ROZWÓJ POTENCJAŁU I OFERTY DYDAKTYCZNEJ POLITECHNIKI WROCŁAWSKIEJ

Wrocław University of Technology

Internet Engineering

Tomasz Surmacz

Secure Systems and Networks

Wrocław 2011

Projekt współfinansowany ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Wrocław University of Technology

Internet Engineering

Tomasz Surmacz

Secure Systems and Networks

Wrocław 2011

Reviewer: Maciej Nikodem

ISBN 978-83-62098-00-3

Published by PRINTPAP Łódź, www.printpap.pl

Contents

List of Figures

List of Tables

Chapter 1

Preface

This book accompanies the lecture called "Secure Systems and Networks" which is taught at Wroclaw University of Technology. It should be treated as a complementary material to the lecture, as it contains some more detailed explanations of covered topics and extended examples.

Topics covered in the lecture focus mainly on security of networked systems, so the main focus is set on networking protocols, their current and historic vulnerabilities and the impact these problems have on software. They are organized in a similar order that in the lecture.

As computer systems control more and more aspects of our everyday life, the importance of security of these systems becomes even more important. We need to be aware of potential problems of the existing systems so we can protect ourselves from intentional or accidental mischief, and as programmers and future authors of software running on these systems – we are obliged to write the software in a responsible way, knowing the pitfalls and traps that we may fall into.

The physical security of computers, network equipment and the infrastructure is very important, as well as proper access control systems and protocols that are used, as they all form the chain of interconnected elements that build the complete working system. Integrity of that system depends on all elements, starting with software, and to maintain the reliability and safety of the whole system it is important not to underestimate the relevance of each underlying element, being it hardware malfunction, a programming error, or the user ignorance.

Chapter 2

Basic security in computer systems

2.1 The main problem are people

Although it is possible to automate many access rules and procedures and deploy very sophisticated security policies, research shows that the weakest link in the computer system security are the people who use it. This is not even because people want to break the security intentionally, but due to several factors. These include:

- Lack of understanding for the rules.

  People tend to create simple, easy to remember passwords or store them in easily accessible places, such as a yellow sticky note under the table. Or are easily tricked to give away information that should be kept confidential. If they do not understand that their actions can sabotage the security of the whole network, they will not pay enough attention to real problems.

- No security when at home.

  Once people finish their work and go home, they often forget about the security constraints. Some people bring their work home and process confidential data on home computers full of viruses and malware programs, others talk openly with friends in pubs and share the information that should be kept secret. The same passwords are often used at work and in online social networks or other public servers.

- Disregard for threats and risks.

  Many times security is seen as an obstacle as people do not know personally any examples of a security breach or any other security problem. If it did not happen yet, what is a real chance of happening soon? Or is it just a nuisance in every-day work?

Last but not least – it is natural to us to stand in opposition to anything that is enforced on us. If a security policy is implemented, the overall impression may be that people are not to be trusted. Some may even feel offended. In overall, this may in turn stimulate sub-concious resistance and in fact weaken security instead of strengthening it.

2.2 Usernames and access rights

Unix and any other modern operating system intended for desktop/laptop computers are *multitasking* systems, which means they can run several programs at the same time. Such programs are called processes and each of them is running in a separate address space and is isolated from all other processes, i.e. they do not share the same memory and the communication and interaction between processes is limited to the mechanisms provided by the operating system. The actual multitasking is usually achieved by time-sharing, i.e. processes run on a processor for a specified maximum time (the *timeslot* and are then put away (frozen) to allow other processes to access a CPU

(This is called a *preemptive* multitasking – each process may be interrupted at any point of execution[*], not just the system calls). On multi-processor machines many processes may run at the same time on different CPUs, but still preemptive multitasking is used to allow running more processes than the actual number of CPUs.

Another feature of the system is its ability to run processes on behalf of different users. This gives ability to distinguish different users, offer them their own share of the disk space protected from curious eyes of other users, and protect their processes from those run by others. Different users may be assigned different access rights to filesystem, external devices and various other system resources. This feature is called a *multiuser* environment.

2.3 File access

File access in Unix is controlled by checking access rights associated with each individual file or directory[†]. Basic access rights include read, write and execute permissions (`r`, `w` and `x`) and are repeated in threecategories, for the file owner, group of users, and all others (`u`, `g` and `o`) – see fig. 2.1.

Access to a file for a process will be granted if one of these conditions applies:

- Effective uid of the process is equal to 0;
- Effective uid of the process is the same as the owner of the file and the "user" permission bits allow the required access;
- Effective uid of the process is *not* the same as the owner of the file, but the effective group id, or one of the extra groups that the user belongs to is the same as the group owner of the file and the "group" access bits allow the required access;

[*]Some operating systems may allow context switch only inside a system call. In such case we talk about non-preemptive multitasking.

[†]A directory is actually regarded as a special file containing the list of other files.

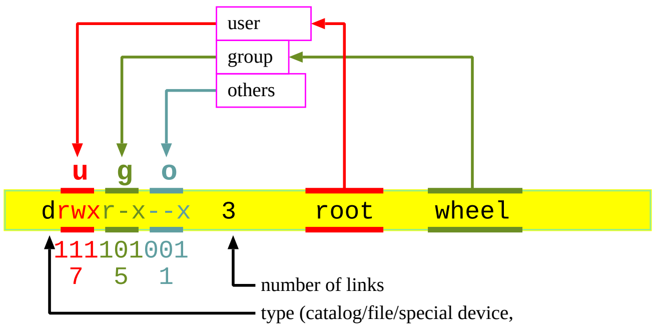


Figure 2.1: Unix file access rights

- Neither the *euid* or *egid* (or the auxiliary groups) match the file owners, but the "others" permission bits allow access.

In addition to the basic file permissions, Unix offers some special permission bits shown in fig. 2.2.

Set User ID bit (`04000` – (1) in fig. 2.2) allows changing the user ID related to a running process for the duration of running the program.

(1)	`-rwsr-xr-x`	–	set-user-id
(2)	`-rwxr-sr-x`	–	set-group-id
(3)	`-rw------T`	–	sticky bit
(4)	`-rwxr-xr-t`	–	sticky bit (`T+x`)
(5)	`drwxr-sr-x`	–	set-group-id
(6)	`drwxr-lr-x`	–	obligatory locking (`s-x`)
(7)	`drwxrwxrwt`	–	sticky bit

Figure 2.2: Special access rights in Unix

Set Group ID (`02000`) on an executable file (2) works much the same, but changing the effective group ID of the executed process. When used on directories (5) – allows inheriting the group owner on all files and subdirectories created (BSD behaviour). Without the corresponding execute bit set, as in (6) – forces the obligatory locking mechanism (i.e. file locking functions are blocking) over the advisory locking used normally.

Sticky bit (`01000`) – (7) in fig. 2.2, is used in directories with group/other write permissions (such as */tmp*). Normally, write access to a directory means not only the right to create files but also to delete the existing ones or modifying their names. So it is also possible that one process removes a file owned by other process and possibly replace it with another file or a symlink, which may lead to race condition (see section 8.6) or different problems. Sticky bit on a directory disallows removing of files owned by other users, or changing the attributes of such files.

Modern Unix systems also provide an extension to these typical access rights, which is called *Access Control Lists* or *ACLs*. These allow each individual file or directory to have a special extra list of users or groups with additional *allow* or *deny* rules.

2.4 UID and effective UID

In order to differentiate users and processes run by them, different accounts can be created in a Unix system, each requiring a separate login name and a password. For a system, however, it is much easier to distinguish different users not by names, but by unique numbers – so called "user IDs", or *UIDs*. The mapping between a user name and his UID is provided by the */etc/passwd* file.

Once a user accesses the system by providing a valid username and password and logging in, the system creates a login shell process by running the user's login shell (the last parameter in the */etc/passwd* file). This shell is run with user's UID, listed in the second field of the */etc/passwd* user entry and this UID is inherited by all subprocesses that this shell can spawn. So all the processes run by the user will be created with the same UID and will access file systems and system resources with access rights checked against this particular UID.

Special access for such programs is achieved by combination of two special features of the authorization system – *set user id* bit in file access rights and *effective user ID* (*euid*) in process information. Effective user ID is just like the normal *uid*, except that it is actually *the* user ID that the system checks when it needs to find whether a process can open a file or access a device. Usually, EUID and UID are the same, i.e. containing the same value inherited all the way down from the login process and identifying a user. So for most cases we can pretend that the UID value is the one we care about.

However, when a process executes a program that has a "set user ID" bit set in a filesystem, things change a little[‡]. The UID of the process remains the same, but the EUID is set to the owner of the file, i.e. the program that is being run from the disk. In the most common case of such programs, the file is owned by root, i.e. its owner ID is 0, and the resulting process will have its EUID set to 0, while the UID value is preserved from the parent process (and possibly from the login shell). In the same way, if a file has a "set group ID" bit set, that group is set as the running process's effective group.

[‡]this actually is done during the *exec*() system call.

Chapter 3

Authentication and authorization

One of the fundamental problems in computer systems security is establishing trust and identifying who is trying to access the system and whether he or she should be granted access to perform desired actions or not.

Humans have no problems identifying other humans – or at least a small subset of friends, relatives, or even other people seen from time to time. We can recognize the face, the overall look, the voice and many other details characteristic to a particular person and once we know this person's identity, we can easily connect these two.

Computers, on the other hand, have very limited peripheral devices when compared to human senses. They may have cameras installed, but face recognition software is not readily available and needs a lot of processing power. Same goes with voice recognition or other biometric information processing. So the mostly used devices for actual user identification are the simple ones – keyboards and appropriate logic behind the procedures being used for user *authentication*. If you enter a secret password, the computer system will grant you access, assuming you really know how to keep your password secret, so if anybody provides this password, it must be you. This assumption, although very fundamental for the security of system passwords, is hard to meet nowadays, as we learn about network sniffing or other password targeted attacks, such as phishing (see section 5.4).

Other methods of authentication include special tokens that a user must use every time when accessing a computer system – if we cannot identify a user directly, we may perform better with special devices designed especially for identification purposes. But again – if identifying such a device means identifying a user, we must assume that the device has not been lost, stolen or misplaced and is used by its sole owner, not somebody else.

3.1 Passwords

For long time passwords have been the mostly used system for identifying users. They are easy to use even by "computer illiterates", as the idea behind them is really simple – you need a username, and a secret. If you know the secret, you are in. The typical problems coming with passwords are not so obvious though – it is a common knowledge, that you are not expected to share your password with other people (still, some people do), and be careful when typing, so nobody is looking at your keyboard over your shoulder, but not too many people care about changing their passwords regularly or using a different password for every service they create accounts on. Nowadays, operating systems usually implement some better ways of authentication, such as fingertip scanning or configurable modules (by methods such as PAM described in section 3.3), but even now, passwords are the most commonly used authentication method in many web services, databases and many other programs.

Unix system used to store the passwords in the *letc/passwd* file together with some other data regarding user accounts. Passwords stored in a system

```
/etc/passwd:
  root:x:0:1:Super-User:/root:/sbin/sh
  rootjs:x:0:1:Super-User:/root:/usr/bin/bash
  js:x:500:10:Joe Shmoe:/home/js:/usr/bin/bash

/etc/shadow:
  root:Gna56gd2rA@fG:11581:0:99999:7:::
  rootjs:$1$Fixx5fG2nvvZAhskrei.sw$ns0SmHS1:11554:0:99999:7:::
  js:aZA56GD2Ra@FG:11581:0:99999:14:::
```

Figure 3.1: Sample lines from *passwd* and *shadow* files

are encrypted using a one-way hash function, so that decrypting the password is impossible. However, techniques such as *passwords cracking* (explained in detail in section 3.1.2 on page 23) forced the passwords to be moved to the */etc/shadow* file which does not have to be readable to every user of the system. The traditional Unix *crypt*() hash function allows 8-character passwords (truncating any following characters) which nowadays is not considered secure enough (passwords shown for users root and js in fig. 3.1 are samples of passwords encrypted this way), so it is now replaced with MD5 (user rootjs in fig. 3.1) or SHA. The other fields of the */etc/shadow* file contain the "aging" information, i.e. how often it has to be changed or when it finally expires.

3.1.1 Password vulnerabilities

Passwords are the traditional and the most widely used mechanism for accessing computer systems, but they also pose a lot of problems.

Contrary to the popular belief – they are also costly in maintenance. If they are not changed regularly, they may be broken. If, on the other hand, the computer system forces a user to change them too often, users will tend to forget them or write them down on sticky yellow notes attached to their screens, which is even worse.

Typical passwords are easy to guess or break. As more and more systems require some sort of access control, we are overwhelmed with more and more passwords to remember, so it is natural to expect, that people may use the same passwords over and over again, and that these passwords will not be very complicated. This way they will be also vulnerable to *password cracking* (see section 3.1.2). Passwords are also easy to find by just "looking over shoulder" while someone is just typing it. Maybe not the first time in whole, but definitely a part of it, and the whole password can be definitely found if there is a chance to see it typed just a few times.

Also, as they are often used to access remote systems over the network, and the standard network layers do not provide any transmission privacy, they may be stolen by tapping into the network and eavesdropping on just the very few packets of every connection being established.

If people have too many passwords to remember, they tend to create simple ones, that they will easily remember. Unfortunately, these also are easy to guess. So do not use passwords such as:

- Your name, spouse's name, girlfriend's name, dog's name. Anybody's name.
- Names of your favorite fantasy characters.
- The hostname of your computer.
- Your phone number or your license plate number.
- Anybody's birth date.
- Some information easily obtained about you (birth date, address)
- Some information based on your username.
- Words, which can be found in a dictionary. Any. Including Polish or Japanese `:-)`
- Simple patterns of letters on the keyboard, like `qwerty` or `3edc`

- The same as already set on some other computer system.
- Any of the above spelled backwards (!elpmaxe elttil siht ekil).
- Any of the above with first/last letter capitalized.
- Any of the above followed or prefixed by a single digit or punctuation.
- Any 0f 7he abov3 5pe11ed 1n 50m3 funny way (like `r00t` or `f00tba11`).

Thinking of a good password which will be not easy to guess may be sometimes difficult, so here is some advice what is considered a good password:

- Having both uppercase and lowercase letters.
- Hard to guess.
- Containing digits and/or punctuation characters. Also in the middle.
- Seven or more characters long.
- Easy to type quickly, i.e. hard too see when they are typed
- Not containing some problematic characters, such as `@` or `Ctrl-H`.

3.1.2 Password crackers

Traditionally passwords have been stored in the */etc/passwd* file which is readable by every user in the system, as it also provides the mapping between user IDs and usernames, which is needed on many occasions, such as listing files with `ls` or finding home directories of users.

In order to protect passwords, they have always been stored in the encrypted form, using a one-way hash function*. Whenever a user logs in, he provides his username and password, but to verify this information it is not necessary to have the password stored in the same form. The `login` program

*originally this was a *crypt*() function, which in modern systems is now replaced with MD5.

encrypts the password using the same one-way function that was used originally and compares the encrypted versions. If they match, the password given by the user must be correct, if they don't – there must be some difference. It is not possible to tell whether it was a typo, a simple one-character error or a completely different password. The password must be given exactly the same as when it was generated or the login process will fail.

Although it is not possible to reverse the process of encoding the password and retrieve its plaintext version from the encrypted form, it is quite feasible to try guessing it, as the encrypted version provides an easy way of verification.

With the constantly increasing computing power available for single processors or clusters of computers, this process is getting much easier than anticipated in the past. Table 3.1 shows estimated breaking times, assuming 8000 encryption operations per seconds (achievable on a single Pentium-3 100 MHz processor). With the current state-of-the-art quad-core 2.4 GHz processors we may expect 100-fold speed increase, which means the listed times will be 100 times shorter, so a 8-letter lowercase password is now crackable in about 8 hours instead of 302 days (still on a single processor).

Password cracking is a task that can be split into independent parts very easily, so it can benefit from massively parallel computing. Separate tasks may either work on a partial dictionary, or can be given a subset of all passwords. So the time needed to crack some passwords decreases in linear proportion to the number of computers that participate in the task. E.g. if we could afford to dedicate 1 000 000 computers for such a job, the time required decreases by 6 orders of magnitude, i.e. just over 3 months for a 8-digit password that may contain any characters of codes 1-127.

