Developing Secure Software Applications
DRAFT

Frank Piessens

November 28, 2005

Contents

Part I

Introduction

Chapter 1

Introduction

Software is often an *enabler* of new possibilities. Software running on an e-commerce site enables customers from all over the world to purchase goods. Software running on a company web server enables people to look up information about the company or its products. Software on your PC enables you to manage and efficiently use your documents, images, digital audio files and so on. The main driving factor for developing or installing software is typically that the software will be an enabler for some useful functionality.

On the other hand, by making information, resources, procedures or other assets more easily available, you also increase the risk of abuse or damage. An e-commerce server could potentially be abused to buy goods for an incorrect price, or to buy goods on some other customers account. Through the web server software, a hacker could modify company information, making the company look ridiculous on the Internet. Your privacy could be seriously compromised if a hacker can work his way into the software on your PC and browse through your personal files.

So while we use software for its useful functionalities, we also often rely on the software to enforce certain rules. The e-commerce server should make sure that any purchase is always correctly paid for by the purchasing party. The web server should make sure that it only gives read-access to the information it serves to the web. And the software on your PC should only give you access to your personal files.

Software security is about enforcing these correct rules of usage. These correct rules of usage are often referred to as the *security policy* that the software implements. Hence, computer security deals with the prevention or detection of actions on a computer system that are not allowed by the security policy. This book will teach how to build software in such a way that you can be reasonably sure that it will enforce correct use.

In this chapter, we start by giving a broad overview of the field of IT

security, and we explain on what parts of this large field this book will focus.

1.1 Security in IT systems

Security in IT systems is basically about *risk management*. On the one hand, owners wish to maximise the usefulness of information and IT-related assets by making them easily available to legitimate users. On the other hand, easy availability might lead to abuse or damage and thus increases the risks to these assets. Securing an IT system is about imposing countermeasures to reduce these risks. But such countermeasures might negatively influence usefulness or availability of assets. And countermeasures always come with a certain cost. Hence, in most practical cases, risks will never be completely eliminated. Instead, they will be reduced to an acceptable level, for an acceptable cost, and an acceptable impact on the usability of the protected assets.

The most important concepts in IT security are very well summarized in the diagram in figure 1.1, extended from a similar figure in the Common Criteria, a well-known standard on IT security evaluation ([22]).

So security in general is about imposing countermeasures to reduce the risks caused by identified threats to a system.

1.2 Different viewpoints on security

The kind of countermeasures that one can take depend heavily on the context in which one operates, and not all countermeasures are software related. If, for example, you are the security officer in a company, responsible for safeguarding company confidential information, you will need to worry about physical security of company buildings and screening of new personnel as well as about IT security. In this course, we will not discuss these non-IT related aspects of security. For a broad view on information security management, including non-technical issues, the reader can consult ISO IS17799 ([23]), a standard that spells out best practices for information security management.

But even if we restrict ourselves to purely IT-related aspects of security, the countermeasures one can take depend on the context. In this section, we will discuss security from two different viewpoints: the viewpoint of a network administrator, and the viewpoint of a software developer.

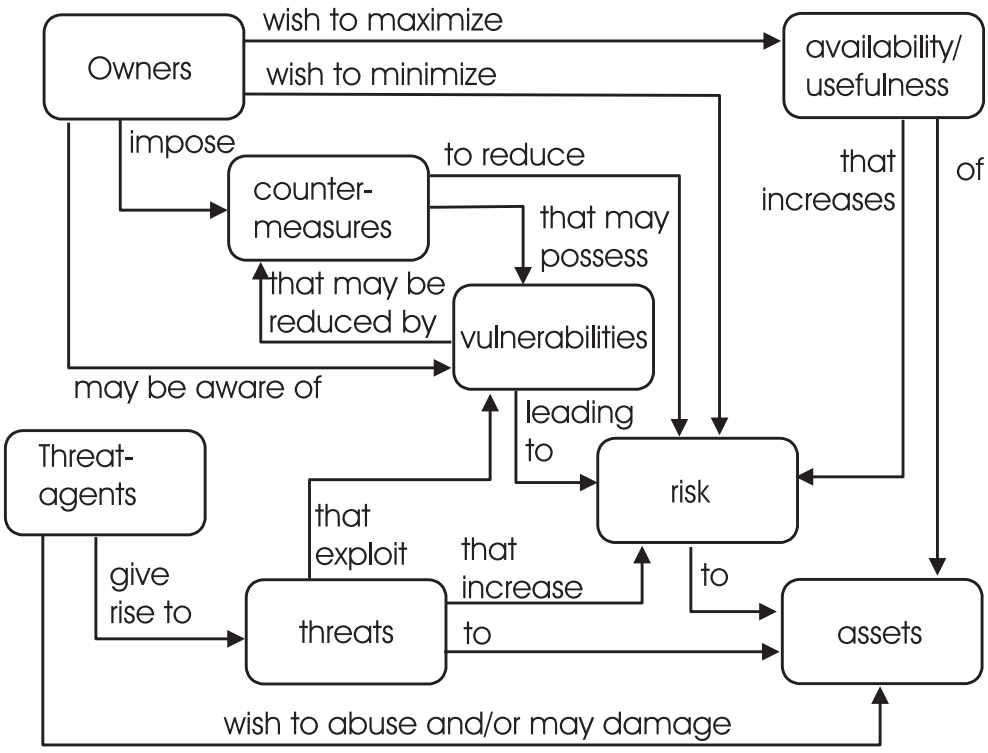

Figure 1.1: IT security concepts (adapted from [22])

1.2.1 Security for the network administrator

A network administrator will typically operate and administer his site using commercial software that was bought from other companies. He will buy operating system software, server software (e.g. e-mail server, web server, database server) and application software (e.g. enterprise resource planning systems) to satisfy the IT needs of his company. If the software contains vulnerabilities that allow an attacker to do unpleasant things (and we will see further on that most software unfortunately does contain such vulnerabilities), the administrator cannot change the software itself.

Instead, the countermeasures that he will take to reduce the risks of an attack could include:

- deploy additional layers of protection around the software, for example install a *firewall*: a machine that tightly controls network access to the company intranet.
- monitor the usage of the network and the software to detect security breaches, by using intrusion detection or fraud detection systems.
- keep his software up-to-date: if the vendor of the software releases an upgrade that patches a known vulnerability, the administrator should install this patch as soon as possible.
- proactively scan his own site for possible vulnerabilities using so-called vulnerability scanners.

1.2.2 Security for the software developer

On the other hand, the software developer responsible for making the software will take a completely different stance. This developer should try to make sure that the software he produces itself enforces as much as possible the correct rules of usage. He can, for instance, try to reduce risks by:

- correct usage of software security technology, including cryptography and access control.
- good software engineering practices to increase robustness of the software.
- source code review by somebody who is familiar with typical software vulnerabilities

1.3 The goal of this book

In this book, we take the viewpoint of the software developer: we will not discuss firewalls, intrusion detection systems or vulnerability scanners. Instead, we aim to give the reader a good overview of software security technology, and a thorough familiarity with typical software vulnerabilities.

After reading this book, you should be better equipped to write secure software, i.e. you should be able to make your software enforce correct rules of use.

The book is structured as follows. We start of by looking at an example distributed application (a simplified e-mail system), and give extensive examples of possible threats to, and vulnerabilities in this system. This is meant to give the reader a good idea of what insecure software is, and it also motivates the need for the techniques we will discuss further on in the book.

The next two chapters consider cryptographic technology. We will not discuss the intricate details of cryptographic algorithms. Instead, we will discuss the behaviour of the most important algorithms from a black-box point of view, and we will show how these basic building blocks can be composed to form cryptographic protocols that can achieve useful security goals. A fair amount of attention will be given to software interfaces to cryptographic technology: the typical software engineer will never implement cryptographic algorithms or protocols, but instead use an existing implementation from a cryptographic library. Hence, being familiar with the typical structure of such libraries is important.

Then, we turn to access control. We first look at classical access control models, and we see how implementation errors can lead to vulnerabilities in access control software. We also consider more advanced access control techniques that work well in a distributed setting, or in the presence of untrusted or mobile code. Again, we look in detail at existing security architectures that implement some of these access control models.

We finish by tracing security aspects throughout the software lifecycle, and discuss (at a high level) what role security plays in the analysis, design, implementation, deployment and maintenance phases of a software system.

