NIS and we have an NFS server but we are not serving anything (we could close it, but we will leave it for future use). We will create a wrapper (we are assuming that the TCP wrappers are already installed) modifying the files hosts.deny -j allow. In /etc/hosts.deny:

```
ALL : ALL : spawn (/usr/sbin/safe_finger -l @%h \
| /usr/bin/mail -s "%c FAILED ACCESS TO %d!!" root) &
```

we are denying all of the services (be careful, some of them are related to inetd) (primer all), and the next step to take will be to find out who has requested the service and from what machine and we will send an email message to the root user reporting the attempt. We could also write a log file... Now, in /etc/hosts.allow:

```
sshd: 1.2.3.4
```

we enable access for the IP 1.2.3.4 machine in the sshd server (of the ssh). We could also enter the access to portmap, all we would need is a portmap line: la_ip. We can enter a list of machines or subnets that can use the service (see man hosts.allow). Remember that we also have the tcpdchk command to check that the configuration of the wrapper is correct and the tcpdmatch command to simulate what would happen with a specific attempt, for example:

```
root@machine:~# tcpdmatch sshd 1.2.3.4

warning: sshd: no such process name in /etc/inetd.conf client:
hostname machine.domain.es
client: address 1.2.3.4
server: process sshd
matched: /etc/hosts.allow line 13
access: grantedv
```

tells us that access would be provided. One detail is that it tells us that sshd is not in inetd.conf and, if we verify it, we see that it is not: it is not activated by the inetd server, but by the daemon in the runlevel on which we are operating. Besides, in Debian, this is a daemon that is compiled with the included wrapper libraries (which does not therefore require tcpd to work). In Debian, there are various daemons such as: ssh, portmap, in.talk, rpc.statd, rpc.mountd, among others. This allows us to secure these daemons using wrappers.

Another question that should be verified concerns the existing current connections. With the netstat -utp command, we can list the tcp, udp connections established with the exterior, whether they are incoming or outgoing; therefore, at any time, we can detect the connected clients and who we are connected to. Another important command (with multiple functions) is lsof, which can relate open files with established processes or connections on the network through lsof -i, which helps to detect any inappropriate accesses to files.

We could also use a firewall for similar processes (or as an added mechanism). We will begin by seeing how the rules of the firewall are at this time: (iptables -L command)

```
root@aopcjj:&#732;# iptables -L
```

```
Chain INPUT (policy ACCEPT)
target prot opt source                    destination
Chain FORWARD (policy ACCEPT)
target prot opt source                    destination
Chain OUTPUT (policy ACCEPT)
target prot opt source                    destination
```

In other words, the firewall is not placing any restrictions at this time and all the packets can be sent, received and resent.

At this point, we could add a firewall that would provide better management of the packets that we receive and send, and which would be a preliminary control for improving security. Depending on our needs, we would establish the necessary rules similarly to the firewall examples provided in this unit.

If we set up the firewalls, we can consider whether to use this mechanism as a single security element and remove the wrappers: this could be done, because the firewalls (in this case through iptables) offer a very powerful feature that allows us to follow up a packet by type, protocol and by what it is doing in the system. A good firewall could be enough, but, to be on the safe side, more security measures will always be helpful. And if the firewall were not well designed and let some packets escape, the wrapper would be the level-service measure for stopping any non-desired accesses. To offer a metaphor that is often used, if we consider our system to be like a medieval castle that must be defended, the moat and the front walls would be the firewall, and the second containment wall would be the wrapper.

Activities

1) Suppose that we locate a website in our machine, using Apache for example. Our site is designed for ten internal users, but we do not control this number. Subsequently, we consider making this system accessible on the Internet, as we think that it could be useful for the clients, and the only thing we have to do is assign a public IP address on the Internet to the system. What types of attack might this system suffer?

2) How can we detect the files with suid in our system? Which commands are necessary? And the directories with SUID or SGID? Why is it necessary, for example, for /usr/bin/passwd to have a SUID bit?

3) The .rhosts files, as we have seen, are a significant danger for security. Could we use some type of automatic method for regularly checking these files? How?

4) Let us suppose that we want to disable a service that we know has its /etc/init.d/service script that controls it: we wish to disable it in all the runlevels in which it appears. How do we find the runlevels in which it is present? (for example, searching for links to the script).

5) Examine the active services in your machine. Are they all necessary? How would we protect or deactivate them?

6) Practice using some of the described security tools (nmap, nessus etc.).

7) Which IPtables rules would be necessary for a machine that we only wish to access through SSH from a specific address?

8) What if we only want to access the web server?

Bibliography

Other sources of reference and information

[Deb] [Hatc] The security sites for the distributions.

[Peñ] Essential for Debian, with a very good description of how to configure security, that can be followed step by step, [Hatb] would be the equivalent for Fedora/Red Hat.

[Mou01] Excellent security reference for Red Hat (also applicable to Debian).

[Hat01] GNU/Linux security books covering extensive techniques and aspects.

[Line] Small guide (2 pages) to security.

[Sei] Step-by-step guide identifying the key points that have to be verified and the problems that may arise.

[Net] Project Netfilter, and IPTables.

[Ian] A list of TCP/IP ports.

[Proa] [Sno] [Insb] [Nes] Some of the most commonly used security tools.

[NSAb] Linux version focused on security, produced by the NSA. Reference for SELinux.

[CERa][Aus][Insa][Incb] [NSAa] Security organisations' sites.

[CERb][Ins][San] Vulnerabilities and exploits of the different operating systems.

[NSAa][FBI][USA] Some cybercrime "policies" in the United States.