There are a lot of ready-to-use passwords crackers available from the Internet. Some of the most commonly used include JOHN THE RIPPER, THC HYDRA and the original CRACK program. There are also system- or application-specific crackers that can be dedicated to cracking passwords needed for accessing *PDF* or *ZIP* files, cracking WEP/WPA keys (AIRCRACK OR AIRSNORT),

Characters used	no. of passwords	breaking time required
4-digit password (PIN code)	10000	1.2 seconds
typical language vocabulary	100000	12 s
4-letter password	456976	1 minute
8-digits password	$1 \cdot 10^8$	3 hours 25 mins
6-letter password	$3 \cdot 10^8$	11 hours
8-letter password, lowercase	$2 \cdot 10^{10}$	302 days
8-character password, lowercase and digits	$2.8 \cdot 10^{12}$	11 years
8-letter password, mixed case	$5 \cdot 10^{13}$	212 years
8 characters, all ASCII 32-127	$7 \cdot 10^{15}$	28594 years
8 characters, all keyboard 1-127	$6.7 \cdot 10^{16}$	268246 years
All possibilities for MD5 hash function	$3.4 \cdot 10^{38}$	$1.3 \cdot 10^{27}$ years
Estimated age of the Universe		10^{10} years

Table 3.1: Strength of different passwords

or “recovering” passwords used to protect MS-Windows systems (e.g. Cain and Abel or SolarWinds). A useful list of available crackers may be found in [Lyo06].

3.2 One Time Passwords

The obvious solution to the problem of password sniffing are the passwords, that are changed frequently. So frequently, that a new password is required every time that the user logs in, so even when the password is sniffed, it will be useless. There are several systems allowing such way of operation, some of them requiring specialized hardware, others using just some clever ideas and a proper setup. The next few pages will show the most popular systems in use.

3.2.1 S/Key system

S/Key One Time Passwords system makes use of MD5 or MD4 hash functions to achieve uniqueness of passwords sent over insecure media, such as network. The idea behind the system is a very simple extension of the traditional password system which normally takes a user-provided password, encrypts it using a hash function and then stores such an encrypted form in the */etc/passwd* file. In S/Key, the password is run through the hash function not just once, but several times – usually 100 times (see fig. 3.2), but this is just an arbitrary number. When changing a password, user provides a new one – let's call it password number 0, and system has to store it in the encrypted form. In S/Key, password no. 0 is encrypted using MD5 function, producing 128 bits of data. Higher and lower 64 bits are XOR-ed, producing a 64-bit data, which becomes password no. 1. This in turn is run through MD5/XOR again to produce password no. 2. The process is repeated until we get password no. 99 which is finally stored in a password file – */etc/skeykeys*.

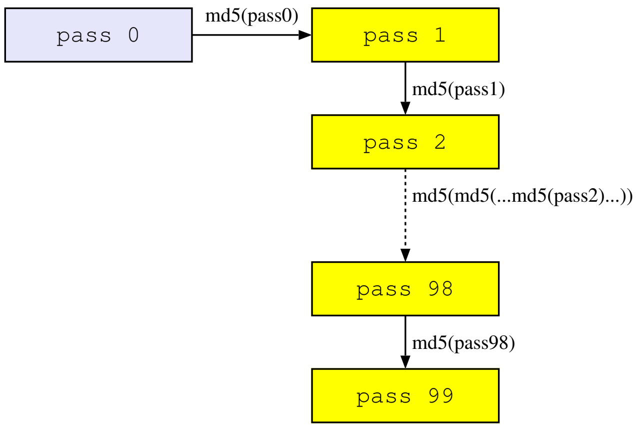


Figure 3.2: S/Key principles

Now – the password no. 99 is stored in a system and the user wants to log in. All he has to do is to provide a password no. 98. The system may calculate its MD5 hash, and if the password is correct it will match password no. 99 stored in the */etc/skeykeys* file. In such case, the stored password (i.e. encrypted password 98) will be replaced with the just entered password which is both the encrypted version of password 98 and encrypted version of password 97. So, next time the user wants to log in, he will be asked for password no. 97.

Security and usefulness of this system is based on simplicity of generating all subsequent passwords from any password we already have when compared to difficulties of doing the same in the opposite direction. If password no. 40 is known, generating passwords no. 41, 42, 43 ... is trivial, but finding passwords numbered between 0 and 39 is impossible or at least improbable (it would require processing power taking billions of years to finish).

3.2.2 Tokens

Tokens are small pieces of hardware that can be used as authentication devices, much like the keys may be used to open locks. For token authentication to succeed, the user has to prove that he has the token with him which may be checked in many ways. Tokens connected to RS232, parallel or USB port of a computer system may be used as a kind of a hardware license key, but nowadays it is much easier to depend on some short cryptographic data to be shown by a token itself and entered by a user.

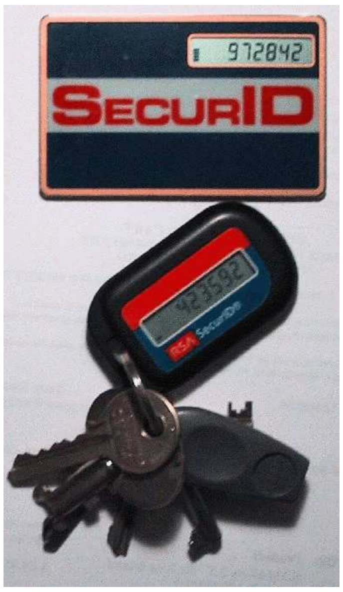


Figure 3.3: RSA OTP tokens

Tokens may either work as standalone devices independent of the outside world (i.e. only producing some output needed for authentication) or work bidirectionally –

requiring some input (a challenge) and producing the adequate response. Challenge-response tokens either require this data to be entered numerically through the built-in keypad, or contain some input device, such as photodiode, to allow some simple transmission by a "blinking" a special applet on the computer screen and holding the token in the right position to receive that data. Once the data is entered into the token, it calculates the response and displays in on a LCD screen, so that it may be typed by a user wishing to authenticate himself. The response calculation algorithm has to be secret, otherwise the token function could be duplicated.

Standalone tokens produce the output that changes from time to time. It may be given for example as a 6-digit number changing once a minute, looking like a sequence of random numbers, but the algorithm for that must be of course predictable, as the same sequence has to be generated at the authentication server. The authentication process then compares the data provided by the user with the current state of the sequence generator on the authentication server and if they match, the user is presumed to have the token, so the authentication succeeds. Figure 3.3 shows how these tokens may look like.

One of the problems with standalone tokens is the proper time synchronisation between the token and the server.

Even though it may look as a complicated task, the procedure is simple and straightforward. Both the token and a server contain the sequence generator, so if the server is able to produce the number which is considered a "current" one, it can produce also the next one and already knows the previous one which has just expired. In fact – it may go along the timeline and produce any value expected from the token in the past or in the future. So, if the token value is not current, but matches the next one in sequence, the server may note the time difference and adjust the sequence generator.

3.3 Pluggable Authentication Modules (PAM)

New authentication methods appear from time to time as new ideas are developed and new hardware becomes more popular and cheaper. If new methods of authentication are to be implemented, it is necessary to recompile lots of programs that usually do user authentication. This includes the login process done by the `/bin/login` program, but also connections by TELNET, RLOGIN, SSH, FTP, e-mail services provided by POP3 or IMAP, and any other network service.

Adding support for new emerging authentication methods by manual insertion of appropriate code of existing applications was the only option for some time, but now is considered to be a very inefficient way. Instead, there is a standardised way of calling library functions providing the authentication services, so that new methods can be added as shared libraries. This is called *PAM*, or *Pluggable Authentication Modules*.

Applications compatible with PAM do not check user credentials by themselves but rather call the appropriate functions in the PAM library. The main configuration of the PAM system is stored in */etc/pam.conf* file, but most of the actual functionality comes from the */etc/pam.d* directory, where there is a separate config for each service available in the system (see fig. 3.4). Adding new authentication methods can be achieved by simply installing the appropriate library (dynamically linked, in a form of a *libSomething.so* file) and referencing it from those services, that should use this method, either as a `sufficient` or a `required` prerequisite (fig. 3.5).

```
% ls -la /etc/pam.d/
  -rw-r--r--   1 root root   384 2008-04-03 03:02 chfn
  -rw-r--r--   1 root root    92 2009-07-31 15:46 chpasswd
  -rw-r--r--   1 root root   581 2008-04-03 03:02 chsh
  ...
  -rw-r--r--   1 root root  4113 2009-07-31 15:46 login
  -rw-r--r--   1 root root   520 2008-04-09 22:22 other
  -rw-r--r--   1 root root    92 2008-04-03 03:02 passwd
  -rw-r--r--   1 root root   168 2007-10-04 21:56 ppp
  -rw-r--r--   1 root root    69 2008-10-10 18:55 samba
  -rw-r--r--   1 root root  1272 2009-01-28 21:58 sshd
```

Figure 3.4: Sample contents of the */etc/pam.d* directory

```
  auth       [success=1]    pam_unix.so nullok_secure
  # here's the fallback if no module succeeds
  auth       requisite      pam_deny.so
  auth       sufficient     pam_rootok.so

  password   [success=1]    pam_unix.so obscure md5

  password   required       pam_permit.so
  # passwords may be provided by the gnome keyring
  password   optional       pam_gnome_keyring.so
  # or the   S/Key OTP system
  password   optional       pam_skey.so
```

Figure 3.5: Sample contents of a service using *pam.d* configuration

Chapter 4

Network Security

IP protocols, as we know them and use them every day, have been designed and developed in early 1970s as part of the ARPANET project. At that time, there were very few considerations for security of the actual data transmission, and most of it depended on the physical security of the wires used to connect the computing centres. Today we not only cannot control access to all the data cables and routers that are located along the route across the globe, but we go wireless using WiFi and WEP, meaning "Wired Equivalent Privacy" which we already know is misleading. In todays reality a lot of assumptions adopted long ago have to be revised, but some of them would require changes so fundamental, that they are just impractical, as the whole protocol stack and networking applications would have to be rewritten from the bottom up.

To solve such fundamental problems either new protocols have to be developed, or a workaround may be implemented. Both of these solutions however require a lot of effort spent in adopting the existing software. The best example is IPv6 – a new protocol designed to solve several problems existing in IPv4 (most notable ones being address space scalability and security). Even though it has been standardized in late '90s and the first implementations followed shortly, it has not replaced IPv4 until today. Most of the modern computer systems have nowadays the full support for IPv6, but it has not replaced the dominant IPv4 network. Even the security parts of the IPv6, such as IPSec need a lot of effort with rewriting existing applications and are implemented very slowly.

4.1 Basic IPv4 problems

IP networks suffer many problems inherited from the past, and the most notable ones include these:

- IP layer does not guarantee any data confidentiality (or encryption), so everything sent over the network may be overheard (*sniffed*).
- Many times we have to trust the data which is served from hosts beyond our control (e.g. the up-hierarchy DNS servers).
- Many protocols designed to improve security are the workarounds and no real solution is possible.
- The biggest advantage of IP networks – unlimited encapsulation and interoperability – is also the weakest security point, as any restrictions enforced by firewalls or similar mechanisms may be circumvented by encapsulating "unwanted" IP packets within other "good" IP packets that will match the firewall passing criteria.
- Security extensions and improvements have to be introduced as completely new protocols or separate protocol layers, on a one-by-one basis and only after a wide acceptance (e.g. SSL, IPSec)

All of this makes it really hard to deal with security problems of today's networks. Some of them are network-layer problems, some are application-layer problems. The most common ones will be described in the following sections, starting from the lower layers of the TCP/IP protocol stack.

4.2 Sniffing

Regardless of the actual physical media, ethernet networks are broadcast networks. Even if some data is sent from one host directly to another, it may be overheard by other machines located on the same local network, or on the path.

A sample conversation between two hosts is shown in fig. 4.1 where we can see a login attempt from host *asic* to an interactive login server at *cyber* using the TELNET protocol. The cleartext password "`abc`" can be clearly seen when it is sent from the client to the server, even though it is not echoed back. The sample shown is the output of a SNOOP program run on Solaris platform, but similar tools are available on other operating systems. In Linux we may use TCPDUMP which is usually installed by default, or programs such as SNORT, IPGRAB, ETTERCAP, ONE WAY NETWORK SNIFFER, etc. A lot of these programs are available from `http://freshmeat.net/` and `http://sourceforge.net/`.

Sniffing is very hard to detect, as it does not generate any extra traffic and is rather a passive way of grabbing the packets from the network interface. Tools for sniffing are also widely available – the most common ones are the protocol analyzers, i.e. network monitoring tools, such as TCPDUMP [JLM09] or WIRESHARK [wir10]. There are also libraries, such as LIBPCAP, with APIs that may be used for sniffing from within programs [Gar08].

4.3 Spoofing

Spoofing is a technique of injecting some contents into a conversation between two communicating computer systems in a way that impersonates somebody else. There are many ways of spoofing that can be done on different levels, starting from physical/network layers and ending in particular protocols or authentication methods.

```
49 asic ts/pub/src# snoop cyber
  asic -> cyber   TELNET C port=53218
 cyber -> asic    TELNET R port=53218 login:
  asic -> cyber   TELNET C port=53218
  asic -> cyber   TELNET C port=53218 t
 cyber -> asic    TELNET R port=53218 t
  asic -> cyber   TELNET C port=53218
  asic -> cyber   TELNET C port=53218 s
 cyber -> asic    TELNET R port=53218 s
  asic -> cyber   TELNET C port=53218
 cyber -> asic    TELNET R port=53218 s/key 90 cy11009\r\n
  asic -> cyber   TELNET C port=53218
 cyber -> asic    TELNET R port=53218 PASSCODE or Password
  asic -> cyber   TELNET C port=53218
  asic -> cyber   TELNET C port=53218 a
 cyber -> asic    TELNET R port=53218
  asic -> cyber   TELNET C port=53218 b
 cyber -> asic    TELNET R port=53218
  asic -> cyber   TELNET C port=53218 c
 cyber -> asic    TELNET R port=53218
  asic -> cyber   TELNET C port=53218
```

Figure 4.1: Sniffing network data using the SNOOP program.

4.3.1 DNS Spoofing

Lets consider a local network that consists of a few servers and several workstations that access services located in these servers. These may include disks exported from the servers, authentication services, internal WWW pages, online library resources or any other services. Servers do not offer these services just to anybody in the world but usually restrict the access to just a range of IP addresses or – especially in case of larger networks – to a specified domain owned by the organisation.

When a request for a service comes, all that is known is the IP address of the client. In order to find the corresponding domain name of the client it is necessary to consult the DNS service – in this case, a reverse mapping from IP addresses to names. This information however, will be provided by the DNS servers belonging to the client site and may give false information.

Let us assume that the server in question is *sun1000.pwr.wroc.pl* with IP address *156.17.1.33* and there is an incoming connection to that server from IP *63.0.3.1*. To check the name of the client, it is necessary to find the mapping for *1.3.0.63.in-addr.arpa*. The information will be provided by the server responsible for *3.0.63.in-addr.arpa* or *0.63.in-addr.arpa* subdomain and there is nothing that can prevent this server from saying that, for example, reverse mapping for *63.0.3.1* is *sun1000.pwr.wroc.pl*, so that the server will assume the connection is coming from its own address (or it may be just any name belonging to the local "trusted" network).

In order to check whether this information is valid, it is necessary to cross-check the given name with a non-reverse mapping, i.e. translate it from the domain name to the IP address (or addresses, as there may exist many IP interfaces with different IPs, all associated with the same host). The mapping done in this direction will be provided by local DNS servers (i.e. in this case – the ones responsible for *pwr.wroc.pl* domain). If one of the returned IP addresses for that host name matches the IP address that we used to find the reverse mapping, everything is OK and the reverse translation is valid. No match in the address list is a clear indication of DNS spoofing taking place, i.e. someone trying to steal some of services from the *pwr.wroc.pl* domain by pretending to belong to this domain as a client.