Chapter 2

(In)secure software: a case study

Broadly speaking, software is insecure if it can be (ab)used to do things that should not be possible. To make this more concrete, this chapter discusses an extensive case study: Internet e-mail. We first give a simplified and high-level account of how Internet e-mail works. Then we go on to discuss potential threats to such an e-mail system, and we discuss how well current e-mail systems protect against these threats.

After reading this chapter, you should have a very good idea of what insecure software is, and you should be convinced of the usefulness of the security techniques that we will discuss further in this book.

2.1 A simplified description of Internet e-mail

We discuss the inner workings of a simplified e-mail system. We assume the Internet is subdivided in a number of *domains* with names like `domain1.com` and that each domain has a number of users. E-mail addresses have the form `username@domain.com`. The goal of an e-mail system is to allow any user in any domain to send electronic messages to any other user in any other domain.

Our simplified E-mail system uses three kinds of programs.

An *E-mail client* is a program run by an end-user to make use of the e-mail system. Think of Microsoft Outlook or Netscape Messenger as examples of e-mail clients. The e-mail client has two important purposes:

- It allows its user to compose a new message. The user supplies an e-mail address of a recipient and the content of the e-mail, and the

e-mail client will take care of dispatching the message. (What exactly happens when a message is sent is discussed later in more detail.)

- It allows a user to retrieve any new messages that have arrived for him to read them and perhaps file or delete them.

A *Mail Storage Server (MSS)* is a service that acts as a collection of mailboxes for e-mail messages. There is one MSS per domain. The MSS is continuously online and accepts e-mail messages for any user in its domain. After it has accepted a message, it will store it in the mailbox of the recipient. Later, an e-mail client can connect to the MSS to retrieve messages for a specific user. An MSS is essentially a simplified version of an IMAP server. When an e-mail client connects to the MSS, it has to supply a user name and password. If the password is correct, the MSS gives the e-mail client access to all the messages of the corresponding user.

A *Mail Transfer Server (MTS)* is a service responsible for transfering mail to the destination domain. There is again one MTS per domain. The service continuously waits for somebody to connect to it, and after somebody connects, the service accepts a complete e-mail message. The service then looks at the recipient address. If the address is in another domain, the MTS will forward the message to the MTS of the recipient domain. If the address is in the domain of the MTS itself, the MTS forwards the message to the local MSS. An MTS server is a simplified version of an SMTP server.

Figure 2.1 summarizes the operation of our simplified e-mail system.

When `u1@domain1.com` sends a message to `u2@domain2.com` the following things happen:

1. User u1 composes the message using his e-mail client, and supplies the address `u2@domain2.com`.

2. When the user presses the "Send" button, the e-mail client forwards the complete message (including address) to the MTS.

3. The MTS notices it is a non-local message (destined for another domain) and hence forwards it to the MTS of `domain2.com`.

4. The MTS of `domain2.com` accepts the message, and since it is a local message delivers it to the MSS, where it is stored in the mailbox of user u2.

5. Some time later, user u2 starts up his e-mail client and asks his client to check for new mail (also typing in his password).

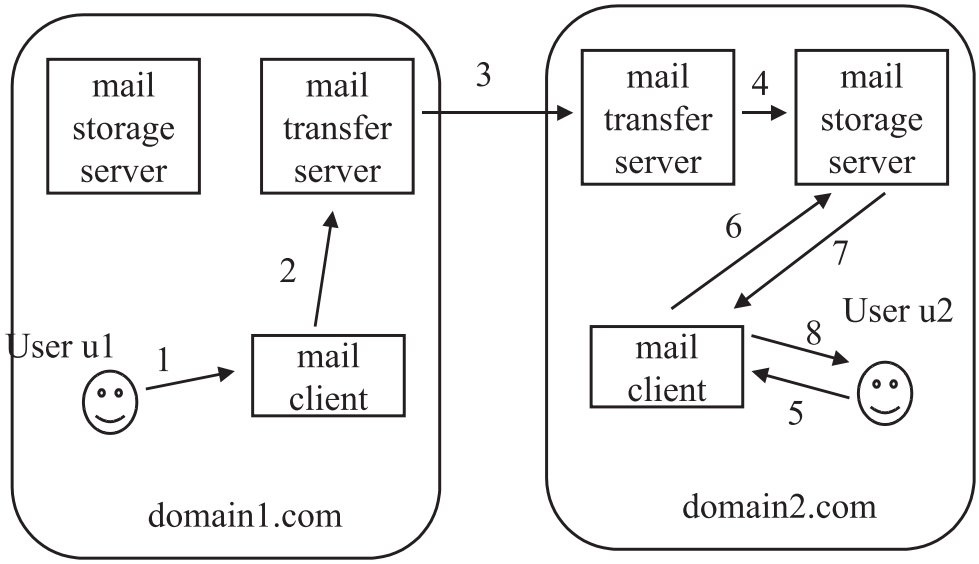

Figure 2.1: Simplified e-mail system

6. The e-mail client contacts the MSS of his domain supplying the username and password.

7. After verifying the password, the MSS sends any new messages in u2's mailbox (including the one just arrived from user `u1@domain1.com`) to the e-mail client.

8. User u2 can now read the message.

While being a simplification, the system described above is very similar of the actual e-mail system used on the Internet. We will discuss potential threats to this system, and all threats we discuss are real threats in the sense that they also apply to the real Internet e-mail system.

2.2 Threats to an e-mail system

2.2.1 Eavesdropping on e-mail

In steps 2,3,4 and 7 in figure 2.1, the e-mail message is sent over a TCP/IP connection. TCP/IP connections can be eavesdropped on relatively easily. Hence, there is clearly a threat to the confidentiality of an e-mail message. Moreover, the message passes through a number of servers (two MTS's and one MSS). The administrators of those servers also have the ability to see

any message passing through the server. Hence, it is by no means certain that a message to user u2 will only be read by user u2.

2.2.2 Modifying e-mail or e-mail spoofing

In a similar way, if an attacker can intercept a TCP/IP connection (e.g. by posing as the MTS of domain2 to the MTS of domain1 and as the MTS of domain1 to the MTS of domain2), it is relatively straightforward to modify the e-mail message in transit. Also, administrators of the servers through which the message passes have the ability to modify it at will. Hence there is a real threat to the integrity of an e-mail message. User u2 can not be assured that the message he receives is actually the message sent by u1.

On a related note, it is fairly easy to send an e-mail in somebody else's name. When an e-mail client delivers a message to an MTS, he includes the name of the sender. But nothing prevents an attacker from telling his e-mail client that he is user u1 while he is not. For sending e-mail, typically no password is required. Sending an e-mail in the name of another user is called e-mail *spoofing* and is very easy to do, also on the Internet e-mail system.

2.2.3 Attacks against e-mail servers

The two threats described above are communication-related threats: communication is eavesdropped on or intercepted and modified. These threats are typically the first that come to mind when thinking about the security of a networked or distributed system.

But in practice, attacks against servers are much more common. We discuss some possible attacks against MTS's or MSS's.

Picking up somebody else's mail

The MSS uses passwords to verify the identity of a user: it will only deliver the mail of a user if the correct password for that user is supplied. But verifying identities through passwords is not very secure. If passwords are sent over the Internet (as might happen when a user retrieves his mail while being abroad) they can be eavesdropped on. If a password is badly chosen, it might be guessed. Hence the threat of an attacker getting hold of a password and using it to retrieve somebody else's mail is real.

Exploiting vulnerabilities in the server software

A server is essentially a software layer that offers some limited form of functionality (e.g. transferring mail in the case of an MTS) to the Internet. If the

MTS server software is completely correct and bug-free, it will never allow somebody to do anything else but transfer mail. But history has shown us that software is seldom completely correct, and any bug in the server software may allow an attacker to make the server do something else than what it was intended to do.

A very important example of such a bug is a so-called *buffer overflow*. We will discuss this kind of bug in detail later in this book. If the server forgets to check the size of some of its input, an attacker can make the server do anything. The `sendmail` program, one of the most popular MTS's on the Internet has a long history of such buffer overflow problems.

In fact, it does not even have to be a bug that allows an attacker to abuse the server software. Older versions of `sendmail` had a so-called *debug mode*, introduced to help administrators solve configuration problems. This debug mode essentially allows anybody to do anything on the server machine. If one forgets to disable this debug mode when putting the server in production (as has happened many times with `sendmail` on the Internet), it is very easy to compromise the server machine.