For all the services that are started through the `inetd` daemon, spoofing prevention is very simple with the TCPD package [Ven95]. The TCPD program is run by `inetd` instead of the actual service daemon (such as `telnetd` or `rlogind`) but with original parameters, allowing TCPD to resume normal operation and executing the appropriate server once all the security checks have been finished (see fig. 4.2). These include checks against DNS spoofing first, but a customized configuration file may be provided, giving the server a choice whether to accept the connection or not, based on criteria such as the client's IP address, DNS name, or DNS reverse mapping status. A sample

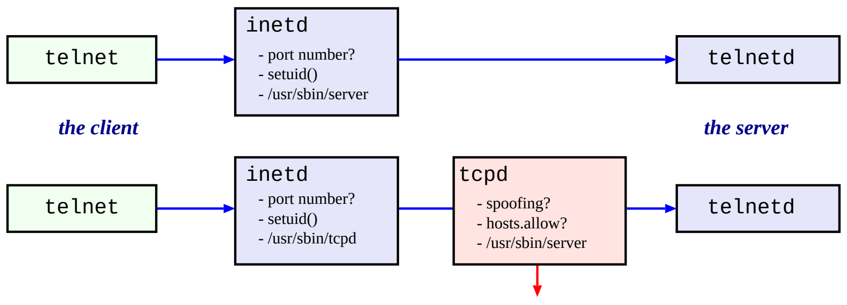


Figure 4.2: Anti-spoof checking by TCPD

of such configuration is shown in fig. 4.3.

```
# our router reads its config using tftp, other hosts are banned
# from tftpd access
in.tftpd: router.pwr.wroc.pl : allow
in.tftpd: all : rfc931 : deny

# other services
in.talkd,in.fingerd : 156.17.0.0/255.255.0.0 : allow
in.talkd,in.fingerd : ALL : deny

# we allow telnet only from hosts registered in DNS
in.telnetd : KNOWN : rfc931 : allow
ALL : ALL : rfc931 : deny
```

Figure 4.3: Sample *hosts.allow* file for TCPD

Even though the TCPD uses both */etc/hosts.allow* and */etc/hosts.deny* files (in that order) with the assumed ALLOW/DENY actions set accordingly, it is usually better to put all entries in just one file and explicitly set whether each matching line should allow or deny access. In the given example we first deal with TFTP service, which is normally used to boot up local diskless clients, provide configuration files for routers, switches and other network equipment or backup their configurations, so access may be limited to a specific host and all other addresses may be denied. The `rfc931` option requests querying the IDENTD daemon on the other side to find who is making the connection.

The TALKD and FINGERD programs are then allowed from a metropolitan area network (including Wroclaw University of Technology as well as other academic institutions in Wroclaw), denying access from all other sources. Interactive access to TELNETD is then allowed from hosts that have proper reverse-DNS mapping and finally, all connections to other services from any IP addresses are denied by the last line.

4.3.2 SMTP Spoofing

SMTP spoofing is even simpler and may be done without any special access or tools. All is needed is some kind of a client (such as TELNET or NETCAT allowing to establish a connection to port 25 on which mail servers listen to packets. Mail servers do not normally ask clients or any connecting servers for their credentials, allowing anybody to connect and drop mail that will be later delivered.

The SMTP protocol (or its enhanced version called ESMTP) is actually quite simple and easy to use – even to type the conversation manually. A sample SMTP session of sending spoofed email is shown in figure 4.4.

4.4 Link and network layer problems

Some of the security problems are not specific to a particular application, but are common to all networking programs due to their nature. The next four sections describe some problems that are located in layers 2 and 3 of the OSI model and may affect all applications running on top of them.

```
--> telnet localhost 25
    Trying 127.0.0.1...
    Connected to localhost.
    Escape character is '^'.
<-- 220 doh.pwr.wroc.pl ESMTP Sendmail; Wed, 6 Oct 2010 12:43:01 +0200
--> ehlo spoofer
<-- 250 doh.pwr.wroc.pl Hello localhost [127.0.0.1], pleased to meet you
--> mail from: <santa@heaven.org>
<-- 250 <santa@heaven.org>... Sender ok
--> rcpt to: <ts@localhost>
<-- 250 <ts@localhost>... Recipient ok
--> data
<-- 354 Enter mail, end with "." on a line by itself
--> From: Santa <santa@heaven.org>
--> To: ts
-->
--> What will you do if you catch me?
--> .
<-- 250 CAA05466 Message accepted for delivery
```

Figure 4.4: SMTP spoofing

4.4.1 ARP

Hosts on a network communicate using various methods, depending where they are located.

To send a packet to a host on a local network the network layer (IP) puts the appropriate IP address as the destination, and the link layer adds the MAC address of the corresponding host. This needs to be a correct address so that the destination host's network card will pick the packet off the network. If the packet is sent to a host on a remote network, the destination IP will be of that host, but the network layer determines that it has to go through a forwarding router with a specified IP address, so the destination address in layer 2 data of the packet must be set to the router's MAC address. The proper mapping between layer 2 and layer 3 addresses is needed then every time a packet is

```
solaris> arp -a
Net to Media Table
Device   IP Address               Mask      Flags   Phys Addr
------ -------------------- --------------- ----- ---------------
le0    ALL-SYSTEMS.MCAST.NET 255.255.255.255       01:00:5e:00:00:01
le0    rush                  255.255.255.255       00:10:5a:48:1e:20
le0    hop.ict.pwr.wroc.pl   255.255.255.255       00:e0:63:04:1c:c0
le0    okapi                 255.255.255.255 SP    08:00:20:73:c8:42

Linux> arp -a
asic (156.17.41.90) at 08:00:20:7B:07:FA [ether] on eth0
gorg1 (156.17.41.81) at 00:C0:DF:AC:9B:63 [ether] PERM on eth0
gorg2 (156.17.41.82) at 00:C0:DF:C1:A2:CB [ether] PERM on eth0
test (156.17.41.69) at * PERM PUP on eth0
```

Figure 4.5: Sample ARP tables on two hosts

sent over the network, whether its destination is local or remote.

These mappings are required for all hosts on the local network. Depending on local topology and determined by the netmask, local segments may span over just few host (e.g. 16) or may contain several hundreds of them (like 256 or 4096). Adding the required IP/MAC mappings manually would be a tedious and ineffective task, so special protocols are used to automate this task – ARP (*Address Resolution Protocol*) and RARP (*Reverse ARP*).

Whenever a packet is sent to a local IP address, the ARP table is consulted (a sample of such tables obtained with the `arp -a` is shown in fig. 4.5). If a match is found, the corresponding MAC address is used. If not – the network layer sends the ARP packet, which is the request for a mapping from the requested IP address to the yet unknown MAC address. This packet is sent as broadcast, so if the target host discovers that the requested IP address is his IP address, it sends a reply, which is then cached at least for several minutes. Whenever the mapping is used for sending packets, its "time to live" in the ARP table is prolonged, and if it is ever expired and deleted, it may be regained by the ARP protocol again, when needed.

This mechanism is simple, yet very effective in maintaining the up-to-date information about the local network, even when it changes dynamically, as people come and go connecting and disconnecting their laptops or switching on and off their desktop computers.

But as all other simple dynamic protocols this one is also prone to abuse. If any host on a network decides to respond to *all* ARP requests providing its own MAC address, it will effectively steal the packets that were meant for other hosts. This not only escalates the sniffing problem, but provides a way for man-in-the-middle attacks, as the stealing host may inspect the packets addressed for the victim, alter their contents and send them using the proper MAC address. Other possible misuses come from whatever may be done by redirecting the packets from the victim to the attacker. For example, if the victim is a WWW server, the attacking host may launch a copy of that server with modified web pages, leading to server identity theft and server spoofing.

4.4.2 Denial of Service attacks

Under this category we can file every type of an attack in which one user or a system process takes so much of a system resource or resources, that it gets exhausted and other users and processes can no longer use it. It may include disk space, CPU time, processes, network connections, network bandwidth, access to particular devices or services, communication mechanisms or anything that is available in limited amounts. Some of these attacks are in fact a side-effect of the ill-written programs or scripts or the effect of a simple mistake, such as the wrong comparison at the loop conditional, no error checking of the results value from the system call (or endless repeating of an action in case of an error that cannot be corrected this way).

For the attacks originating locally, i.e. within a system, there are some tools that may prevent such an abuse from happening. These mainly include mechanisms that limit the usage for particular types of resources for a single process or a process group. If such limits are in place, a process trying to exhaustively overuse a system resource (being it a disk space, a CPU time or a memory), will be either simply denied access to that resource, once it reaches the limit, or killed, if the limit appears to be the CPU time. It may prevent some unexpected side-effects of programs being tested, but is usually not a good idea for production servers and programs run in such environments, as it may actually interfere with legitimately running programs that happen to be servers running all the time and thus accumulating over time the CPU time or other resources in a way that is really hard to predict. If we overestimate the imposed limit, it will not be effective, but if we underestimate it – it may actually prevent the legitimate server from running under normal conditions.

SYN flooding

One of the network DoS attacks is *SYN flooding*. It exploits TCP protocol vulnerability which is present since the creation of this protocol (i.e. around 1970), but has been declared a problem in 2000. Establishing TCP connection requires a 3-way handshake between the client and the server, as shown in fig. 4.6.

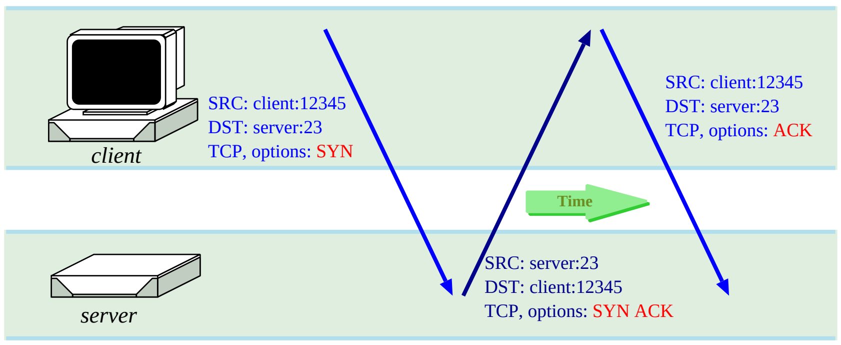


Figure 4.6: Establishing TCP connection

Client sends a connection request to a well-known server port. This request is a TCP packet with special combination of flags: the SYN flag is set and the

ACK flag is reset. The server responds to such a packet by accepting the connection and responding with another special packet, which has both SYN and ACK flags set, and then waits for the final confirmation from the client. The connection at this stage is in a *half-open* state, meaning it has already been accepted by the server, but still in the connection establishment phase, so that the server cannot simply discard the connection, but has to wait until it is finally acknowledged from the client by sending the packet with ACK set and SYN flag reset.

The problem appears if some malicious clients start sending the first SYN packets, but then never complete the connections. The typical queues in the TCP/IP stack of protocols allow only a small number of outstanding connections (ranging from 50 to 1000) so this limit may get exhausted very easily. And because it is a system-wide limit affecting the TCP connection establishment, once it is reached, it will not be possible to create *any* new TCP connection. So, for example, the WWW server may be attacked by SYN flooding a SMTP port on the same server, or any other open port.

Broadcast storms

Also called *ping flooding*. Exploits poorly configured gateways which allow sending co-called *directed broadcasts*. When using IP sub-netting the address space assigned to an entity such as a company or a university is divided in smaller parts, called subnets. It is always done at predefined boundaries (determined by a netmask), so for example we may have a network spanning across the addresses *156.17.1.0* to *156.17.1.63* and another one from *156.17.1.64* to *156.17.1.127* when the netmask is set to *255.255.255.192*. There are 26 bits set in the binary form of the netmask, so we may also write those networks as *156.17.1.0/26* and *156.17.1.64/26* respectively. In every subnetwork there are two addresses which have special meaning – the first one (i.e. *156.17.1.0* or *156.17.1.64*) is the network address. The last one (i.e. *156.17.1.63* or *156.17.1.127*) are the directed broadcast address. If you send a PING packet to such an address, you get the response from every single host that is present on that network, much the same when sending a packet to the *255.255.255.255* address. The only difference is that the packets to *255.255.255.255* are "real" broadcasts which are cut-out at the subnetwork boundary by a router, while the directed broadcasts are legal unicast packets throughout the whole Internet (and can be forwarded by routers) except their final destination, i.e. their own network, where they are treated as broadcasts.

4.4.3 TCP/IP stack vulnerabilities

There are also vulnerabilities in TCP/IP stack that had been present in different implementations of most of the operating systems for many years and have been discovered only during the last few years. It happened so, because in the early days of TCP/IP development the main goal to achieve was the interconnectivity and efficiency of the network protocols, not the security aspects, although many problems were addressed at that time and solved instantly. However, some bugs or loopholes remained undetected and were adopted in many implementations ported then to new and new operating systems and hardware platforms.

The first of such problems that were detected in 1997 was Ping of Death. It relies on the RFC 971[DeS86] specification of a maximum packet size being 65535 bytes (i.e. $2^{16}-1$). Packets longer than MTU (which is usually set to 1500 bytes on ethernet media) have to be split into fragments, which are reassembled at the receiving end of the communication. However, due to a bug in implementation of packet reassembly code (affecting not only ICMP, but just any IP protocol, e.g. UDP or TCP) it was possible to send a fragment with a maximum allowed offset (65528, which is $(2^{13}-1)\cdot 8$), and a size causing the reassembled packet to exceed its maximum size. As this was not anticipated, almost all of the TCP/IP implementations caused buffer overflow errors while in the kernel mode, which affected the whole operating system, usually causing a system crash.

Once this attack has become widely known, people started browsing the source code of TCP/IP implementations for possible other errors and some other problems have been found in a very short period of time. Some exploit programs have also been created. Teardrop is one of such attacks where overlapping fragments of a packet may cause a system crash. In other systems the Land attack was successful, where packets coming from the network interface had the same source and destination address.

4.5 Networking protocols

Different networking protocols have security problems of their own. According to the OSI model, a protocol is generally placed in the highest layer (the *application layer*), but in fact most of the applications and the communication protocols that they use define all aspects from session establishment, through data interpretation, up to application-level functional specification, so they occupy the three highest layers of the OSI model (i.e. *session*, *presentation*, and *application*). In the following sections we look in detail into some of the most popular application-layer protocols and their vulnerabilities.

4.5.1 Interactive login

Standard connection protocols such as TELNET, RLOGIN, RSH, or REXEC, do not use encryption and are vulnerable to data sniffing and password stealing. Accounts that are left open and active when a user no longer needs to use the system (e.g. after changing the employment, or moving out to another city), are also a security threat, as they may get taken by someone breaking into the system. Having a "login shell" in a system across the globe is a very desirable thing, as it may help the attacker to cover his tracks when trying to break in into even more systems, by using several "hops", so that tracking back the origin of the connection will be hard or impossible, as it will require cooperation of many system administrators of all the computers used in such a chain of connections. A hunt for an attacker using this technique (based on a real story) is described in [Sto90].

To make things even worse, RLOGIN and RSH protocols allow users to log in without checking a password if there is an entry for a (hostname, username) pair in user's *~/.rhosts* file. For checking this, the destination server trusts entirely the information provided by a `rlogin`/`rsh` client. The solution to this problem is to use the `ssh` [BSB05] program instead, which encrypts the whole communication, so that the passwords cannot be sniffed, but also, for other methods of authentication, first of all it checks the credentials of the connecting host/client for any signs of spoofing, and only then allows alternative authentication methods, such as user login keys (similar to certificates).

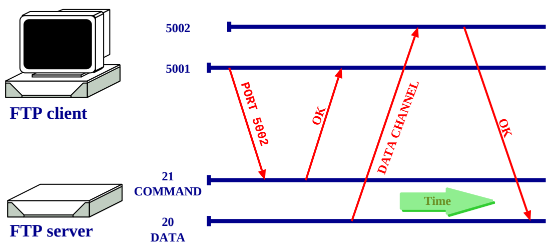
4.5.2 FTP

FTP protocol is one of the most complex and sophisticated protocols developed so far. Its main goals are to provide a fast an reliable data transfer with maximum possible transfer rates, but also allowing to abort the lengthy transfers at any time or to resume transfers that have failed for any reason. All servers allow also conversion between text formats between Unix/DOS/Mac machines, and modern servers allow features like file compression "on the fly" or transfer of the whole directories with a single GET command.