2.2.4 Spam e-mail

The e-mail system does not limit in any way the amount of messages that anybody can send. As a consequence, it is possible to do mass-marketing by e-mail. A marketer can collect millions and millions of e-mail addresses and then bombard these recipients with unsollicited mail, e.g. with commercial purposes. Moreover, since an e-mail can be sent without revealing your identity, this kind of mass-mailing can be done without the risk of being prosecuted.

This is an example of a "feature" that some people will consider to be a threat, while other people will consider it useful.

2.2.5 Denial-of-service

The e-mail system can be brought to its knees by a determined attacker in a number of ways. Attacks that try to bring down or at least strongly degrade a service are called *denial-of-service* attacks. For the e-mail system, denial-of-service could be achieved by:

- Flooding the e-mail box of a recipient, thus exhausting his storage space on the MSS.
- Using a buffer overflow vulnerability in an MSS or MTS to shut down the server.

- Sending large amounts of e-mails to an MTS making it unavailable to other clients.
- ...

2.2.6 Attacks against e-mail clients

Implementation vulnerabilities in clients

An e-mail client is again a piece of software designed to provide a specific functionality (composing, receiving and displaying e-mail). If it is implemented completely correct, it cannot be abused by an attacker. But if it contains buffer overflow bugs, an attacker can send a specially crafted e-mail that will make the client do anything the attacker desires.

Executable attachments

Some e-mail clients will allow users to attach any file to an e-mail and the recipient can then open the file by clicking a link included in the e-mail.

This is a serious threat, since opening a file can mean that you actually execute a piece of code sent to you by somebody else. That piece of code can then do anything it wants on your computer. This technique of attaching executables to e-mails has been used by a number of high-profile viruses and worms.

2.2.7 Other threats

While we have already discussed a large number of potential threats against an e-mail system, our list is by no means exhaustive (in fact making an exhaustive list would be quite hard if not impossible). Some other example threats could be:

- Repudiation of sending: the sender of an e-mail message can easily deny having sent the message. The recipient cannot prove that the message actually came from the sender.
- Repudiation of receipt: the receiver can easily deny ever having seen a message. The sender can not prove that the message actually got to the recipient.
- Privacy threats: even when e-mail content could be made confidential users might worry that attackers can find out who they are communicating with.

- ...

Moreover, the e-mail system can also be attacked by attacking one of its supporting services (e.g. the operating systems on which clients and servers run, or the DNS system used to resolve domain names).

2.3 Discussion

Hopefully, the list of threats in the previous section has convinced you that a straightforward implementation of an e-mail system is certainly insecure: it will by no means counter all identified threats, or in other words, it will not enforce correct rules of usage.

This observation is of course not only true for an e-mail system, but for any more or less complex software system.

As we discussed in chapter 1, it is up to the designers and implementors of the e-mail system to analyze each of the threats and to assess the risk associated with the threat. If the risk is deemed high enough, countermeasures have to be designed into the system. This risk assessment could be strongly influenced by the environment in which the system will operate. For example, if we would deploy the e-mail system described above on a company intranet only, and not on the Internet, some threats would become unimportant.

The rest of this book will mainly focus on countermeasures that you can implement in software. We will discuss technology you can use as countermeasure to the threats discussed above. As a preview, let us consider some possible countermeasures for some of the threats we discussed.

The "communication-oriented" threats, like the eavesdropping and modification in transit will be countered by using cryptographic techniques, discussed in chapters 3 and 4. Using cryptography, one can protect confidentiality and integrity of messages sent over an unsecure communication channel.

The threats related to identity verification (the threat that somebody might pose as somebody else when retrieving mail or sending mail) will be covered by authentication techniques, that build on cryptographic techniques. Authentication is discussed in chapter 4.

Attacks against servers and clients will be handled in chapters 5, 7 and 6 where we cover access control also in the case of mobile code like executable attachments, and where we discuss typical implementation vulnerabilities and how to avoid them.

The real Internet e-mail system has been extended over the years to implement some of these countermeasures. S/MIME for instance, is a standard for cryptographically protecting e-mail messages, thus countering some of

the eavesdropping and modification threats. However, since these countermeasures are add-ons, it takes a long time before they are actually widely used. Although most e-mail clients now support S/MIME, only a very small percentage of e-mails sent today is actually protected with it.

Moreover, keep in mind that any countermeasure you impose can in turn again have vulnerabilities (recall figure 1.1). For example, if you choose to use cryptography to protect e-mail confidentiality, a bad implementation of the encryption, or bad management of the encryption keys will make you vulnerable again. As we will see later in the book, S/MIME relies on a Public Key Infrastructure (PKI) for managing its keys, and as of today, no good Internet-wide PKI's exist.

2.4 Further Reading

TODO

References on E-mail, S/MIME, weak passwords, ...

Part II

Software Security Technologies

Chapter 3

Cryptographic primitives

Cryptography is the scientific study of mathematical and algorithmic techniques relating to *information security*. Cryptographic techniques will help to protect information in cases where an attacker can have physical access to the bits representing the information, e.g. when the information has to be sent over a communication channel that can be eavesdropped on by an attacker.

Cryptographic primitives are the basic building blocks for constructing cryptographic solutions to information protection problems. A cryptographic primitive consists of one or more algorithms that achieve a number of *protection goals*. Possible protection goals are:

- *confidentiality*: information should be protected against reading by unauthorized parties
- *integrity*: information should be protected against modification by unauthorized parties
- *message authentication*: giving strong evidence about the identity of the originator of a message
- *non-repudiation*: giving non-deniable evidence of the identity of the originator of a message
- ...

3.1 Primitive building blocks

There is no well-agreed upon *complete* list of cryptographic primitives, nor are all cryptographic primitives independent: it is often possible to realize

one primitive using a combination of other primitives. However, the following primitives are generally considered useful, and are implemented in most cryptographic libraries:

- cryptographic hash functions
- secure random number generation
- symmetric encryption
- asymmetric encryption
- message authentication codes
- digital signatures

We will discuss each of these primitive building blocks in detail. It is not our goal to describe how each of these primitives is implemented in practice. Instead, we will focus on describing the black box behavior: we describe how the primitives should be used and what protection goals they realize.

3.1.1 Cryptographic hash functions

One of the simplest primitives is the cryptographic hash. In general, a *hash* function is a function that maps an arbitrary long input stream of bits onto a fixed size output. The output is called the hash of the input. Non-cryptographic hash functions are used for example to build so-called hash-tables.

A *cryptographic* hash function is a hash function that realizes a number of protection goals:

- a cryptographic hash is a *one-way function*: it is easy to compute the output given the input, but it is extremely hard (i.e. it takes a prohibitively long computation) to find an input that maps to a given output.
- a cryptographic hash is *collision-resistant*: it is computationally infeasible to find two different inputs that map to the same output.

A cryptographic hash of a message is sometimes also referred to as a *fingerprint* of the message, or a *message digest.*

One possible use of the hash primitive is to detect changes to a file. A hash of the file is computed and stored in a safe place. If one suspects that the file is modified (e.g. by a virus infection) the hash can be recomputed

and compared to the stored value. If the values match, the file was certainly not modified: it would be infeasible to modify the file in such a way that the hash remained the same (this follows from the collision resistance property). More commonly, the cryptographic hash primitive is used in cryptographic protocols and not standalone.

Examples of real-life cryptographic hash functions include:

- The Secure Hash Algorithm (SHA-1): proposed by the U.S. National Institute of Standards and Technology. It has a 160 bit output.
- MD-5: designed by Ron Rivest, with a 128 bit output.

3.1.2 Secure random number generation

Many cryptographic primitives and protocols will rely on random numbers during their use. But random numbers are extremely hard to obtain on a deterministic machine like a computer. Various techniques exist to generate random bits from timing user interface events or measuring other values such as system load in the computer system. However, these techniques are typically slow, and it is hard to measure precisely how random they are.

A secure random number generator is a cryptographic primitive that generates a large number of output random bits from a small number of input random bits. The input is called the *seed*, and should typically be truly random (i.e. obtained using one of the techniques above). The random number generator then expands this small number of random bits into an arbitrary long stream of random bits.