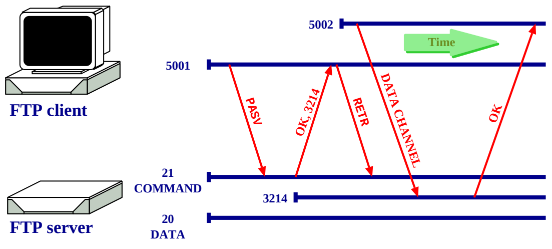
The way that FTP protocol works is actually quite complicated and may not work in modern firewalled networks by default. Port 21 is used for control connection and port 20 for data transfer. FTP server can operate either in *active mode*, which is the default for most of the clients, or the *passive mode*, which is preferred by the web-based clients such as FIREFOX. These two modes of operation are shown in fig. 4.7.

In active mode (fig. 4.7(a)) the client connects first to port 21 of the server and issues commands, but commands requiring data transfer, such as `DIR`, `GET` or `RETR` cause the client to issue a `PORT` command, informing a server about the port number that client has opened. It is the server then who opens the connection to the client from its own port 20 and the data is transferred using this new connection, while the old one may still be used to issue other commands (e.g. to abort the transfer). When transfer is complete, this connection is closed automatically.

Data transfer in passive mode (fig. 4.7(b)) uses additional port on the server. The client first uses the `PASV` command to indicate its willingness to initiate the passive mode data transfer to which the server opens some random additional port and informs the client of its number with the `OK` response. The client then opens connection to the given port and transmits the data.



(a) FTP in active mode



(b) FTP in passive mode

Figure 4.7: FTP modes of operation: (a) active and (b) passive

4.5.3 Anonymous FTP

Anonymous FTP poses even more threats to the computer system, as by the definition, it allows access for any users of the Internet and if any problem with the service is discovered, it may be exploited by anybody. So the proper setup of anonymous FTP service is crucial for the security of the server running it.

```
# Special users
deny-uid %-99 %65534-
deny-gid %-99 %65534-
allow-uid ftp
allow-gid ftp

# various user groups and classes
guestgroup ftpchroot
guestuser *

class  all         real,guest,anonymous  *
class  localreal   real *.pwr.wroc.pl 156.17.*.*

# access limits for various user classes
limit  dead        0  Any                /etc/ftp/msg.dead
limit  localreal  30  Any                /etc/ftp/msg.toomany
limit  anonymous  100 SaSu|Any1800-0800  /etc/ftp/msg.toomany.offpeak

# possibility of compressing files or directories on-the-fly
compress        yes            anonymous guest real localreal
tar             yes            anonymous guest real localreal

# permissions to modify files and execute special commands
delete          no      guest,anonymous       # deleting files?
overwrite       no      guest,anonymous       # replacing them?
chmod           no      anonymous             # changing permissions?
umask           no      anonymous             # changing umask?
rename          no      anonymous             # renaming files?

# uploading configuration for various subdirectories
upload  /pub/ftp        *                 no
upload  /pub/ftp        /pub              no
upload  /pub/ftp        /pub/incoming     yes  ftpadm  ftp  0620 dirs

# filters for potentially unsafe file names
path-filter  anonymous  /etc/msg.badpath  ^[-A-Za-z0-9_\.]*$  ^\.  ^-
path-filter  guest      /etc/msg.badpath  ^[-A-Za-z0-9_\.]*$  ^\.  ^-
```

Figure 4.8: Sample anonymous FTP server setup file

The setup process is complicated and there are lots of common errors that may be made. These include for example publishing real password file inside the server's */etc* directory instead of a specially prepared one, or improper file modes on special files needed in the */dev* directory. Careless configuration of upload directories may cause creation of warez servers and if the FTP server is not constrained by the chroot-ed environment, then any break-in to the FTP service may allow the penetration of the whole host system. Such break-ins are quite probable, given the actual history of various FTP servers, such as WUFTPD or PROFTPD.

A part of sample WUFTPD config file is provided in fig. 4.8 to show the complexity of anonymous FTP server setup. Care has to be taken not only to provide definitions of various classes of users and granting them appropriate access to specific directories within the FTP server's space, but also to limit their access to unwanted places and prevent them from creating potentially dangerous files. It is also important to setup proper file permissions on files left by anonymous users in the upload directories and disallow various operations, such as renaming or overwriting files. Further setup options are needed if a firewall system is used on the server host and the FTP server has to bind to specific port numbers for active or passive mode transfers.

4.5.4 NFS

NFS service provides disk access in local networks. Disks (or rather filesystems) are exported from servers and mounted on NFS clients. The service uses UDP (NFS v.2) or UDP and TCP (NFS v.3) as the transport layer mechanism, but using a higher layers abstracts provided by RPC (Remote Procedure Calls). For a proper NFS operation a helper program PORTMAP is needed (residing on UDP port 111) which translates RPC function numbers to TCP/UDP port numbers. NFS service requests are directed to `mountd` or `nfsd` daemons which also must be running on the server.

In NFS vulnerability problems exist in both directions: servers have to be protected against clients cheating on file permissions or access restrictions, and clients have to be careful about contents of the imported filesystems.

Protecting clients against malicious servers

Clients are vulnerable to problems coming from any special file permissions and elevated privileges present on files exported from servers. If someone has root access to the NFS server, it is trivial to create a suid script or a program that may spawn a root-owned shell or do just anything – create a new user, add a cron job, install a rootkit, etc. It will be pointless to run such a program on the server, but if all special access rights are respected at the client that mounted the filesystem with a planted suid script, it would be possible to break into the client system.

To circumvent these problems, clients may restrict interpretation of some special features from the server's disk. The `nosuid` option tells the client to ignore any suid and sgid bits on a mounted filesystem, `noexec` ignores all `x` bits on regular and special files (they are still needed on directories), so it will not be possible to run any program from a mounted disk. The `nodev` option disallow special interpretation of the device files.

Protecting servers against malicious clients

It is also possible to pose security threats from the client to the NFS server. If someone obtains root access to the NFS client that happens to have a filesystem mounted from the server with a write access, he may create files on that filesystem as root and change their permissions – e.g. by adding the suid bit and creating a suid-shell or a suid-script. If this shell is then run by a regular user on the server, root privileges may be obtained instantly.

In order to block this behaviour the default export rules for a filesystem map root-access from a client to *uid* of nobody, so no harm can be done to the server system. If a client machine can be trusted and really needs root access to the server's disk, the filesystem may be exported with `root=client_name` (Solaris/SysV) or `no_root_squash(client_name)` (Linux) option.

Another problem comes with so called *loop exports*, i.e. exports to *localhost*. Some systems required or advised such exports for automounter services. However, port forwarding or IP tunneling techniques could exploit this as NFS servers could mistakenly identify mount requests coming out of the tunnel as local requests. This may lead to a security breach where any non-root user may mount an exported file system, thus bypassing any access rights protection on that system.

Here are some final remarks on exporting and importing filesystems through NFS:

- Export filesystems only to a closed list of clients, never to "everybody"
- Export filesystems in read-only mode, whenever possible
- Use `no_root_squash` or `root` options with caution.
- Turn on verification of a source port of NFS client requests (Solaris: `nfs_portmon`)
- Avoid loopback exports.
- Restrict access to PORTMAP and RPCBIND daemons through TCPD or a firewall.

Some useful commands to check the current state of the NFS system are: `mount`, `showmount -e servername` or `cat /etc/mtab`.

4.5.5 DNS

DNS service translates host names ("human readable form") to IP addresses and IP addresses to host names ("reverse DNS") and is one of the very fundamental network services. Practically every program uses it through the *lib-*

nsl.so library. Servers use caching for storage of all the information learned from other servers.

The DNS system is organized in hierarchical way, i.e. delegating subdomains to their respective servers. The top-level domain is the "." domain, i.e. the dot domain. All queries that cannot be answered from the cache have to start with this domain and go from the top level down to the needed subdomain. For example, when asking for a translation of *sun1000.pwr.wroc.pl* the following chain of events happens:

1. The first query finds servers responsible for the "." top-level domain:

```
.              NS      A.ROOT-SERVERS.NET
.              NS      B.ROOT-SERVERS.NET
.              NS      C.ROOT-SERVERS.NET
[...]
.              NS      M.ROOT-SERVERS.NET
```

2. These servers only delegate top-level domains (both functional and national) to other servers, e.g.:

```
pl.            NS      NMS.CYFRONET.KRAKOW.pl
pl.            NS      DNS2.MAN.LODZ.pl
pl.            NS      BILBO.NASK.ORG.pl
pl.            NS      DNS.FUW.EDU.pl
pl.            NS      DNS2.TPSA.pl
pl.            NS      SUNIC.SUNET.SE
```

3. Each of these servers has information about all the subdomains and hosts listed directly under 'pl' or any other top-level domain, so they list 'wroc.pl' subdomain too.

```
wroc.pl        NS      ldhpux.immt.pwr.wroc.pl
wroc.pl        NS      sun2.pwr.wroc.pl
wroc.pl       NS      alfa.nask.wroc.pl
wroc.pl       NS      bilbo.nask.org.pl
wroc.pl       NS      dns.uw.edu.pl
wroc.pl       NS      wask.wask.wroc.pl
```

4. For “sun1000.pwr.wroc.pl” the above servers have to be queried about “pwr.wroc.pl” subdomain:

```
pwr.wroc.pl   NS      wask.wask.wroc.pl
pwr.wroc.pl   NS      ldhpux.immt.pwr.wroc.pl
pwr.wroc.pl   NS      sun2.pwr.wroc.pl
pwr.wroc.pl   NS      dns.uw.edu.pl
```

5. And finally the query may be made for the `IN A` record:

```
sun1000.pwr.wroc.pl  IN    A      156.17.1.33
sun1000.pwr.wroc.pl  IN    A      156.17.250.2
```

All of this happens automatically whenever any DNS query is made, but is not visible in such detail. It is possible to see a summary of the information needed to find a desired host by adding the `-v` (verbose) flag to the `host` command. Figure 4.9 shows such a detailed information for the final stage of the above query. Servers use caching for storage of all the information learned from other servers. Clients can be often served from cached information. Numbers before the `IN` keyword are the TTL (Time To Live) values for each record, i.e. how long (in seconds) a particular record will be kept in the cache of the DNS server that sent this answer.

This however also poses security threats if some server in the chain is modified to provide falsified information – it will be caches for as long as it was designed to, so a service redirection through DNS is possible, which is extremely dangerous as falsified information circulating between servers can “live” there for a long time, and a poisoned server may propagate this information to other servers.

```
% host -v -t a sun1000.pwr.wroc.pl

;; QUESTION SECTION:
;sun1000.pwr.wroc.pl.           IN      A

;; ANSWER SECTION:
sun1000.pwr.wroc.pl.    3600    IN      A       156.17.1.33
sun1000.pwr.wroc.pl.    3600    IN      A       156.17.250.2

;; AUTHORITY SECTION:
pwr.wroc.pl.            532     IN      NS      dns2.pwr.wroc.pl.
pwr.wroc.pl.            532     IN      NS      sun2.pwr.wroc.pl.
pwr.wroc.pl.            532     IN      NS      ns1.net.icm.edu.pl.
pwr.wroc.pl.            532     IN      NS      wask.wask.wroc.pl.
pwr.wroc.pl.            532     IN      NS      dns.pwr.wroc.pl.
pwr.wroc.pl.            532     IN      NS      ns2.net.icm.edu.pl.

;; ADDITIONAL SECTION:
dns.pwr.wroc.pl.        1433    IN      A       156.17.18.10
ns1.net.icm.edu.pl.     82798   IN      A       193.0.71.133
ns2.net.icm.edu.pl.     83318   IN      A       212.87.0.71
dns2.pwr.wroc.pl.       518     IN      A       156.17.18.11
sun2.pwr.wroc.pl.       3568    IN      A       156.17.5.2
wask.wask.wroc.pl.      82798   IN      A       156.17.254.3
```

Figure 4.9: Summary of a DNS query

Typical DNS problems

DNS service problems that affect security of the local network may have a local or remote sources. Very often they are just a configuration error, but sometimes they may be a remotely exploitable bug in the server software.The typical problems are:

- Server poisoning – in answers given to client queries.
- Attempts to use security holes in DNS servers allowing remote com-

mand execution and host break-ins.

- Configuration errors leading to either information leakage, service disruption or a server remote exploit vulnerabilities. For example – loops between secondary servers may cause a long-term DNS-poisoning propagated from one server to another.
- There are many known security bugs in old versions of the name server program (`bind` versions 4.x.x and 8.1.x – buffer overflow errors).

4.5.6 WWW

WWW, since its creation and public release in 1993, has become the most popular internet protocol. The versatility of available media and techniques and easiness of implementing new extensions have greatly helped in gaining this popularity. Nowadays for many users WWW services are "The Internet", as they do not even realize the existence of any other IP protocols, maybe with a tiny exception for messaging/communications services such as Skype or Jabber or peer-to-peer file sharing, but even these are often substituted by web page gateways.

Security problems follow – Easiness of installing plugins and various extensions encourages spreading of malware such as viruses or trojan horses. Weak passwords and authentication schemes lead to sensitive information leakouts. Weak server protection mechanisms and misconfigured servers are abused to destroy the reputation of many companies for which the web presence is now a must.

There are several security concerns regarding WWW usage:

Stealing sensitive information

Information stored on the WWW server may have limited circulation – e.g. it can be stored in areas with limited access, protected by a password, or access rules based on IP address of a client. However, it may be possible to bypass these, or obtain a list of files that are not visible through index pages, etc.

Spreading of malware
Easy file transfers and automatic file type recognition – possibility of installing trojan horses or other malicious programs by clients.

Server break-ins
A WWW server is a complicated and complex piece of software that includes many plug-ins and pieces of other helper programs that have to interact properly. Configuration errors or software bugs may lead to remote exploit vulnerabilities leading to server break-in and further possibilities of mischievous actions.

Bad programming leading to further problems
Server break-ins and all other web-related problems are usually caused by bad programming habits of the programmers who develop WWW-based services. As these may use several programming languages, different data models and various back-ends for data, interoperability problems become one of the main issues. The OWASP Top 10 document [The10] lists the most critical web application security risks identified today.

HTTP protocol threats

HTTP protocol is vulnerable to DNS spoofing (as any other internet service) and eavesdropping (sniffing). This is especially important due to the type of offered service and the type of data (online shopping, e-banking, access to restricted data). Web proxies and caching servers add to this problem, as they are another potential place for stealing or modifying data.

Web browsers are a piece of software that changes very rapidly. They act on a perimeter – interpreting the contents which is offered by WWW servers, but as the technology changes, the services are modified in the same way and very often incompatibilities are found with older versions of the same software, or with a different browser which has not yet implemented some features that web page authors find trendy.

WWW pages have long gone from the static stadium to dynamically changing contents. Different technologies are used for that:

JavasSript

Code is executed inside a client web browser, loaded from the server.

JavaScript has access to a local filesystem and cookies. It can also perform intentional or unintentional DoS attacks when a JavaScript code starts consuming a lot of processing power in a very non-efficient way due to lazy programming. File access exposes users to stealing personal data and sending them (without user's consent) to the server. This is especially important as some browser errors further expose users to so-called *cross-site scripting*, where the contents of one web page redirects the browser to run a script loaded from another server.

Java

Constrained environment in which Java code is executed.

Strong restrictions preventing programs from "escaping" this constrained environment and from allocating too many system resources (e.g. CPU time). Java standard has been introduced by Sun Microsystems and browser developers were required to acquire compatibility certificates after rigorous testing to actually be able to use Java name in their product description. However, many incompatible extensions have been introduced and gained popularity (mostly in Microsoft servers and clients), forcing other software to use them too, in order not to be left behind.

Server-side includes

Executed on the server side, so affecting the server security, not the client. However, lousy written (and poorly tested) code may cause effects similar to DoS attacks. Also, they need very strict setup of the server to prevent elevation of access rights (e.g. when a server-side script can access and present some files on the system that would normally be inaccessible to a user due to file permissions).

Another source of HTTP-related break-ins are the software bugs and holes in the server code. WWW servers have evolved to a very complicated pieces of software where they no longer just serve the files identified by an URL. There are server-side scripts, CGI and PHP extensions, plug-ins and other extensions that all need separation of privileges and access rights checking, potentially different in each context in which some part of a web site is executing that dynamic part.

Some examples of server problems are listed below:

- Apache – access rules in old versions of the server software allowed bypassing the "allow" and "deny" directives for directory access, thus exposing the contents of hidden directories.
- MS IIS – vulnerable to viruses and worms spreading through specially crafted HTTP queries (e.g. Nimda virus).
- WWW page requests are generally context-free. Binding several requests into a session may be done through user *cookies* or by encoding session information in returned URLs. Parsing these information must be resistant to any modifications made by user.
- Some programs implement a web-type model of user interface (such as Frontpage, Satan/SAINT), by running as a kind of temporary www-server and starting a web browser for user interaction.

Counteracting WWW service security threats

Lessening the security threats regarding HTTP protocol requires acting on many levels – both at the server side and in the clients. Some techniques may prevent the actual problems, some others may decrease the size of damage possible if something bad happens. Some other eliminate the threats by avoiding the potential problems.

In the server:

- Running the server with *uid*=nobody.
- Using *chroot*() constrained environment.
- Symbolic links and server `<!--include ...>` directives.
- Users should be able to run processes in WWW server context (e.g. CGI scripts).
- Proper access rights on directories containing the served documents and running a user-provided scripts by the server with user access rights or nobody to avoid giving out sensitive data or abuse by local users.
- Session context should be kept at the server. Client should only be given a "handle" to this context.
- Detailed validation of all the data sent by the client and appropriate error handling.

In server scripts:

- Detailed analysis of all error return codes in server scripts and its functions (they should do whatever is expected or return an error message).
- Scripts must be prepared to deal with malformed data and queries.
- Data provided by users may be modified maliciously to bypass server checks (special characters, injected cookies, exceeded length of variables, replaced data in drop-down lists, etc.)

On the client side – scepticism, mistrust and a common sense are the best remedies:

- Various browser additions allow blocking malicious code (such as NoScript, FlashBlock and similar).
- When accessing pages in secured mode (e.g. when making online purchases or during online banking) – paying attention to whether the access is really encrypted and if the server certificates are valid.

4.5.7 SSL and TLS

Secure Sockets Layer (SSL) is a collection of library routines used to improve security of several services using TCP protocol. It has gained the most popularity with HTTP, but can also be used in SMTP, IMAP, TELNET and many other protocols.

SSL has been developed as an intermediate layer between transport and session layers (fig. 4.10). SSL acts as a lower session layer and is applicable to connection-oriented underlying transport protocols, such as TCP. To use SSL, first a normal TCP connection is established, that yields two connected sockets, one at each endpoint of the communication. These sockets are in turn used to "upgrade" the connection to the encrypted/authenticated status by calling appropriate SSL functions.

An extension to SSL is the TLS protocol (Transport Layer Security) described first in RFC 2246 [DA99] and still developed on the IETF standards track (currently in RFC 5246 [DR08] with various updates in RFC 5746). It is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. Using TLS, the client and the server negotiate how the connection should be established in the handshake phase, where they exchange information about ciphers and hash functions that each site can use and decide on the actual algorithms – the strongest ones from the common subset.

The server also sends back to the client its digital certificate which acts as a form of identification to prevent server identity theft. A session key is generated to use symmetric encryption for the duration of the connection. If any

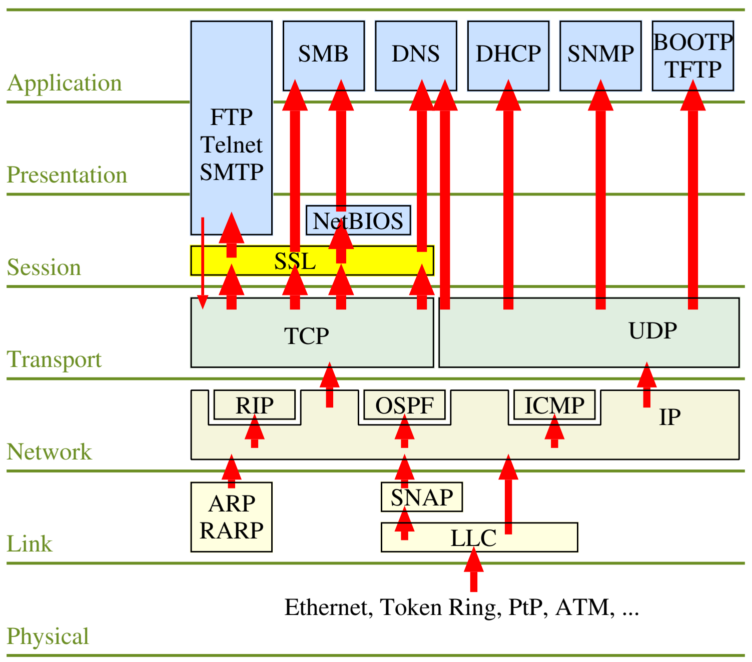


Figure 4.10: SSL position in the TCP/IP protocol stack

problems are found during connection setup (such as too weak algorithms or an invalid certificate) the connection is dropped.

TLS may be used in many connection-oriented protocols, mostly seen in HTTPS for web browsing, but extensions exist to use TLS in FTP, SMTP [Hof02], NNTP and in creation of VPNs. It has also been implemented in datagram-oriented protocols (UDP) as *Datagram Transport Layer Security* (DTLS [RM06]).

4.5.8 SMTP

Electronic mail is probably the most heavily used network application, which is used not only to communicate in the traditional way, but also to share documents, pictures or large binary files. As an application it may be considered as a distributed program that runs cross every computer architecture and virtually any platform. People expect to be able to use it on any operating system, on a PC, a laptop or a mobile telephone.

SMTP architecture

First e-mail systems date well before anybody heard about the Internet – when TCP/IP protocols were just being developed for local networks and the typical site-to-site connectivity was achievable by modems calling over long distance telephone lines. E-mail worked on these through the UUCP (Unix to Unix Copy) protocol, by the accept-store-forward paradigm. This holds true also today – servers accept e-mail from users, store them locally for reliability, then forward it to other servers, only the underlying media has changed, greatly improving both the throughput and the direct connectivity between the e-mail systems.

The e-mail system consists of several types of programs that have to co-operate to make the system work (fig. 4.11). *Mail User Agents* (*MUA*) allow

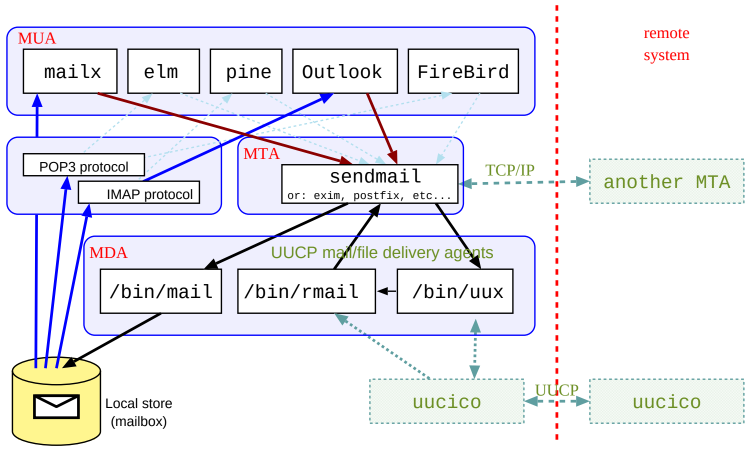


Figure 4.11: E-mail system architecture.

users to read mail, write new messages and send them. *MTA* programs (*Mail*

Transfer Agents) accept mail from external hosts or send it between hosts. *MDA* programs (*Mail Delivery Agents*) put mail in user mailboxes on local machines, after the mail reached a local system.

As MTA server accept SMTP commands directly from other servers, and are reachable directly from the Internet, they may be vulnerable to remote exploit attacks. DoS type attacks are also possible and even today many e-mail servers are configured in a way that allows relaying spam through them.

MTA programs can be also run from command-line by local users and be exploitable this way. Some programs such as `sendmail` have a very bad reputation on having new and new security-related bugs discovered, but even `sendmail` replacements such as QMAIL, EXIM or POSTFIX have suffered from security problems.

MDA programs need special privileges in order to append messages to different user mailboxes, thus are especially vulnerable to any local exploits, even if not directly accessible from the outside. In Unix systems, MDA programs are usually suid programs or are a part of the MTA program.

MUA programs do not usually connect to the outside world and are run on behalf of the user with access rights restricted to that user only, so they cannot be used to elevate user privileges to the level of a system administrator, but a much greater security threat comes from the fact, that MUAs usually can run arbitrary programs based on MIME encoding and attachments present in the received mail. This may lead to exploiting bugs in these helper programs, and to spreading of viruses and worms if an e-mail attachment is treated as a program to run.

Open Relays

E-mail systems pre-date the Internet and deep in their design the basic mode of operation is still accept-store-forward. Much the same, as the traditional Post system works – letters are accepted over the counter or from postboxes, then they are sorted and forwarded to next Post Office in the chain, which either delivers the letter locally, or forwards it to the next Post Office, closer to the final destination.

All e-mail servers work in a similar way, forwarding messages on behalf of other systems, in a shared effort to deliver them to final users. However, what was good when the general system-to-system connectivity was not very common, leads now to a mail-relay vulnerability often abused by spammers. A spammer that has poor Internet connectivity (either in terms of a poor bandwidth or if his IP address is filtered by anti-spam filters) may connect to a vulnerable e-mail system and inject his spam messages there. The vulnerable MTA server will accept such a message and try to deliver it to all the recipients, even if they are not located on the server's system. Each such a message is addressed to many recipients, so the spammer saves a lot of his bandwidth, as all of the burden of delivering the messages is taken by the cooperating MTA (see fig. 4.12).

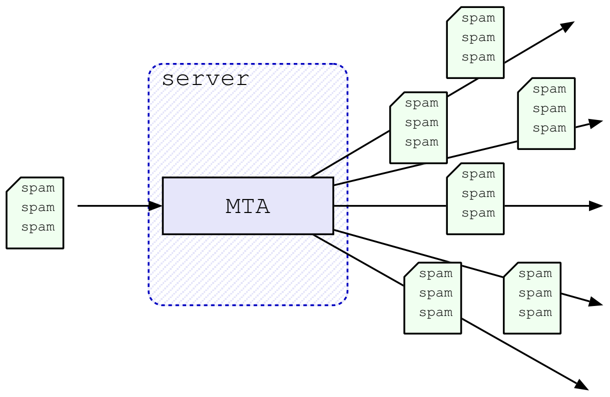


Figure 4.12: Spamming through an open relay

This is due to the "open relay" server configuration, which has been the default mode of operation on many servers until around 2005, when finally most of the typical MTA programs changed their policy by distributing sources and precompiled binaries with default configs that disallowed such a behaviour. Other factors that add to this problem are the lack of authorization when submitting email from clients and impossibility of user identification which further encourages e-mail spoofing.

Mail Protocols

E-mail system uses different protocols for different tasks. MTAs connect to each other using SMTP or ESMTP. The same protocols are also used for initial mail submission from clients to servers. From the server's point of view there is no indication and no distinction whether it is the client or another server who is connecting. As a result, clients are over-trusted and may in fact submit messages with any headers, as servers do not have reliable ways of checking whether the given From: addresses are legitimate, so e-mail spoofing and other types of attack are possible.

Some attempts to stop e-mail abuse try to do reverse-checking of the addresses, but this technique has a limited usability now, as spammers have learned to use only valid addresses.

Chapter 5

Malware

Malware programs are computer viruses, trojans or any other software that intentionally interfere with other programs to disrupt their functions or exploit their vulnerabilities and propagate to other systems without the user's consent.

5.1 Viruses

Viruses are the malware programs that try to replicate themselves by attaching their code to other executable programs in a system. Once an infected program is run, the virus code is executed and tries to infect other programs found on disk.

Viruses are usually associated with programs distributed in binary form, loaded from untrusted sources, such as shareware archives, usenet groups, etc. They may take different forms – in the beginning they were usually some programs that infected MSDOS/MS-Windows binary programs and got transferred through floppy disks either attached to specific files or to the boot sectors. Later, with the changes in media technology, the favorite way for virus spreading became the CDs – again with the binary programs distributed this way, and with a new armor – autorun scripts. Nowadays most of the viruses get sent over the e-mail and are based on mail client bugs to actually run the e-mail attachments without user permission and against system defense mechanisms.

Computer viruses may actually appear in any file which is in some way "executed", whether it is a direct assembly code or an interpreted language. So, even though these may not be so popular, the viruses may appear in:

- PostScript files which are automatically interpreted and contain commands to be executed by an interpreter lacking security checks.
- Web server applets executing at client side, in browser context, if they are able to modify files or spawn new processes.
- MIME-encoded mail, containing automatically executed programs (IE bugs: *.exe*, *.bat* and *.pif* files with *.gif* extension, still being executed automatically, without any warnings.
- Macro-based viruses in various multi-platform tools, such as Staroffice, MS-Office, etc. This includes also JavasScript embedded in PDF files or any other file format.

Protection against viruses involves both active prevention and "damage control" – i.e. active scanning of any potentially dangerous code before it has a chance to run, as well as data integrity checks of the installed programs and pro-active checking, usually based on virus signatures.

In Unix systems the virus problem is almost non-existent, as proper file permissions in the system setup prevent users from modifying system-owned binaries, and due to varying system architectures and different hardware platforms, it is not easy for a virus to spread out from one system to another. However, it may still be possible to infect some user-owned programs on most popular hardware platforms (e.g. a Linux system running on PC hardware), so a few precautions should be taken at least:

- Avoiding non-standard directories in your path (especially "." – i.e. the current directory)
- Avoiding running code loaded from untrusted sources.
- Keeping safe permissions on directories (no group/other write permission).

It is also worth noting, that very often, the erratic behaviour of a program is not a virus, but a result of poor programming and software bugs.

5.2 Trojan Horses

Even though the viruses do not affect the email servers, the incoming mail is automatically scanned for viruses just to protect the clients running their mail browser on unprotected client workstations. The attachments are scanned for virus signatures and even if they are compressed and packed using archivers such as ZIP or ARJ, they are unpacked first, to really find any malicious content.

Malware authors also try to outsmart these automated checks, and as always, the weakest link to attack is the people. So for example the attached ZIP archives may be password protected to prevent the automatic archive opening by the anti-virus software, but the password for the archive is attached in the text or graphical form with an explanation that may look like "*Your requested file is attached. The password is...*".

The unlocked archive contains an executable program, which is then run by a user, infecting his system. As the active cooperation from the user is needed, such a program is called a *trojan horse*.

5.3 Worms

Computer viruses propagate in passive way – they cannot survive on their own, but need a program or a file they can attach to and spread to other programs when their host is executed. They may also infect other computers, but mainly when transferred by e-mail, or when the infected program is copied from one system to another and then executed.

Computer *worms*, on the other hand, actively seek ways to propagate to other computers by exploiting know vulnerabilities in some programs or typical configuration problems.

Worms need some security holes in order to propagate, so the protection against them is basically the same as protection against break-ins. This includes monitoring the system for unusual activities, such as rapidly increasing umber of emails sent, skyrocketing CPU usage, rapidly growing size of the log files and number of logged errors – especially the ones that are generated by `identd` which show unusual number of outgoing connections.

5.4 Phishing attacks

The term *phishing* comes from joining together two words – *password* and *fishing* and describes the fraudulent actions of setting up fake servers collecting sensitive information from unsuspecting users – such as passwords (hence the name), usernames, credit card numbers, authorization tokens for bank operations, etc. Phishing attacks are usually targeted at popular social web sites, auction sites, online banks, but also the IT departments of various big institutions.

In the sample e-mail shown in fig. 5.1 the link is shown twice – once in the mail body where it looks like the Amazon reference (but in fact is a *link name*), and second time – in the "References" section, where all links are expanded to show their actual targets. This is how textual e-mail clients (such

```
From: "Amazon.com" <account-update@amazon.com>
Subject: Revision to Your Amazon.com Account

  Amazon Information
  We are contacting you to inform you that our Account Review
  Team identified some unusual activity in your account. In
  accordance with Amazon's User Agreement and to ensure that
  your account has not been compromised, access to your account
  was limited. Your account access will remain limited until this
  issue has been resolved.  This process is mandatory, and if not
  completed within the nearest time your account or credit card
  may be subject for temporary suspension. To securely confirm
  your Amazon information please click on the link bellow:
  [1]https://www.amazon.com/cgi-bin/webscr?cmd=login-run

  We encourage you to log in and perform the steps necessary to
  restore your account access as soon as possible. Allowing your
  account access to remain limited for an extended period of time
  may result in further limitations on the use of your account
  and possible account closure.

References

  1. http://96.10.247.13/evo/www.amazon.com/index.html
```

Figure 5.1: Phishing e-mail targeting Amazon.com users

as MUTT or PINE) show such e-mails. Webmail-type browsing in FIREFOX or INTERNET EXPLORER will show just the name, and the actual link target will be shown in the browser status line only when the mouse cursor is over the link name. And even not for certain, as some bugs in INTERNET EXPLORER may truncate the link being shown, or some special characters in the link may cause only the trailing part of the link being visible, and that part may look like the normal link.