A secure random number generator realizes the following security goals:

- its output should pass statistic tests of randomness
- if the seed is unknown, it is computationally unfeasible to predict the next bit of the random number generator, even if an arbitrary long part of the output is known

Non-secure random number generators typically only realize the first of these two goals. Hence it is extremely important to make sure you use a *secure* random number generator when you rely on the second property.

Secure random number generators are used to generate cryptographic keys or as a basis for building stream ciphers (see later).

Examples of real-life secure random number generators include:

- ANSI X9.17, approved by U.S. Federal Information Processing Standard to generate cryptographic keys.

Figure 3.1: Symmetric encryption system

- Blum-Blum-Shub, provably secure under the assumption that factoring large integers is intractable.

3.1.3 Symmetric encryption

A symmetric encryption system consists of two algorithms: an *encryption* algorithm and a *decryption algorithm*. Both of these algorithms take two input parameters. The encryption algorithm takes as input a fixed size *key*, and an arbitrarily sized bit stream called the *plaintext*. It outputs a bit stream, called the *cipher text,* of a size comparable to the plaintext size. The decryption algorithm takes as input a fixed size key, and an arbitrarily sized bit stream, called the cipher text, and outputs a bit stream, called the plaintext, of size comparable to the input cipher text.

The idea is that these two algorithms will be used together as shown in Figure 3.1. If the keys are the same, the decryption function will be the inverse function of the encryption function. On the other hand, if you encrypt with one key, and decrypt the resulting cipher text with another key, the plaintext output by the decryption algorithm will typically look like random noise.

A symmetric encryption system essentially tries to realize confidentiality: the goal is to make the cipher text unreadable for anyone who does not know the key. Hence, a symmetric encryption system reduces a message confidentiality problem into a key management problem: you should make sure only authorized parties have access to the key.

We can be more precise about the protection goals realized by a symmetric encryption system. A good modern system achieves all of the following goals:

- secure against *cipher text-only* attack: given an arbitrary number of cipher texts, it is infeasible to compute the corresponding plaintexts.

- secure against *known-plaintext* attack: given an arbitrary number of plaintext-cipher text pairs, it is infeasible to compute the key, or to compute the decryption of other cipher texts.

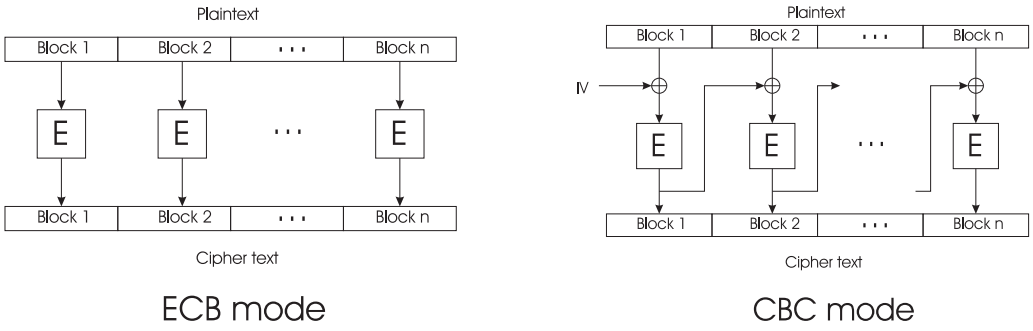

Figure 3.2: Modes of operation

- secure against *chosen-plaintext* or *chosen-cipher text* attack: even if the attacker is allowed to choose the pairs in a known-plaintext attack, it is still infeasible to compute the key.

It is important to emphasize that the realization of these security goals relies only on the secrecy of the key: the encryption algorithm and decryption algorithm are assumed to be known by the attacker.

Symmetric encryption systems can be realized using *block ciphers* or *stream ciphers.*

Block ciphers

A block cipher is actually a symmetric encryption system operating on fixed size blocks of plaintext. Hence, for a given value of the key a block cipher encryption implements a mapping between plaintext blocks and cipher text blocks and the decryption implements the inverse mapping. Block ciphers can be extended to full symmetric encryption systems, by chopping an arbitrarily sized plaintext up in equally sized block, and encrypting these blocks one-by-one. There are various ways in which the block cipher can be used to encrypt the blocks, and these are called different *modes of operation* of the block cipher. Figure 3.2 summarizes some possible modes of operation.

The *Electronic Code Book (ECB)* mode treats every input block separately. The disadvantage of this mode is that repeating plaintext will lead to repeating ciphertext. The *Cipher Block Chaining (CBC)* mode combines a plaintext block with the previous cipher text block (using exclusive or of the bits) before encrypting. After decryption, the exclusive or is undone. The first plaintext block cannot be combined with a previous cipher text block and is combined with a so-called *initialization vector (IV)* instead. The different modes of operation have different advantages and disadvantages, and it is important to pick the right mode for a given application.

Finally, when using a block cipher, one should consider what to do when the plain text is not an exact multiple of the block size. In that case, the last block will not be completely filled with plain text, and some kind of *padding* scheme is necessary. For symmetric algorithms, padding is usually implemented according to the standard PKCS#7. This padding algorithm works as follows: if n bytes need to be added to create a full block, the algorithm will give each of these bytes the value n. (E.g. to pad 2 bytes, the byte string 02 02 will be used).

Examples of real-life block ciphers include:

- The Data Encryption Standard (DES), a block cipher with 64-bit block size and 56-bit key, designed by IBM and the NSA in the seventies. DES has been a very widely used algorithm in the eighties and nineties, but has now reached end-of-life: the key size and block size are too small and exhaustive key search machines for DES have been built.

- Triple DES, performs three DES encryptions with independent keys (actually encryption-decryption-encryption for backward compatibility reasons).

- The Advanced Encryption Standard (AES), the successor of DES, selected by the U.S. National Institute of Standards and Technology. The AES was selected through an international contest, eventually won by the Rijndael algorithm, designed by Vincent Rijmen and Joan Daemen. Rijndael (and hence the AES) allows for variable key and block sizes.

Stream ciphers

A stream cipher is a symmetric encryption system that can operate bit-by-bit, and hence is very useful in interactive applications where the encryption system is not allowed to introduce delay in communication. A stream cipher is a stateful algorithm that accepts plaintext bits and emits cipher text bits, but of course every bit is encrypted in a different way. A typical example of a stream cipher is the *one-time-pad*. The one-time-pad uses a key as long as the plaintext, and encrypts every plaintext bit by performing exclusive or with the corresponding key bit. This encryption scheme is provably secure, but not very practical because of the required key size.

More typically, a stream cipher is implemented by generating a key stream from a seed using a *key stream generator* (i.e. essentially a secure random number generator), and combining plaintext bits with this generated key stream. The seed for the key stream generator is then the actual key of the symmetric encryption system.

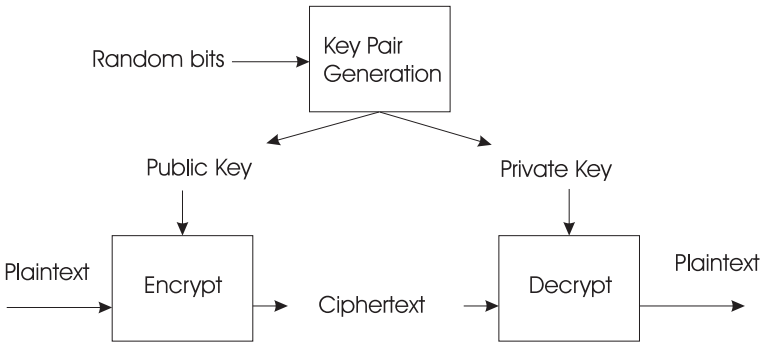

Figure 3.3: Asymmetric encryption system

Examples of real-life stream ciphers include:

- SEAL, Software-optimized Encryption ALgorithm, considered insecure now.
- RC4, a proprietary stream cipher of RSA labs.
- It is possible to use block ciphers as stream ciphers, when using the *Output Feedback Mode* of operation.

3.1.4 Asymmetric encryption

An asymmetric encryption system consists of three algorithms. The *key pair generation* algorithm generates two keys, a *public key* and a *private key*, from some random input bits. These two keys together are called a key pair. For different random input bits, the key stream generator will generate different key pairs. The encryption algorithm takes a public key and a plaintext as input, and generates the corresponding cipher text as output. The decryption algorithm takes a private key and a cipher text as input and outputs the corresponding plaintext. If the public key and private key belong to the same key pair, the decryption algorithm is the inverse of the encryption algorithm. However, if the private key belongs to a different pair, the decryption algorithm will output random noise.