5.5 Backdoors

Historically, backdoors (or trapdoors) are pieces of code in a program or an operating system, that allow a "shortcut" access for programmers, to make debugging easier. Debugging is a tedious task, requiring a repetition of particular actions to set-up a desired state of an application in which the problem may be reproduced and then studied. A backdoor may be then a special command or a hidden feature that leads exactly to that testing point, or at least makes some repeating steps easier to do by skipping over them. This often includes logging to the system, providing passwords, accessing particular database records, or in general – setting a particular state of the whole program using some shortcut, that skips other parts of the application that would normally get executed.

Nowadays however, the term "backdoors" has moved its meaning rather to new and intentional holes in system security planted there by the attacker in the sole purpose to (possibly) make accessing this system easier in the future, especially if the original hole used for breaking-in is discovered and plugged in.

Usually, a backdoor is also hidden, and planted in non-obvious way.

A typical backdoor may for example replace a LOGIN, TELNETD, FTPD, RSHD or any other network service program, so that a replaced version of that program will work perfectly normal under usual circumstances, but will also grant access to a computer system if some special user/password combination is provided (not listed in system's */etc/passwd* file. In order to further hide changes made to the system, a *rootkit* may be also installed% .

There are many ways to install a backdoor. A replaced version of `login`, `telnetd`, `ftpd`, `rshd` or other network service program may allow an attacker to login without a password. Something added in *.rhosts*, *.ssh/authorized_keys* or other access control file will have a similar effect. Changed owner of some directories or files (e.g. */etc*), allows file deletion or modification in future.

Some other ways of installing a backdoor include:

- Added aliases in the mail system, allowing remote execution of programs.
- Added harmless-looking network service, allowing remote access.
- Installed a suid program giving a shell with root access when some conditions are met.
- Tweaked */etc/exports* or */etc/fstab* files, allowing NFS mounts by unauthorized hosts and modifying files or running suid programs.

Chapter 6

Firewalls

Most of the operating systems come with lots of features enabled by default. Many of these features appear to get some security holes discovered sooner or later, and even for services appearing to be secure, many default configurations cause serious security problems.

Computer systems need constant patching and software updating to keep them current and safe.

Unexperienced systems administrators are tempted to install "just everything" and then forgetting or not knowing what to uninstall. Installing a firewall around them does not protect from local exploits, but at least keeps "remote" intruders out.

If any security hole gets discovered for some *needed* service, the system administrator has some extra time to correct the problem, as the exploitable service is not immediately seen to the outside world.

6.1 Firewall types

Firewalls vary in their design and the level on which they operate. Most commonly used are operating on layers 3 and 4 or the OSI model, the more sophisticated ones do also *packet inspection*, interpreting the contents of the higher layers.

6.1.1 Stateless firewall

Simple firewalls use unsophisticated mechanism called packet filtering to decide what traffic should be allowed to go through. The criteria for allowing or disallowing a particular packet may be based on:

- Protocol type (UDP, TCP, ICMP, ARP, etc.)
- Source or destination port number (for TCP or UDP)
- Source IP address
- Destination IP address
- Packet type: SYN, SYN/ACK, data, ICMP echo, FIN, etc.
- Some specific IP options (e.g. fragmentation)

As this kind of filtering considers each packet independently of all other packets, it is also called *stateless filtering* or a *stateless firewall.* It is useful for simple protection, but there are more advanced network scanning techniques, such as stealth scanning, which will succeed in passing through a stateless firewall.

6.1.2 Stateful firewall

Modern network scanning attacks often use packet forging and spoofing techniques to get past a firewalled system. Such attacks easily get through, if the only criteria is based on as single packet characteristics. To prevent these more sophisticated attacks the firewall must keep track of all the incoming and outgoing packets and then update its state table, which contains the information about all known connections (for TCP) and packet flows (for UDP and ICMP). By default, a stateful firewall blocks all packets and has special rules only for connection-initiating packets, i.e. in TCP – the packets that have the SYN flag set. If such a packet is allowed (i.e. the connection may be initiated in the given direction – from, or to a protected network), and the actual connection is really established (appropriate packets are observed) the firewall automatically adds *allow* rules for packets belonging to that connection. These rules are in operation until the connection is closed (a packet with a FIN flag is observed) or there are no packets exchanged for a predefined time.

6.2 IPTables and Netfilter

IPTables and Netfilter is a framework for packet mangling, outside the normal Berkeley socket interface. It is used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel (starting with version 2.4.x).

A firewall using IPTables is defined by means of *tables* containing *chains* through which packets are checked. There are several predefined tables, such as `filter`, `nat` or `mangle`, each containing a number of predefined chains, such as `INPUT`, `FORWARD` and `OUTPUT` for the `filter` table. Each chain is a list of rules which can match a set of packets. The flow of the packets through the `filter` table is shown in fig. 6.1.

Each rule of the chain specifies what to do with a packet that matches. A firewall rule specifies criteria for a packet, and a target. If the packet does not match, the next rule in the chain is the examined; if it does match, then the next rule is specified by the value of the target, which can be the name

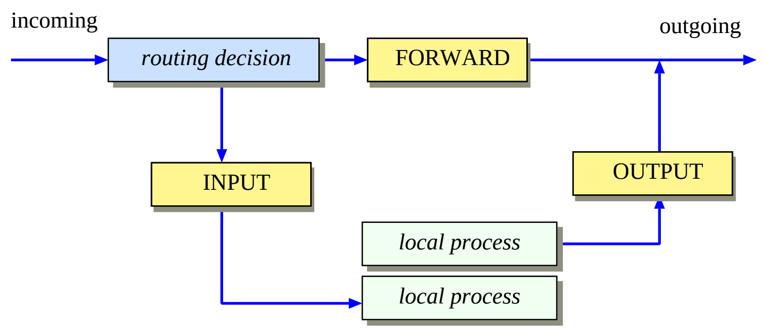


Figure 6.1: Packet flow through the filter table.

of a user-defined chain or one of the special values `ACCEPT`, `DROP`, `QUEUE`, or `RETURN`.

The `ACCEPT` rule accepts the packet and stops further processing. The `DROP` means that the packet should not be allowed, so it gets discarded without any message sent to the sender. `QUEUE` passes the packet to user-space filters, if they are supported by the kernel, and `RETURN` stops traversing this chain and resumes at the next rule in the previous (calling) chain. If the end of a built-in chain is reached or a rule in a built-in chain with target `RETURN` is matched, the target specified by the chain policy determines the fate of the packet.

Filter rules may be added to specified chains by means of the `iptables` command. The `-P` option defines a default chain policy (i.e. what to do with packets that do not match any specific rule in a specified chain), while the `-I` and `-A` options *insert* and *append* rules to a particular chain.

A minimalistic IPTABLES firewall based mostly on the `DROP` policy is shown in fig. 6.2. The `-P` option sets the policy for a specified table, the `filter` table being the default. After setting the default policies for all tables a specific `ACCEPT` rule is added to the `INPUT` chain with the `-A` option. In this particular case, the rule applies to all packets coming from the *eth1* interface whose source address belongs to the network *10.0.0.0/24*. These packets will be accepted while all other packets will be handled by the default `DROP` policy.

```
iptables -P INPUT DROP
iptables -P FORWARD DROP
iptables -t nat -P POSTROUTING DROP
iptables -t mangle -P OUTPUT DROP

iptables -A INPUT -i eth1 -s 10.0.0.0/24 -j ACCEPT
```

Figure 6.2: Simple IPTables firewall based on `DROP` policy

Another firewall example is shown if fig. 6.3. The configuration shown assumes that there are two network interfaces named *eth0* and *eth1* and *eth1* is the internal network. After defining default `DROP` policies the next two commands specifically allow TCP packets coming from the outer interface be forwarded if they are addressed for a SMTP server at host *156.17.1.5* or a WWW server at host *156.17.1.1*. The `-p` option specifies the IP protocol such as TCP, UDP, ICMP. Options such as `--dport` or `--sport` (destination and source ports) are protocol-specific and valid only for TCP or UDP, while `-s` and `-d` (source and destination IP) are valid for any IP protocol. The last rule allows all "returning" packets from the SMTP/WWW servers that traverse the firewall in the opposite direction.

```
iptables -P INPUT DROP
iptables -P FORWARD DROP

iptables -A FORWARD -i eth0 -p tcp --dport 80 -d 156.17.1.1 -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp --dport 25 -d 156.17.1.5 -j ACCEPT
iptables -A FORWARD -i eth1 -j ACCEPT
```

Figure 6.3: Simple IPTables firewall for SMTP and WWW services

The last example shows the typical difficulty with the typical stateless firewalls – it is necessary not only to define rules for packets coming to specific services, but also for the returning packets – the ones sent from the actual server back to the client. If local clients are allowed to connect to the outside world, things get even more complicated, but may be sorted out using the `--syn` option, that matches only the TCP packets that have the SYN flag set. Such packets are sent from the client to the server when the connection is being established, and may be used to determine the direction of the connection, i.e. where are the client and the server located. A sample setup allowing all connection from the local network and only the incoming connections for the SMTP server is shown in fig. 6.4.

```
iptables -P INPUT DROP
iptables -P FORWARD ACCEPT

iptables -A FORWARD -p tcp --dport 25 -d 156.17.1.5 --syn -j ACCEPT
iptables -A FORWARD -p tcp -s 156.17.1.0/24 -i eth1 --syn -j ACCEPT
iptables -A FORWARD -p tcp --syn -j DROP
```

Figure 6.4: IPTables firewall allowing only incoming SMTP and all outgoing TCP traffic

The first `-A` option adds a rule for the incoming SMTP connections the server. The second rule allows all connections originatin from the network *156.17.1.0/24* (the local network). The last rule drops all TCP packets trying to establish a new connection that did not match the previous two rules, and all the remaining packets (i.e. other protocols or TCP packets belonging to the already established connection) are passed through the default `FORWARD` chain policy, which has been set to `ACCEPT`.

This however still allows *stealth scanning* to take place. Modern network scanners such as NMAP [Lyo09] use this mode to bypass simple filtering rules by sending TCP packets in the established state instead of the typical connection requests with the SYN flag set. If the port is open, the target host will respond with the TCP packet with the RST flag set, and if the port is closed, we should get an ICMP type 3 response (i.e. "port unreachable"). The connection to such a service cannot be established as the firewall will still prevent it, but at least the attacker can be sure that there is a potential attack point that he may try to reach by some other means.

```
iptables -P INPUT DROP
iptables -P FORWARD DROP

iptables -A FORWARD -p tcp --dport 21 -d 156.17.1.21 -m state \
   --state NEW,ESTABLISH -j ACCEPT
iptables -A FORWARD -i eth1 -p tcp -d 156.17.1.21 --sport 20  \
   --state RELATED -m state -j ACCEPT
iptables -A INPUT -p tcp -m state --state INVALID -j LOG \
   --log-level error
```

Figure 6.5: Stateful firewall using IPTABLES

Prevention of stealth scanning is possible with stateful packet inspection as shown in fig. 6.5. The first rule after the typical policy setup allows all packets coming to the FTP server on port 21 that are either establishing a new connection (the `NEW` state) or continue a connection established earlier (the `ESTABLISHED` state). The `-m` flag tells IPTABLES to modify the `state` table each time a matching packet is passed. This means also the packet inspection, so if a `PORT` command is spotted inside a FTP packet, the forthcoming connection from server's port 20 (data transfer) will also be allowed as a `RELATED` packet. The last line deals with all packets that do not match any valid connection in the `state` table (the `INVALID` state) by dropping them and logging as network scan attempts.

For more examples on using IPTABLES and FWSNORT, see [Ras07]. A deep analysis of firewall design principles and different architectures can be found in [CZ95].

Chapter 7

Cryptography

There are numerous reasons why we cannot trust computer systems to store or transmit data securely and without being vulnerable to eavesdropping or malicious alteration. There are however techniques that can help us to achieve both security and privacy even when using a system that cannot be trusted. For this purpose, cryptography comes in handy.

Cryptography has many usages:

- Data encryption can protect data stored on your system even if other people have access to your computer.
- Encrypted data can be safely transmitted over the network and even if this is intercepted by somebody, the contents is still safe.
- Encryption can be used to detect accidental or intentional alteration of your data.
- Electronic signatures can be used to verify if the author of the document you have is really the one who you think it is.

However, it is not the universal remedy for all the problems we may have with data security, as it cannot prevent you or anybody else from *deleting* your files, or from leaking some sensitive information when exposed to some social engineering techniques.

Also, it is important to watch what is installed in a computer system you use – if an attacker is able to replace the encryption program with its own, he may have access to the plain text version of your files. And a lot depends on the quality of the cryptographic software being used – not only if it uses the right encryption algorithms and proper size of keys, but also how it handles temporary files, whether it leaves traces of its actions for anybody to read, etc.

7.1 Basics

Cryptography uses bit manipulation techniques to mangle the data in such a way, that its original contents cannot be seen anymore, but it can be restored when handled properly. Different methods for encrypting the data may be used, that differ in their quality and immunity to various attack methods. More secure algorithms usually require more processing power, so it is always a trade-off between the quality of protection that some algorithm can provide and the time that is needed to protect the data. Also – some algorithms are more suitable to particular applications and other ones may be better when the usage pattern changes.

The following sections describe different methods of operation that are used by encryption algorithms.

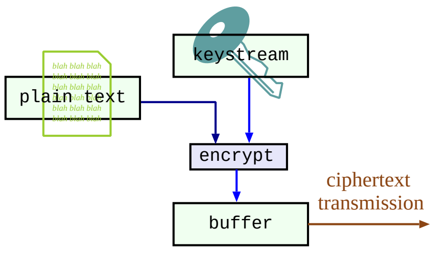
7.2 Stream ciphers

Stream ciphers convert plaintext to ciphertext bit by bit by XOR-ing the input stream with a continuously changing "running key". For that they need a key stream generator (see fig. 7.1(a)), which given some initial value of a limited length (such as 64 or 256 bits) creates a continuous never-ending stream of cipher key bits $k_1, k_2, ..., k_i$. These are in turn EX-ORed (added modulo 2) with the stream of plaintext bits $p_1, p_2, ..., p_i$, and that creates a ciphered bitstream $c_i = p_i \oplus k_i$ which may be sent over the insecure channel. After reception, the plaintext is calculated as $p_i = c_i \oplus k_i$ – the key bitstream on both ends has to be exactly the same.

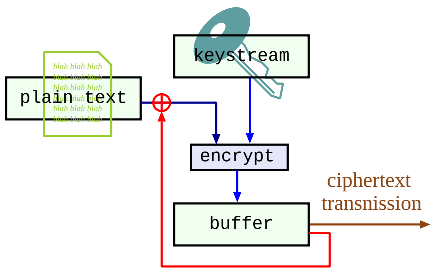
For cryptoanalytical immunity, the system should not start from a "zero" state after a reset, but from a changing initialization vector or a different key.

Because of this reset condition, stream ciphers are better suited for continuous bitstream transmissions between 2 parties, rather than occasional message transmission.

Although it may seem that the keystream generated from a key looks like a random noise, it is important to remember, that the keystream generator is a periodic function. Its period is very long, as its length is the key factor to the security of the system.



(a) ECB mode



(b) CBC mode

Figure 7.1: Stream ciphers – basic encryption modes

7.3 Block ciphers

These types of algorithms convert plaintext to ciphertext in blocks of a fixed length (depending on actual algorithm, e.g. 64 or 128 bits). Even if we need to send just few bits of data (e.g. a keystroke or a single byte) we need to fill the whole block, as the encryption process will manipulate many bits at once. They are easier to implement than stream ciphers, but their minimal block length requirement may be a disadvantage – e.g. in a character-oriented transmission of an interactive terminal data.