An asymmetric encryption system also tries to realize confidentiality of messages. It is intended to be used as follows: an entity wishing to receive confidential messages generates a key pair, and distributes the public key widely. Anybody wishing to send a confidential message can use this public key for encryption. Only the owner of the private key will be able to read the encrypted message.

A good asymmetric encryption system is also secure against the cipher text-only, known-plaintext and chosen-plaintext or -cipher text attacks. In particular, resistance against chosen-plaintext attacks is extremely important, since an attacker can perform encryptions.

The main advantage of an asymmetric system over a symmetric one is that key management is easier: for a symmetric system, a key must be agreed upon and distributed in advance between the parties wishing to communicate confidentially. This key distribution must happen over a secure channel: the key must remain secret. For an asymmetric system, only the public key must be distributed, and distribution must happen over a channel that is only integrity protected: the public key must not remain secret, but should only be protected from modification in transit. (We will see why the channel must be integrity protected in chapter 4.)

The block size of an asymmetric algorithm is typically much larger than that of a symmetric encryption algorithm. Since asymmetric encryption is typically used just to encrypt a symmetric key (see further why), only single block encryption is needed. Padding algorithms for asymmetric encryption systems are more elaborate than in the symmetric case, among others because the padding algorithm also takes care of *randomization* of the plaintext to protect against an attacker guessing the plaintext and verifying his guess by encrypting it with the (widely known) public key and comparing with the cipher text. By using a padding algorithm that inserts random bytes, the same plaintext will encrypt to different values, thus countering the guessing attack.

The main disadvantage of asymmetric systems is that they are typically many times slower than symmetric systems. Hence, in practice they are often only used to send a symmetric key to the other side. This symmetric key is then used for bulk encryption. This will be discussed in more detail in the chapter on cryptographic protocols.

Examples of real-life asymmetric encryption systems include:

- RSA, by far the most widely used system.
- Rabin encryption system, provably secure, but not widely used.
- El Gamal system, based on the difficulty of taking discrete logs
- Elliptic Curve Cryptography (ECC), based on El Gamal, uses smaller keys for the same security level, and hence is suitable for embedded or mobile systems.

Example padding algorithms include PKCS#1 v1.5 and OAEP.

3.1.5 Message Authentication Codes (MAC's)

A Message Authentication Code (MAC) is an algorithm that takes two inputs: an arbitrary long message and a fixed size key. It outputs a fixed size hash value of the message. MAC's are very similar to cryptographic hashes. They are also one-way and collision-resistant. But they have one extra security property: only someone knowing the key can compute the hash.

MAC's are typically used for integrity protection. To send a message to another party, one first agrees on a secret key, and then one sends the message, together with a MAC of the message computed using the agreed key. The receiver recomputes the MAC and if the transmitted and computed MAC are equal, he can be assured that the message was not changed in transit: an attacker changing the message will invalidate the MAC, and since the attacker does not know the key, he cannot compute a new valid MAC.

Examples of real-life MAC's are:

- encrypting a cryptographic hash with a symmetric encryption system
- H-MAC is a construction that turns any cryptographic hash function into a MAC
- DES-CBC-MAC: encrypt the message with DES in CBC mode and take the last block as MAC

3.1.6 Digital Signatures

A digital signature system consists of three algorithms. The *key pair generation* algorithm generates two keys, a *public key* and a *private key*, from some random input bits. These two keys together are called a key pair. The *signing* algorithm takes a private key and a message as input, and generates a so-called *signature* as output. The *verification* algorithm takes a public key, a message and a signature as input and outputs a boolean value. If the public key and private key belong to the same key pair, the verification algorithm will return true if the message used to generate the signature is identical to the message used in the verification process. In all other cases (non-matching public and private key, or different message), the verification algorithm returns false (see Figure 3.4).

A digital signature system protects message integrity: it is computationally infeasible to compute a message-signature pair unless you know the private key. In that sense, a digital signature is similar to a MAC. But a digital signature can in addition realize non-repudiation: since the private

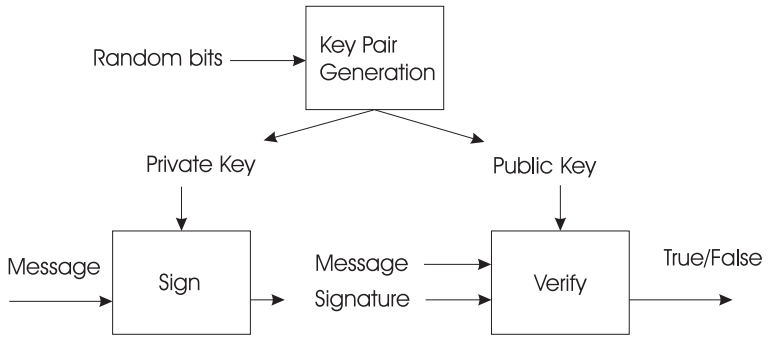

Figure 3.4: Digital Signature system

key is known to only one party, only that party could have produced the signature. Verification is possible without knowing the private key.

But similarly to asymmetric encryption systems, digital signature systems are typically quite slow, and certainly much slower than MAC's or cryptographic hashes. In practice, digital signatures are always computed in two steps: first a hash of the message is computed, and secondly the hash is signed using a digital signature system. Hence, the digital signature primitive itself is usually applied to one fixed size message block that contains a hash of the to-be-signed message, and that is suitably padded up to the block size of the primitive.

Examples of real-life digital signature systems include:

- RSA: the RSA public key encryption can also be used as a digital signature system. This fact often leads to the misconception that signing is just "encrypting with the private key" (because that is how you sign with RSA). This is an incorrect generalization: most other digital signature systems are by no means "encryptions".

- DSA, a U.S. National Institute of Standards and Technology approved system, designed by NSA.

- ElGamal signatures that can again be generalized to Elliptic Curve signatures, suitable for implementation in systems with limited resources.

3.1.7 Notational conventions

TODO

3.1.8 Conclusion

A fairly large number of cryptographic building blocks have been described. It is important to understand well for every primitive what it offers. There is a long history of abuse of these primitives, in the sense that they have often been applied in situations for which they were not designed.

Also, the design of algorithms that implement the primitives is known to be very hard: many cases are known of custom-designed algorithms to implement cryptographic primitives that turned out to be quite easy to break. Hence, as a general rule, you should use well-known and published algorithms, and refrain from inventing your own.

Finally, even implementing published algorithms is known to be error-prone: when a crypto system fails, it is usually because it was not well implemented. As a consequence, developers should avoid implementing these primitives themselves, but should instead rely on libraries implemented by a reputable vendor.

3.2 Key Management

Cryptographic algorithms are never complete solutions for security problems: typically, a cryptographic primitive can be used to *transform* a security problem (e.g. a confidentiality problem) into another security problem, namely a *key management* problem. Encryption and signatures rely completely on secrecy of certain keys for their security. Hence, the security of the overall system is determined not only by which cryptographic algorithms you use, but also (and typically more so) by how you manage your keys.

In this section, we shortly discuss some key management issues.

3.2.1 Generating keys

Cryptographic keys should be hard to predict or guess, and it should be unfeasible to do an exhaustive key search attack. Both these points should be kept in mind when deciding how keys will be generated.

If keys are generated by a computer, you should take care to use a secure random number generator that has a seed that is at least as long as the keys you will generate. If not, an exhaustive key search attack will be reduced to an exhaustive seed search attack.

If keys are generated by humans (as in password derived keys), you should be aware that user-generated secrets are typically fairly easy to guess (e.g. using a dictionary attack). The longer the user generated secret, the harder it gets to guess or predict it: hence a pass phrase is better than a password and

a password is better than a PIN. However, the security-convenience tradeoff should be kept in mind here: longer pass phrases are typically less convenient for the user, and if this makes the system too inconvenient for him, he will probably find ways to avoid this inconvenience (like never logging out or locking his screen) that compromise security in another way. Of course, a password or a pass phrase cannot be used directly as a cryptographic key. Standardized algorithms exist to derive an encryption key from a password.

Also the length of the keys you generate is determined by a tradeoff: the longer the key, the harder it will be to break the cryptographic primitive. But longer keys will also make the primitive more computationally expensive. Hence the key length you choose will depend on the value of the information you are trying to protect, and on the amount of time you want the protection to stand.

Choosing a key length is different for symmetric systems than for asymmetric systems.

For symmetric systems, the strongest attack against the system should typically be an exhaustive key search attack, and hence key length is typically determined by this. Your key should be long enough to make exhaustive key search infeasible. 64 bit keys were considered secure until a few years ago. Nowadays, a 128 bit key should be good enough for a few years to come.

For asymmetric systems, the strongest attack is typically dependent on the system in use. For RSA for instance, the system will never be attacked through exhaustive key search, but through integer factoring algorithms. Hence, for asymmetric systems, the key length you choose depends on the system you use. For RSA, a 1024 bit key will give you medium and short-term security. A 2048 bit key will be secure for at least a few years against a powerful adversary. Elliptic Curve asymmetric systems on the other hand, typically require much smaller keys for a comparable level of security.

Finally, a key should never be used arbitrarily long: key lifetime should be limited in time and in amount of data that is encrypted with it.

3.2.2 Storing keys

Since the security of your system depends on the secrecy of your keys, keys should be protected well while stored. This protection can be done in a number of ways:

- you can store keys in human memory (i.e. use keys derived from passwords or pass phrases)

- you can use operating system access control to limit access to keys

- you can store keys in an encrypted form: in that case the key used to do the encryption (the *key encryption key*) must be safely stored again somewhere else
- you can store keys in some tamper-resistant device like a smart card, or a dedicated hardware cryptographic coprocessor.

In practice, often a combination of these is used: e.g. an encrypted key store that is also protected through operating system access control, and for which the key encryption key is derived from a password.

3.2.3 Key agreement or distribution

By far the hardest part of key management is making sure the correct keys get to the authorized parties in a secure way. There are two basic techniques to tackle this problem. With *key agreement* two parties wishing to share a key execute a protocol that leaves them with a shared secret. With *key distribution* one party (often a trusted third party) generates the key, and distributes it to the authorized parties.

Key agreement

A key agreement protocol is a protocol between two parties that has the following property: at the end of the protocol, the two participating parties have a shared secret, and an eavesdropper that saw all the communication taking place, can not compute the agreed secret. As a very simple example of a key agreement scheme, the two parties could each generate a random bit string and send it over encrypted with the public key of the other party. The final secret is the exclusive or of the two bit strings.

A key agreement protocol is typically vulnerable to an active adversary, that can modify messages instead of just eavesdropping on them. An *authenticated* key agreement protocol detects this tampering but will typically require help of a third party.

We will discuss key agreement in more detail in the chapter on cryptographic protocols.

Key distribution

Key distribution tackles the problem of safely distributing key material generated by one party to other authorized parties. Typically, a (trusted) third party is used to assist in the distribution.

Key distribution typically requires rather complex protocols and will be discussed in detail in the chapter on cryptographic protocols.

Standards for key encoding

TODO

3.3 Software interfaces to cryptographic primitives

A good cryptographic library should have the following properties:

algorithm independence It should make it easy for developers to use the primitives in a way that is as independent as possible from the concrete algorithm used. Developers should be able to just use "the default hash algorithm". New algorithms should be easy to add to the library, and it should preferably be configurable at runtime what the defaults are for each of the primitives.

Algorithm independence is important because cryptographic algorithms can be broken over time. Hence, applications should be able to switch to a different algorithm with relatively little effort.

implementation independence In a similar way, a library should allow for easily adding implementations of algorithms. It should be possible that multiple implementations of the same algorithm are present, and it should again be configurable what the default implementation is. For example, for a server machine, a (fast) hardware implementation might be needed whereas for clients a (slower but more portable) software implementation is preferred.

Modern cryptographic libraries or API's for often use the concept of *Cryptographic Service Providers (CSP)* to offer cryptographic primitives in an algorithm and implementation independent way. The library is structured as a framework in which CSP's can be plugged in.

We first look at the concept of CSP's, and then look in detail at a few real-life examples of cryptographic libraries.

3.3.1 Cryptographic Service Providers

Engine classes and algorithm classes

Modern cryptographic API's offer cryptographic building blocks in an abstract way, more or less as we have described them in this chapter. Every cryptographic primitive is represented by a so-called *engine class* that offers an API to use the primitive. For example, a MessageDigest class could

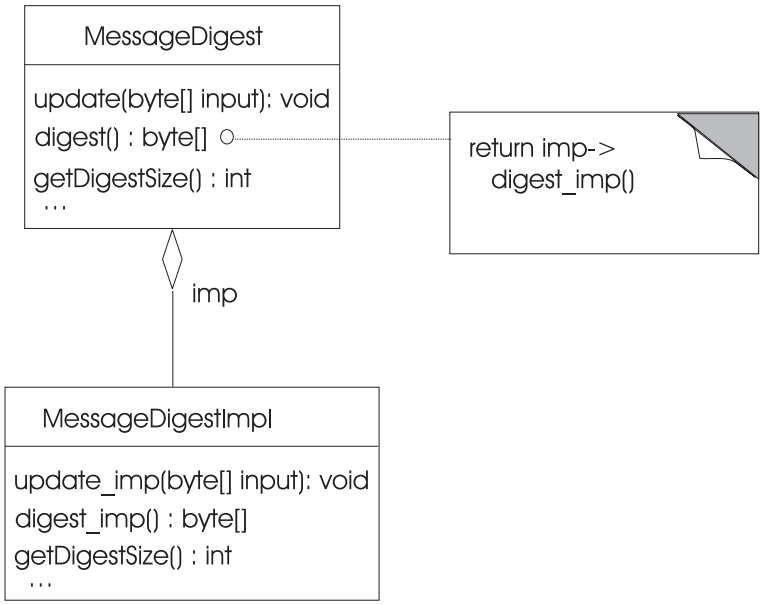

Figure 3.5: Bridge pattern for engine classes

represent the primitive of cryptographic hash functions or message digests. It could have a method for providing the MessageDigest with input bytes (e.g. a void update(byte[]) method that can be called one or more times), a method to compute the digest (e.g. byte[] digest()), and a method that returns the (fixed) size of the digest (e.g. int getDigestSize()).

To decouple the engine class from its implementation, a number of different techniques are possible.

A first technique is to use the Bridge design pattern to make the implementation changeable as is shown in the class diagram in Figure 3.5.

A factory method will allow an application to create a MessageDigest object, and that factory method will contain the logic to decide which MessageDigestImpl object will be used to implement the message digests. For example, the factory method could consult a configuration file, or could accept parameters to determine the implementing class.

Once created (and hence configured with an implementation object), the MessageDigest object will delegate all the real work to the implementation object.

This first technique for decoupling is used in the Java platform, and will be discussed in more detail in section 3.3.2.

A second way to do the decoupling is to use inheritance (see figure 3.6). Engine classes are abstract, and define abstract methods for working with the corresponding cryptographic primitive. The abstract MessageDigest class