Block ciphers may operate in different modes. The earliest modes (ECB, CBC, OFB, and CFB have been defined in [Nat80] for DES which was the standard block cipher at that time, but the modes apply to any block cipher. These have been revised as new block ciphers appear and the CTR mode has been introduced in [Dwo01] when AES has been accepted as the new standard block cipher. Some other modes, such as XTS-AES or CTS are either cipher-specific or experimental and not yet approved for wider usage. Table 7.1 lists the most common block cipher modes of operation.

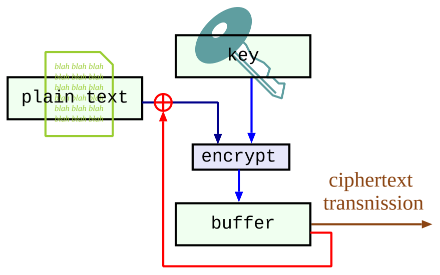
The ECB mode of operation is the simplest one, but does not perform well for repeatable data. If the same plaintext is encrypted in different blocks of input data, the resulting blocks will be the same, so even if the encryption cannot be broken, there is a clear information that this data has been sent before. Assuming some common patterns (streams of zeros in various common formats of documents or some other common patterns in picture formats) may further lead to simplified cryptoanalysis. It is also possible for an attacker to record the encrypted transmission and send it later (this is called the *replay attack*) – for example, sending again the same data from the ATM machine without even understanding what is being sent may be still enough to repeat the recorder transaction and successfully withdraw the money.

CBC mode (as well as other modes) helps to overcome this kind of attack by chaining all blocks together – i.e. each block depends on all other blocks that has been sent so far, so encrypting the same plaintext several times results in different cipherblocks. It is simply done by XOR-ing each plaintext block fed to the encryptor with the cipherblock obtained in the previous round. This however means that if a single encrypted block is lost

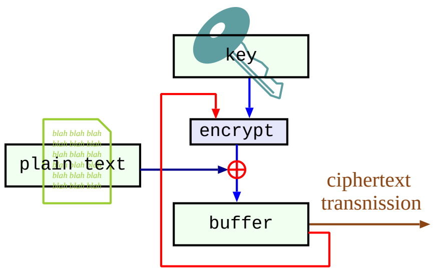
Mode	Description
ECB Electronic Code Book	Each plaintext block is encoded independently of other blocks
CBC Cipher Block Chaining	Instead of just the plaintext, the algorithm is fed with the XOR of the plaintext and the previous ciphertext.
CFB Cipher Feedback	Input is processed n bits at a time (les then or equal to the block size). Previous ciphertext is used as input to produce the pseudorandom data which is then xored with current plaintext.
OFB Output Feedback	Similar to CFB, except the input is not the final ciphertext, but the output of the DES encryptor (i.e. before XOR-ing it with the plaintext)
CTR Counter	Every plaintext block in XORed with an encrypted counter, which is incremented with each block
CTS Ciphertext Stealing	Used in addition to ECB or CBC for padding the last plaintext block with the high order bits of the second to last cipherblock (stealing it from the output). The new full last block is then encrypted and exchanged with the stolen cipherblock, truncated by removing the stolen bits, so that the overall message length does not change.

Table 7.1: Block cipher modes of operation

or altered during transmission, it will be not possible to decrypt the rest of the transmission. A little modification of this mode called *Output Feedback Mode* (OFB, see fig. 7.2) helps overcoming this problem by moving where this chaining information is attached – if a single block is malformed, it will be lost and impossible to decrypt, but the synchronisation is regained with the next block.



(a) CBC mode



(b) OFB mode

Figure 7.2: Comparison between CBC and OFB block encryption modes

7.4 Symmetric cryptography

Encryption in symmetric key cryptography takes some arbitrary length input, called a *plaintext* and using a *key* (which is a small piece of data kept in secret) converts it to *ciphertext*, which does not resemble the original plaintext in any way. In order to get the plaintext back from the ciphertext, the reverse process has to be applied, again – with the same key as the parameter.

7.4.1 Algorithms

Below is a short summary of the most popular symmetric key algorithms:

ROT13

Not a real encryption system, but still used in Usenet news or discussion groups to obscure the contents of some jokes or other material considered offensive. This is a simple substitution code, doing a 'shift by 13 positions', i.e. *a* becomes *n*, *b* becomes *o*, etc. (and also *n* becomes *a* and so on...). For example: `Unpxref bs gur jbeyq, havgr! Qvfyrpgvpf bs gur jbeyf, hagvr!`

crypt

Traditional UNIX system encryption, based on German Enigma engine, using variable length key. It contains a checksum, so there are programs to automatically try decrypting and verifying a message, so it is not secure. To add to the confusion, there is a secure one-way hash function called crypt, used for password encryption, so do not confuse the two.

DES – Data Encryption Standard

Developed in 1970s by National Bureau of Standard Technology in USA and IBM. Uses 56-bit keys and is not considered safe today. Introduced in [Nat77], current definition in [Nat99].

Triple DES or 3DES

DES applied three times with different keys each time (as a pipeline: encrypt with key 1, decrypt with key 2, encrypt again with key 3), giving the actual key size of 168 bits. Much more secure, but still vulnerable to some attacks known nowadays.

RC2, RC4 and RC5

All developed by Ronald Rivest around 1990s, kept secret for some time, then revealed in anonymous postings to Usenet in 1994-1996. All appear to be reasonably strong. RC2 and RC5 are block ciphers, RC4 is a stream cipher. RC2 and RC4 allow keys of size 1 to 2048 bits, RC5 – no limit. All "export" versions until recently were limited to 40-bit keys.

IDEA

International Data Encryption Algorithm, developed in Zurich, then patented. Uses 128-bit key. Used by PGP program, SSH and many other security applications. There are implementations freely available outside US, although software patents may make it unusable inside US.

Skipjack

A secret algorithm developed by NSA using a 80-bit key.

Blowfish

Skipjack alternative. Variable length key, between 32 and 448 bits. Developed in open-source community in 1993 and still being checked for strength. Already used in many security programs.

AES – Advanced Encryption Standard
AES Current standard for symmetric encryption, adopted in 2001 by NIST as a DES successor (FIPS 197 standard [FIP01]), after a 5-year standardization process by means of a public competition. Originally published as *Rijndael*. Uses fixed length keys of size 128, 192 or 256 bits and block size of 128 bits. The block is organized as 4×4 bytes, on which *SubBytes*, *ShiftRows* and *MixColumns* operations are done in a number of rounds (from 10 to 14, based on the key size). Due to the relatively good performance, AES can be used on lots of different hardware, including embedded platforms and even slow 8-bit smartcards.

7.5 Public Key Cryptography

Symmetric key cryptography is fine, when we can securely exchange keys and prepare for the communication beforehand. In Internet applications where we want to establish secure transmission with the other side that we have connected for the first time, this is not feasible – in order to safely exchange the encryption key, we need to have some encrypted/safe communication channel in the first place, otherwise someone might steal out key and decode the following communication.

In *asymmetric* or *public key* it is possible to exchange information over the public channel in such a way, that some of it remains secret, i.e. known only to the communicating parties, and cannot be derived from what has been seen on the public channel. This comes at a cost of a much greater complexity of the algorithms and much slower implementations that the symmetric cryptography uses, but consequently, it can be used to safely exchange some information which may become a temporary *session key* for a symmetric encryption algorithm, so the impact on the overall communication speed will be minimal.

7.5.1 Diffie-Helmann algorithm

This algorithm provides a way of safe exchange of a shared secret key over the insecure communications channel and is one of the earliest examples of asymmetric cryptography. The secret established in this way may be later used as a symmetric key in a subsequent communication that uses a symmetric key cipher.

Mathematical foundations are based on the relative difficulty of calculating discrete algorithm and easiness of calculating powers and modulo divisions.

Two parties need to agree on two large integer numbers n and g, such that $1 < g < n$. These numbers do not need to be secret. Two users – Alice and Bob can exchange them over a public channel.

1. Alice chooses a large random integer x and calculates:

$$X = g^x \bmod n \tag{7.1}$$

2. Bob chooses a large random integer y and calculates:

$$Y = g^y \bmod n \tag{7.2}$$

3. Alice sends X to Bob, Bob sends Y to Alice. Both x and y are kept secret.

4. Alice calculates

$$k = Y^x \bmod n \tag{7.3}$$

5. Bob calculates

$$k' = X^y \bmod n \tag{7.4}$$

From (7.3) and (7.2) we get

$$\begin{aligned} k &= Y^x \bmod n \\ &= (g^y \bmod n)^x \bmod n \\ &= ((g^{y^x}) \bmod n) \bmod n \\ &= g^{xy} \bmod n \end{aligned}$$

In a similar way from (7.4) and (7.1) we get

$$k' = g^{xy} \bmod n$$

so Alice and Bob are now in possession of the same number $k = k'$, and nobody who has overheard the transaction can reconstruct this number, as x and y are kept secret and only n, g, X and Y were made available.

7.5.2 Message Digest Functions

Message digest functions are also called *hash functions* or *cryptographic checksum functions*. They take the input of arbitrary length and produce a fixed size small amount of data in a predefined deterministic way, so if the same input is processed again, it will generate exactly the same output as the first pass, but as the function works one-way only, it is not feasible to recreate the original text from the output, or even find another text, that gives the same output. Another basic feature of all hash functions is good propagation of changes throughout the algorithm, so even a small change in input data changes significant part of the output (or the whole of it), so it is not possible to predict what to change and retain the same output value.

Typical output of the hash function varies between 90 and 256 bits which makes a brute force attack impractical and hard to perform.

Given the above characteristics, the hash functions are mainly useful in digital signature and data integrity checking applications. Cryptographic signing of a large document could be a very time consuming process. So to speed things up, a document digest (a *hash*) is created first through a hash function, and only this hash (which is shorter from the original by orders of magnitude) is then signed. When signature is to be checked, the document in question is parsed (again) through the same hash function to obtain its digest first, and then the calculated digest is compared to the signed and verified digest. If they do not match, or if the digest signature is invalid, the document might have been tampered with, otherwise it must be the same as the original.

There are various hash functions differing in speed and their potential usability for different tasks. Below is a summary of the most commonly used functions:

MD2, MD4, and MD5

Most widely used functions, producing 128 bit hash values. MD2 published in RFC 1319 [Kal92], with no known weaknesses, but rather slow. MD4 published in RFC 1186 [Riv90] and RFC 1320 [Riv92a] – fast, compact and optimized for little-endian machines (such as Intel/Zilog processors). Some potential attacks on it can be possible, which lead to the development of MD5 (RFC 1321 [Riv92b]) – a bit slower, but much more secure and the most widely used so far.

SHA

Related to MD4 algorithm, but produces 160 bits of output.

HAVAL

Modification of MD5, with output varying between 92 and 256 bits and adjustable number of internal rounds. May be used to produce hash values faster than MD5 (but possibly weaker) or slower but more securely.

SNEFRU

Produces 128 or 256 bits of code. Can make variable number of internal rounds, but analysis shows that 4-round SNEFRU has some basic weaknesses, where the recommended 8-round SNEFRU is significantly slower than MD5 or HAVAL.

7.5.3 Digital signatures

Public key cryptography may also be used for message authentication, i.e. tampering prevention. In order to encrypt a message using an asymmetric system, one should use the recipient's public key for the encryption, so that it may only be decrypted with a matching private key.

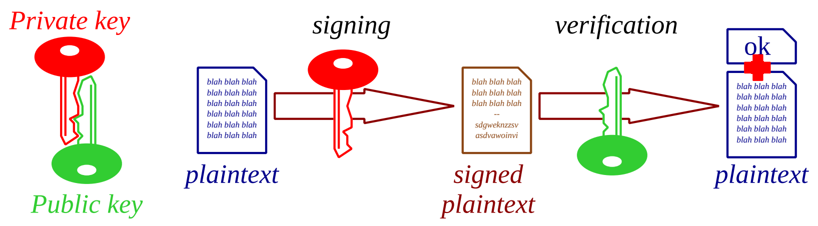


Figure 7.3: Digital signature

Let's consider what happens if a private key is used for encryption (see fig. 7.3). The public key needed to decrypt such a message is, by definition – public, so everybody in the world may get access to it and decrypt the message. In terms of privacy, the encryption is worthless. But the fact, that the public key can be used for the decryption means, that the matching private key had to be used for "encrypting" this message, and as the key is *private* and known only to its owner, it must have been *signed* by him.

7.5.4 Algorithms

Asymmetric cryptography algorithms vary in their usage – they may be used for secure key exchange over an insecure channel, for creating signatures and verifying data integrity of messages sent through e-mail or exchanged over the untrusted media. Below is a summary of the most popular algorithms of public key cryptography:

Diffie-Hellman

A system for exchanging keys between two or more parties. Not an encryption system itself, but a way of secure exchange over a public channel. Widely used as a basis for establishing session key using private key cryptography.

RSA

Named after its developers from MIT university: Ronald Rivest, Adi Shamir and USC Professor: Leonard Adleman. May be used for encryption or for making signatures. Any length is available for keys (secure keys start with 512 bits).

ElGamal

Can be used in a similar way as RSA. Based on exponentiation and modular arithmetic.

DSA

Digital Signature Algorithm. Developed by NSA, and used for signatures only (although it is possible to use it for encryption too). Sometimes called *DSS*. Any key length is possible, but adopted standards use lengths of 512-1024 bits.

Chapter 8

Secure programming

Many security problems come from improper configuration or unwanted interaction between different programs, but may of them are actually consequences of errors in software development – laziness of the programmers, lack of experience, tight schedules and deadlines that have to be met, or just the underestimated role of software testing.

The "Top 25 most dangerous software errors" document [Cor10] lists the most dangerous programming problems considered today that may lead to serious software vulnerabilities. The short summary of these errors is shown in table 8.1. It includes various aspects of programming, from specifics of various programming languages, to WEB-programming and constructing database queries, so the given scores are influenced mostly by the impact on the overall security, which puts a clear stress on WWW access and parts of programming techniques which are WEB-related. Another notable WEB-oriented document s the OWASP Testing Guide [The09]. Nevertheless, it is equally important to avoid the well-known problems when writing programs that will be compiled and run in any operating system, no matter if they are WWW-accessible or run as standalone programs.

When considering shell programming and writing standalone programs, the most common errors include problems that allow exploiting set-uid programs, such as:

- buffer overflows
- signal disposition and process group setup
- *printf*() format string attacks
- relative path in *exec*()
- /*tmp* or other publicly writable directory race conditions

There are many general rules for safe programming and techniques. One of the very basic ones requires processes to operate on the least possible level of required privileges and dropping them whenever they are not needed. Other ones assume that both programmers and users can make errors, and it is the responsibility of a computer program to properly deal with all kinds of problems with data sanity checks, for example by checking various assertions throughout the execution of the program.

The *defensive programming* techniques cover the whole process of software design that covers such aspects, as writing down the required functionality, splitting the whole program into functional objects, reducing the design complexity, writing and testing pieces of software as they are written, writing usable documentation, and finally – writing software that is reusable, bug-free, predictable and running properly in spite of its unforeseeable usage.

Proper software writing techniques include:

- dropping extra privileges when they are not needed,
- 'black list' of functions that should *never* be used in suid programs,
- adhering to the specification of the program,
- keeping in mind the re-usability of code and proper documentation,
- proper testing of all components of a written program.