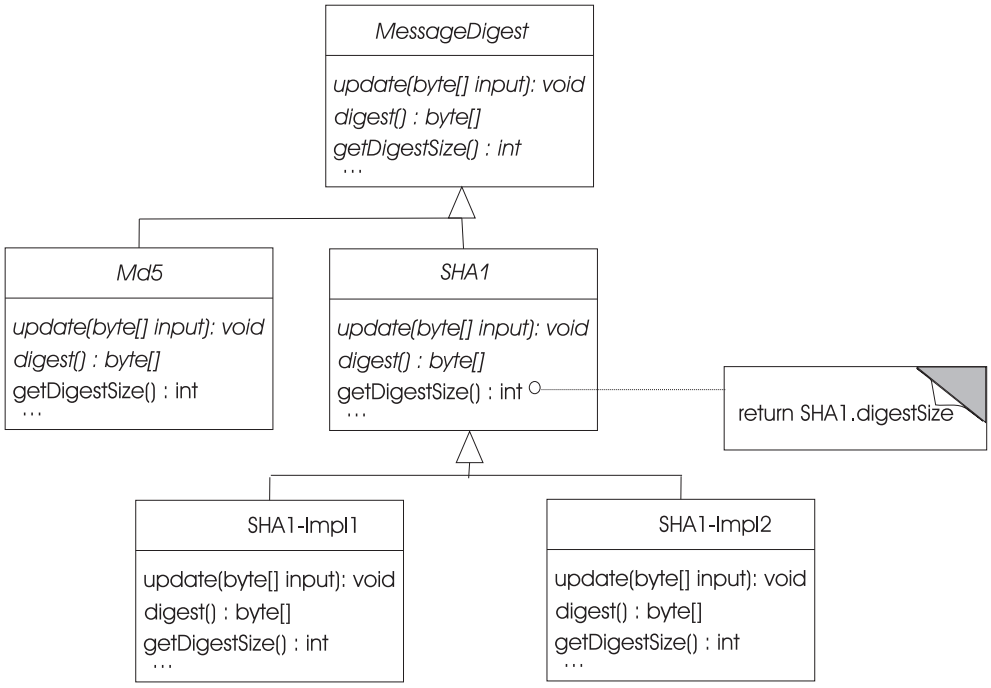

Figure 3.6: Decoupling by inheritance

would have abstract methods for feeding data to be hashed, for computing the hash, and for returning the size of the hash. Every abstract engine class has a number of abstract subclasses that represent algorithms that implement the primitive. The MessageDigest class could have abstract subclasses SHA1 and MD5 that represent two different message digest algorithms. We call such subclasses *algorithm classes*. The algorithm classes override those methods of the engine class that will be identical for all implementations of the algorithm. For instance, they can implement the getDigestSize() method. Finally, every algorithm class can have a number of implementation classes as subclasses. The implementation classes are concrete classes that implement all the required methods.

This second technique for decoupling is used in the .NET platform, and will be discussed in more detail in section 3.3.3.

Opaque and transparent keys and parameters

By using the techniques discussed in the previous paragraph, the cryptographic primitives can be made implementation independent. But the question arises how data items, like keys or parameters for cryptographic algorithms will be encapsulated. There are at least two approaches:

- an implementation class can fully encapsulate keys and parameters, and only give out opaque handles to these objects. This ensures that sensitive key material is not leaked to the application. But keys cannot be shared directly between providers.

- the cryptographic framework can define a standard representation of keys and parameters for each supported algorithm, and implementation classes can import and export these standardized keys. These standardized representations ensure interoperability between providers, but as a consequence the sensitive key material will be handled directly by the application.

In practice, it makes sense to support both approaches. A provider encapsulated representation is called *opaque* because the application is supposed to treat it as a black box containing implementation-dependent data. A standard representation class is called *transparent* because the application and all implementation objects know how to access the information inside it.

Cryptographic framework and CSP's

A cryptographic framework typically defines the engine classes (and possibly algorithm classes), the transparent key and parameter classes for algorithms that are supported by the framework, and the logic for finding and instantiating default implementations based on configuration information. A cryptographic service provider (CSP) is then a set of classes containing:

- implementation classes for some engine classes, e.g. a CSP could provide implementation classes for a MessageDigest engine class and for a DigitalSignature engine class

- possibly opaque key and parameter classes for the algorithms the provider supports, e.g. classes to represent DSA keys and parameters in a provider specific format

- possibly methods to convert back and forth between opaque and transparent representations

The cryptographic framework will typically also define the factory methods for the engine classes, and hence define how a specific CSP is selected if more than one is present, usually based on some kind of configuration mechanism.

By adding more and more CSP's to the framework, more and more cryptographic algorithms can be made available.

3.3.2 The Java Cryptography Architecture and Extensions

The Java Cryptography Architecture and Extensions (JCA/JCE) is an example of a cryptographic framework supporting the notion of CSP's. This framework is split in two parts for historical reasons:

- The Java Cryptography Architecture (JCA) provides support for cryptographic technology that was exportable under U.S. law before 2000.
- The Java Cryptography Extensions (JCE) provide support for cryptographic technology that was not exportable under U.S. law before 2000.

Since U.S. export laws for cryptography were relaxed in January 2000, the difference between JCA and JCE is not so important anymore, and we will not distinguish between the two.

Engine classes and Service Provider Interface classes

The JCA/JCE uses the Bridge pattern discussed above to obtain implementation independence, and the factory method approach to obtain algorithm independence. For every engine class (e.g. MessageDigest), the JCA/JCE also contains a corresponding Service Provider Interface (SPI) class (e.g. MessageDigestSpi). This SPI class is an abstract class that should be subclassed by implementation classes of the engine class. Hence an implementation object for the MessageDigest engine class will have type MessageDigestSpi. An object of class MessageDigest will be configured with such an implementation object and will delegate all cryptographic work to the implementation object via the interface of the SPI class. In the JCA/JCE the factory method for creating objects of engine classes is a static method of the engine class with name getInstance().

The JCA/JCE contains engine classes for:

- cryptographic primitives, including:
  - DigitalSignature
  - Cipher (for both symmetric and asymmetric encryption)
  - MessageDigest
  - MAC
  - SecureRandom (for secure random number generation)

- parameter generation, including:
  - KeyPairGenerator for generating public-private key pairs
  - KeyGenerator for generating symmetric keys
- key storage, including
  - KeyStore
- conversion between opaque and transparent keys, including:
  - KeyFactory for converting asymmetric keys
  - SecretKeyFactory for converting symmetric keys

For each of the engine classes, there is a corresponding SPI class.

Key and parameter classes and interfaces

The JCA/JCE supports both opaque and transparent key and parameter representation. Opaque representations should implement one of the Key interfaces: PrivateKey, PublicKey or SecretKey. Transparent representations are classes that are part of the JCA/JCE and implement one of the KeySpec interfaces. These classes have methods for accessing the contents of the key material. Examples of KeySpec classes are DesKeySpec, RSAPublicKeySpec, RSAPrivateKeySpec, etc...

To convert between opaque and transparent keys, so-called key factories are used. KeyFactory is an engine class, and implementations of it will convert KeySpec objects back and forth to Key objects.

For algorithm parameters (like e.g. an initialization vector for DES in CBC mode), similar opaque and transparent representations exist. But for these parameters, conversion between the two is done using methods in the AlgorithmParameters class: there is no separate factory class.

As a consequence of the different possible representations of key material in the JCA/JCE, there are at least three different ways to convert keys to a byte stream suitable for transmission over a communication channel:

- one can serialize a Key object, resulting in a serialized object that can only be deserialized on a virtual machine that has the same CSP installed (and may not even make sense to that CSP, for instance if the Key object contains an index into a hardware key register).
- one can serialize a KeySpec object, resulting in a serialized object that can be deserialized on any virtual machine that has the JCA/JCE.

- one can encode the key material according to a key encoding standard, resulting in a byte array that can in principle be parsed by anybody (since the format is standardized).

It should be clear that usually the third way is the preferred way. Key objects in the JCA/JCE can implement a getEncoded() method that performs a standardized encoding in a byte array.

Overall structure of the framework

Apart from the engine classes, key interfaces, and key and parameter classes, the JCA/JCE also contains a number of administrative classes.

The administrative classes form the structural backbone of the framework. The Security class is initialized from a configuration file and maintains the list of installed CSP's. New CSP's can be added at runtime if necessary. Every CSP consists of a number of implementation classes (subclasses of SPI classes, and implementations of key interfaces), and one distinguished master class that has information about the different algorithms that this CSP supports. This master class must subclass the Provider class of the JCA/JCE, and through the interface of Provider, the JCA/JCE can query the master class. The master class is essentially a Properties-like class. For any engine class that the provider implements, this property class has a property with name *engineclass.algorithm_name* and as value the name of the SPI class implementing the named algorithm for the named engine class. E.g. if a provider implements the SHA message digest, its master class will have a property named MessageDigest.SHA with as value the full name of the implementation class (that is a subclass of MessageDigestSpi).

Given this overall structure of the framework, it is possible to explain the inner working of the getInstance() factory methods. For example, the static getInstance("SHA") method call on MessageDigest queries the Security class for the available providers. Each provider is asked in turn if it has an implementation of the SHA message digest. The providers are queried in an order specified in the Security configuration file. If a provider answers with a class name, that specific class is instantiated, and a new MessageDigest object is instantiated and configured with the newly created MessageDigestSPI object. It is then ready to start computing message digests.