	Score	Name
1	346	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
2	330	Improper Neutralization of Special Elements used in an SQL Command (*SQL Injection*)
3	273	Buffer Copy without Checking Size of Input (*Classic Buffer Overflow*)
4	261	Cross-Site Request Forgery (*CSRF*)
5	219	Improper Access Control (*Authorization*)
6	202	Reliance on Untrusted Inputs in a Security Decision
7	197	Improper Limitation of a Pathname to a Restricted Directory (*Path Traversal*)
8	194	Unrestricted Upload of File with Dangerous Type
9	188	Improper Neutralization of Special Elements used in an OS Command (*OS Command Injection*)
10	188	Missing Encryption of Sensitive Data
11	176	Use of Hard-coded Credentials
12	158	Buffer Access with Incorrect Length Value
13	157	Improper Control of Filename for Include/Require Statement in PHP Program (*PHP File Inclusion*)
14	156	Improper Validation of Array Index
15	155	Improper Check for Unusual or Exceptional Conditions
16	154	Information Exposure Through an Error Message
17	154	Integer Overflow or Wraparound
18	153	Incorrect Calculation of Buffer Size
19	147	Missing Authentication for Critical Function
20	146	Download of Code Without Integrity Check
21	145	Incorrect Permission Assignment for Critical Resource
22	145	Allocation of Resources Without Limits or Throttling
23	142	URL Redirection to Untrusted Site (*Open Redirect*)
24	141	Use of a Broken or Risky Cryptographic Algorithm
25	138	Race Condition

Table 8.1: Top 25 programming errors according to [Cor10]

8.1 Buffer overflow

Most of currently used programming languages provides support for tables or other structures whose elements can be iterated, as well as local variables and recursion, which impose a certain model of architecture of the memory usage in a program. First of all, it is divided into separate segments: the program code, the data and the stack. As some variables are stored on the stack (i.e. variables declared as *local* inside a function or a code block), it is really important to check boundaries of any write operations that may operate on locally declared tables or vectors. Let's consider a function definition as shown in fig. 8.1.

```
int sample(int n, char *ptr) {
    int i;
    char tab[100];

sprintf(buf, "Read %d samples from %s\n", n, ptr);
...
}
```

Figure 8.1: Buffer overflow problem

This function may be called with any parameters, so when the *sprintf*() function gets called it will happily copy whatever is passed as the second parameter to the buffer allocated on the stack. The buffer's size is set to 100 bytes and some of these will be allocated for the printed text, so effectively, there remains about 80 bytes for the remaining data. If the string passed by the `ptr` parameter is longer, the *sprintf*() will not stop, but continue copying the bytes past the end of the `tab` buffer, overwriting whatever is there – the variable `i`, the function frame and the function return address, so when finally the function will end, the program execution will not be transferred to the place where the function was called, but to some random memory location.

Buffer overrun problem gained widespread attention after being published in the Phrack magazine in 2001 [Ale01] but it seems it is still one of the most important, but still overlooked, problems in writing applications.

The problem may manifest itself in many different situations, whenever some data is being copied between memory locations without the proper boundary checking. It is especially dangerous if the data is copied from standard input (*stdin*), program parameters (*argv*) or environment variables (*getenv*()), as it enables an attacker to easily control what is being copied to the overwritten stack and actually exploit the error by supplying a specially crafted *shellcode* – a piece of assembly code placed on the stack which when executed, runs a shell, possibly with the privileges obtained from the program with a buffer overrun problem.

The actual copying in a dangerous way may be done by different means, some of which include:

- using *strcpy*() instead of *strncpy*(),
- using *gets*() instead of *fgets*(),
- improper usage of *sscanf*(),
- improper usage of pointers and wrong loop ending conditions.

History shows that most of the programs had this kind of problem. A lot of open source programs have been cleaned from dangerous system calls once buffer overflows gained widespread attention, but still a lot of problems may exist undetected in closed source programs or even open source that is not maintained very actively. In some cases it may be even easier to find an exploit for a buffer overrun bug hidden in a program than fix it, as it is quite simple to analyse its binary version, looking for calls to functions such as *gets*() or *strcpy*(). They need to be listed explicitly in order to allow dynamic linking of system libraries. Any calls to these functions found in a program are a clear invitation for further analysis and writing an exploit.

8.2 File permissions

File permissions play an important role in securing access to various system resources as well as individual user files and should not be overlooked when writing any application. Very often temporary files are needed by programs, which are usually created in the */tmp* directory – if they are readable by anybody, some internal state of the program may be revealed, and if the programmer did not care about the *umask* value and created a file with world-write permissions, anybody else may disrupt the execution of the program by modifying such a temporary file.

The basic rules about file permissions when writing programs is to keep the permissions to the minimum – i.e. only enough that the program works properly and within the defined specification, but not more. So it may respect the default *umask* (i.e. defined by the user running a program) when creating output files, but for temporary files or any other transient operations it is safer to set the *umask* to `066` so that anybody else will not be able to read/write these files. If the program needs special privileges and is run in suid mode, even more care has to be taken for accessing files and producing aby output. Access to files that are supposed to be world-readable should be done only after dropping special privileges (by using the *seteuid*() function). Failing to do so may result in *race conditions* (see section 8.6) or problems related to symbolic links usage.

Some well known permission problems are specific to particular versions of the operating system and some are rather the effect of lazy programming and may appear everywhere. Older systems allowed deleting files owned by other people if they were placed in publicly writable directories, such as */tmp*, as the world-write access bit set for such a directory allows it. Modern systems still may allow this, but the sticky bit (`+t`) set on the directory makes these operations safe, i.e. disallowing any file manipulation of files owned by other users. This also helps reducing risks of race conditions, but still some other measures have to be taken to prevent them from happening.

Default `umask` needs to be changed especially in servers – the programs that once executed, will run endlessly in a loop, waiting for clients to connect. Also it is worth knowing, that if a server is run from the system startup scripts (located in */etc/rc?.d* directories), its *umask* value will probably be set to `0`, so a server program should not assume anything and set this value to some more appropriate value, regardless of the initial value.

8.3 Default directories and relative paths

For programs that run as a service or a system *daemon* (i.e. once started, they do not exit, but service new clients as they appear) it is usually desirable to provide some way of reinitialisation or re-checking of the initial state. Typically, if such a process receives a `SIGHUP` signal, it will reopen all log files, re-read its config file and generally – restart its operations. As the program name is kept in `argv[0]`, the simplest way is to use this name in the *execv*() or *execvp*() call (so that all initial parameters are also preserved). However, if the program has been started with a relative path, or from a path which was a symlink in the */tmp* directory, this may lead to actually starting a completely different program if that symlink has been changed in the meantime.

In suid programs care must be taken to drop all special privileges before proceeding with any of the *exec*() system calls – if the privileges are really needed, the *exec*() function will take care of them (i.e. grant them on execution of a file read from the disk), while not dropping them may lead to using them in a program that should be run as normal user.

8.4 Passing parameters and quoting

When one process calls another as a subprocess or as external application, special care has to be taken to passing parameters. In most cases spaces are treated as a separator between parameters, minus sign is treated as an option indicator, and some characters such as < or * may have a special meaning when interpreted by shell. Single and double quotes group parameters changing their interpretation and other characters (e.g. %) will indicate special coding when used in CGI-encoded strings. For these reasons it is vital to realize what kinds of programs are passing parameters and how they should be quoted and possibly cleaned of dangerous characters, so that their interpretation is really what was intended.

8.5 Uninitialized variables and default values

Languages such as PERL, AWK, PHP and shell scripts allow using variables which come into existence by just referencing them. Such variables are usually initialized to 'zero' or null string, but it is easy to forget about the proper initialization or confuse the uninitialized zero value with an intentional zero value.

Another problem comes from the fact, that the type of such a variable is automatically assigned at the first use of the variable, so if it is first used as an integer, but later referred as string, the results may be not what the program author intended.

Other languages (such as C, C++) still allow usage of uninitialized values – it is possible to declare a variable then use its (undefined) value before assigning it first. Some compilers produce compilation-time warnings in such situations, but it is not always possible to detect such problems. Assuming that such a variable will have a value of 0 may be true 99 times out of 100, but leads to unpredictable results.

8.6 Race condition

Race condition happens when two or more processes try to access some file or data in a way, that is dependent on the actual timing of these processes (and thus – unpredictable) rather then on some well-defined conditions.

Examples may include accessing a region of shared memory or writing to a file by several processes. To avoid the possible data corruption the operating system provides tools such as semaphores and file locking and in general – a well-defined interface that guarantees atomicity of critical operations when needed. Semaphores allow mutual exclusion of processes for sections of the program that need to be executed atomically – i.e. uninterrupted, from the point of view of other processes that may want to modify the same region of memory, or a record in a database. Threaded applications may use semaphores or monitors and conditional variables for that. For file access, file locking mechanisms may be used, such as *flock*() or *lockf*() which grant exclusive access to the whole file or its fragment (defined by the starting offset and length).

However, one of the moments that is too often forgotten is the moment of actually opening a file, especially when a file is opened for writing, and even more important – by processes running with special privileges (i.e. suid programs).

Typical abuse involving race conditions includes symbolic links created in temporary directories. A symlink is a special file that points to another file in a filesystem (potentially: a different file system). Every operation on a symlink (i.e. opening it, reading, writing or overwriting) works actually on a file that it points to, except *unlink*() and *rename*() (or `rm` and `mv`) which removes/renames the link, not the target file. Removing a file may cause the link to become invalid (pointing to a nonexistent file), but removing a link and then immediately creating one pointing to another file is a potential security threat.

Consider a typical situation shown in fig. 8.2, when a process wants to open a temporary file in */tmp* directory. If the file already exists, it will be overwritten. If the running process is running with $euid = 0$, it may overwrite any file in */tmp* directory. If that file happens to be a symlink, it may actually overwrite any file anywhere. So one might want to ensure that the file does not exist when we try to open it for writing, or if it exists, it is not a symlink.

The process first tries to check if the file exists with a *stat*() system call (fig. 8.3). If the call succeeds (returning a file info) or fails for any reason other than that the file does not exist, the error message is printed using *perror*() and the program is aborted. Otherwise the file is created with *fopen*() system call.

The problem now lies in a tiny moment between these two operations. There is no guarantee that there will not be a context switch just before the *fopen*() call, and if that happens, another process may create a file or a symlink with the given filename. Even if we open the file in *append* mode, just in case, then check again after opening if it is not a symlink and possibly reopen with

```
FILE * fp=fopen("/tmp/temp1", "w");
if (fp==0) {
  fprintf(stderr, "Error creating temporary file\n");
  exit(1);
}
fprintf(fp, "........");
```

Figure 8.2: Symlink abuse threat

```
#define FNAME "/tmp/temp1"
struct stat buf;
if (stat(FNAME, buf)==0 || errno!=ENOENT) {
    perror("creating temp file");
    exit(1);
}
FILE * fp=fopen(FNAME, "w");
if (fp==0) {
  fprintf(stderr, "Error creating temporary file\n");
  exit(1);
}
fprintf(fp, "........");
```

Figure 8.3: Symlink race condition

a different name, the problem still remains. The only solution is to actually use a system call that will atomically create the file if it did not exist, or fail if the name already existed. For that purpose either *open*() or *creat*() system calls may be used, as both of them allow flags `O_CREAT` and `O_EXCL`. Using both of these flags ensures creation of the file by the calling process or an error reported if for some reason the file could not be created. Later the file descriptor of type `int` may be transformed to a `FILE*` descriptor needed in stdio-style functions by calling *fdopen*(), as shown in fig. 8.4.

```
#define FNAME "/tmp/temp1"
int fd=open(FNAME, O_CREAT | O_EXCL, 0600);
if (fd<0) {
    perror("creating temp file");
    exit(1);
}
FILE *fp=fdopen(fd, "w");
fprintf(fp, "........");
```

Figure 8.4: Safe creation of a temporary file

Some final notes on writing secure programs that avoid race conditions:

- Use *creat*() or *open*() with a `O_TRUNC` parameter when creating new files.
- When creating files in publicly writable directories, such as */tmp*, ignore the *mktemp*() function and use nstead *open*() with `O_CREAT` and `O_EXCL` options to create the file, but also ensure the file did not exist earlier in one atomic operation.
- Use *stat*() on a file descriptor of the already open file, instead of *fstat*() or *lstat*() and giving the file reference by file name.

8.7 Further reading

Command line problems and proper interpretation of quoting and escaping special characters is discussed in [Wau01] and in the book [McG06]. *Secure programming cookbook* [VM03] provides a lot of examples of how to write proper code. *Secure coding: principles and practices.* [GV03] describes a lot of bad and good practices of code development as well as security architecture principles and testing methodology.

Bibliography

[Ale01] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, April 2001. http://www.phrack.org/archives/49/.

[Bau05] Michael D. Bauer. *Building secure servers with Linux*. O'Reilly & Associates, Inc., second edition, 2005.

[BH10] M. Brown and R. Housley. Transport Layer Security (TLS) Authorization Extensions. RFC 5878 (Experimental), May 2010.

[BSB05] Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes. *SSH: The Secure Shell: The Definitive Guide*. O'Reilly & Associates, Inc., second edition, 2005.

[Che92] W.R. Cheswick. An evening with Berferd, in which a cracker is lured, endured, and studied. In *Proceedings of the Winter USENIX Conference (San Francisco)*, January 1992.

[Cor10] The MITRE Corporation. Top 25 most dangerous software errors. http://cwe.mitre.org/top25/index.html, December 2010.

[CZ95] D. Brent Chapman and Elizabeth D. Zwicky. *Building Internet Firewalls*. O'Reilly & Associates, Inc., September 1995.

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999.

[DeS86] A.L. DeSchon. Survey of data representation standards. RFC 971, January 1986.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[Dwo01] Morris Dworkin. Recommendation for block cipher modes of operation, December 2001. NIST 800-38A, http://csrc.nist.gov/publications/nistpubs/800-38A/nist-sp-800-38E.pdf.

[Fed85] Federal Information Processing Standards. Passwords usage, May 1985. FIPS 112, http://www.itl.nist.gov/fipspubs/fip112.htm.

[FIP01] FIPS. Advanced Encryption Standard (AES), November 2001.

[Gar08] Luis Martin Garcia. Programming with Libpcap – Sniffing the network from our own application. *Hakin9*, 3(2), February 2008. http://42.pl/u/2wYl_libpcapHakin9.

[GSS03] Simson Garfinkel, Gene Spafford, and Alan Schwartz. *Practical Unix & Internet Security*. O'Reilly & Associates, Inc., third edition, 2003.

[GV03] Mark Graff and Kenneth R. Van Wyk. *Secure coding: principles and practices*. O'Reilly & Associates, Inc., 2003. http://etutorials.org/Programming/secure+coding/.

[Hof02] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207 (Proposed Standard), February 2002.

[JLM09] Van Jacobson, Craig Leres, and Steven McCanne. Tcpdump. http://www.tcpdump.org/, 2009.

[Kal92] B. Kaliski. The MD2 Message-Digest Algorithm. RFC 1319 (Informational), April 1992.

[Lyo06] Gordon Lyon. Top 10 password crackers. http://sectools.org/crackers.html, 2006.

[Lyo09] Gordon Lyon. *Nmap Network Scanning*. Nmap Project, January 2009. http://nmap.org/book/.

[McG06] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, Feb 2006.

[Nat77] National Institute of Standards and Technology (NIST). DES encryption standard, January 1977. FIPS 46.

[Nat80] National Institute of Standards and Technology (NIST). DES modes of operation, December 1980. FIPS 81, http://www.itl.nist.gov/fipspubs/fip81.htm.

[Nat99] National Institute of Standards and Technology (NIST). DES encryption standard, October 1999. FIPS 46-3, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[Ras07] Michael Rash. *Linux firewalls: attack detection and response with* iptables*,* psad*, and* fwsnort. No Starch Press, San Francisco, CA, USA, 2007.

[Riv90] R.L. Rivest. MD4 Message Digest Algorithm. RFC 1186 (Informational), October 1990. Obsoleted by RFC 1320.

[Riv92a] R. Rivest. The MD4 Message-Digest Algorithm. RFC 1320 (Informational), April 1992.

[Riv92b] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[RM06] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), April 2006.

[RRDO10] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010.

[Sch95] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Sta03] William Stallings. *Cryptography and Network Security, Principles and Practice*. Prentice Hall, 3rd edition, 2003.

[Sto90] Clifford Stoll. *The Cuckoo's Egg*. Pocket, 1st edition, 1990.

[The09] The Open Web Application Security Project. OWASP testing guide. `http://www.owasp.org/`, Aug 2009.

[The10] The Open Web Application Security Project. OWASP top 10 – the ten most critical web application security risks, 2010 release. `http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`, Jul 2010.

[Ven95] Wietse Venema. TCP Wrapper network monitoring, access control, and booby traps. `ftp://ftp.porcupine.org/pub/security/tcp_wrapper.pdf`, July 1995.

[VM03] John Viega and Matt Messier. *Secure programming cookbook for C and C++: Recipes for cryptography, authentication, networking, input validation and more*. O'Reilly & Associates, Inc., 2003. `http://etutorials.org/Programming/secure+programming/`.

[Wau01] Tim Waugh. When is a command line not a line? `http://freshmeat.net/articles/view/337/`, 2001.

[wir10] Wireshark. `http://www.wireshark.org/`, 2010.

Index