The getInstance() method is overloaded, so that an application can either:

- just request an instance implementing a given algorithm (as in the example above)
- or request an instance implementing a given algorithm and implemented by a specific provider

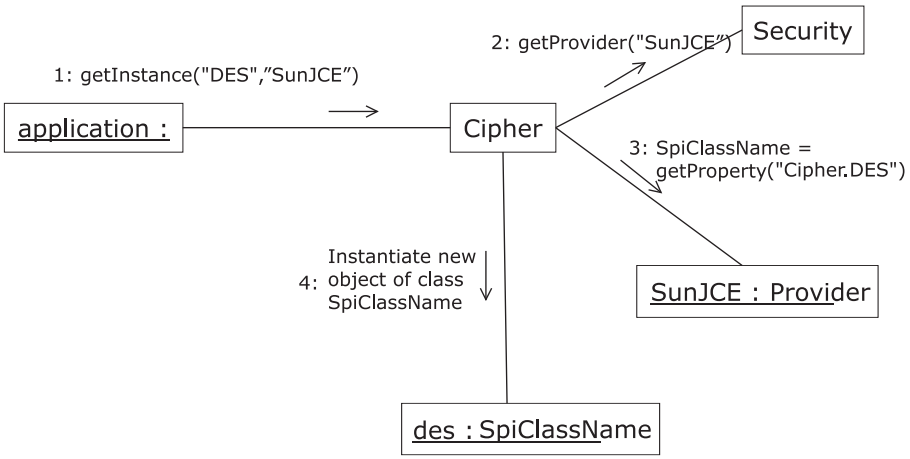

Figure 3.7: Interaction diagram for getInstance()

The interaction diagram in Figure 3.7 summarizes the working of the getInstance() method in the second case.

SunJCE is the name of a provider that is included with the reference implementation of the JCE from Sun.

Utility classes

Finally the JCA/JCE contains a number of utility classes, including:

- CipherInputStream and CipherOutputStream, that decorate Streams with a Cipher, such that everything written to the OutputStream is encrypted first, and everything read from the InputStream is decrypted first.
- SignedObject, that computes signed serializations of objects
- SealedObject, that computes encrypted serializations of objects

Example programs

Each of the engine classes of the JCA/JCE is documented in the API documentation. In this section, we give one simple example of their use.

The example shows how to use KeyGenerators, Ciphers, CipherInputStreams and CipherOutputStreams.

Listing 3.1: JCE/JCA Example

```
/* ExampleJCE.java
 * Sample program showing how to use the JCA/JCE
```

```
 * This program accepts one filename as parameter.
 * It encrypts the entire file using a randomly
 * generated key.
 * Then it decrypts the file again. The resulting
 * decrypted file should of course be identical to
 * the original file.
 */

import javax.crypto.*;
import java.io.*;

public class ExampleJCE {

public static void main (String args[]) {
 try {
  String filename = args[0];
  // Instantiate a KeyGenerator for the DES algorithm,
  // and generate a new key
  KeyGenerator keygen = KeyGenerator.getInstance("DES");
  SecretKey desKey = keygen.generateKey();

  // Instantiate a Cipher using the DES algorithm
  Cipher desCipher;
  desCipher = Cipher.getInstance("DES/CBC");

  // Initialize cipher for encryption with generated key
  desCipher.init(Cipher.ENCRYPT_MODE, desKey);

  // Create a CipherOutputStream to a file
  CipherOutputStream cos = new CipherOutputStream
        (new FileOutputStream(filename+".enc"),desCipher);

  // Read the entire input file and write all read bytes
  // to the encrypted output file
  InputStream is = new FileInputStream(filename);
  byte[] buffer = new byte[1024];
  int bytesread;
  while ((bytesread=is.read(buffer)) != -1)
    cos.write(buffer,0,bytesread);
  is.close();
  cos.close();

  // Now initialize cipher for decryption with the same key
```

```
  desCipher.init(Cipher.DECRYPT_MODE, desKey);

  // Open the encrypted file through a CipherInputStream
  CipherInputStream cis = new CipherInputStream
        (new FileInputStream(filename+".enc"),desCipher);

  // Decrypt the entire file and save to filename.dec
  OutputStream os = new FileOutputStream(filename+".dec");
  while ((bytesread=cis.read(buffer)) != -1)
    os.write(buffer,0,bytesread);
  cis.close();
  os.close();

 // catch-all exception handler
 } catch (Exception e) {System.out.println(e);};
}
}
```

3.3.3 The .NET Cryptographic API

The .NET cryptographic API, located in the system.security.cryptography namespace uses the inheritance based decoupling of algorithms and implementations. The class library contains a collection of abstract engine classes (e.g. SymmetricAlgorithm), and for each of these engine classes a number of abstract algorithm subclasses (e.g. TripleDes, Des, Rijndael). Each of the abstract algorithm classes has in turn a number of concrete subclasses (e.g. RijndaelManaged, an implementation of Rijndael in managed code).

Applications should strive to use the implementation objects (e.g. a RijndaelManaged object) through the interface of one of its superclasses: if an application uses an implementation object only through the abstract engine class interface, the application is algorithm independent. Similarly, if the application uses the object only through the abstract algorithm class interface, the application is still implementation independent. Once the application uses methods that are only available in a specific implementation class, algorithm and implementation independence are lost.

Cryptography classes for bulk data processing

The engine classes that are typically used for bulk data processing (i.e. symmetric encryption and decryption, message digests and MAC's) are all structured around the ICryptoTransform interface and the CryptoStream class.

ICryptoTransform is a very simple interface that contains 2 methods and a few properties. The two most important properties are the input block size and the output block size, and the two methods are TransformBlock and TransformFinalBlock. Both these methods transform a byte array into another byte array. Objects that perform symmetric encryption or hashing make their services available as implementations of this interface:

- For symmetric encryption and decryption, TransformBlock turns plaintext into cipher text and vice versa. TransformFinalBlock performs any padding that is required.
- For message digests and MAC's, TransformBlock copies the input byte array to the output byte array, but computes the hash of all data that has been passed through TransformBlock. TransformFinalBlock then computes the final hash value, and makes it available as a property.

ICryptoTransforms are usually used in combination with the CryptoStream class. This class decorates another stream with an ICryptoTransform: all data that goes through the decorated stream is first chopped in blocks of appropriate size, and passed through the ICryptoTransform.

CryptoStreams make it particularly easy to do combinations of cryptographic operations on streams of data. Example code will be shown in section 3.3.3.

The following engine classes provide their services through ICryptoTransform:

- SymmetricAlgorithm, for the symmetric encryption and decryption primitives. SymmetricAlgorithm defines abstract properties and methods common to all symmetric encryption algorithms, such as a key property, and a method that returns an encryptor/decryptor, an object that implements ICryptoTransform.

  Algorithm classes under SymmetricAlgorithm include TripleDES, DES and Rijndael.

- HashAlgorithm, for message digests. HashAlgorithm implements ICryptoTransform and makes the hash available as a property.

  Algorithm classes under HashAlgorithm include SHA1 and MD5.

- KeyedHashAlgorithm, for MAC's. KeyedHashAlgorithm itself inherits from HashAlgorithm.

  Algorithm classes under KeyedHashAlgorithm include HMACSHA1 and MACTripleDES.

To remain implementation and algorithm independent, each of the abstract classes provides a Create() factory method that can be used to instantiate a (configurable) default implementation object.

The key data structures for symmetric encryption are all transparent in .NET: keys are just represented as byte arrays.

Engine classes for generating keys

The .NET cryptography classes include two engine classes for generating key material:

- RandomNumberGenerator for generating secure random numbers. A Create() factory method is again provided to remain algorithm independent.

- DeriveBytes for deriving key material from passwords. DeriveBytes does not have a factory method: its subclasses must be constructed explicitly.

Engine classes for asymmetric cryptography

The following engine classes make the asymmetric primitives available:

- AsymetricKeyExchangeFormatter, for asymmetric encryption and AssymetricKeyExchangeDeformatter for the corresponding decryption. Implementation objects for these classes implement asymmetric encryption including the necessary padding, and asymmetric decryption including the necessary checks on the padding material after decryption.

- AsymetricSignatureFormatter and AsymmetricSignatureDeformatter, for digital signatures.

Since asymmetric encryption and digital signatures can be implemented on top of the same mathematical transformation (but with different padding and formatting), the .NET library also contains an AsymmetricAlgorithm engine class, that encapsulates raw asymmetric operations without padding.

The idea is that you first construct an AsymmetricAlgorithm that encapsulates the raw mathematical transformation. Then you pass that object to a constructor of one of the formatter or deformatter classes to construct an object that implements the full primitive, including padding.

In the current status of the .NET crypto library, the implementation classes for the asymmetric crypto operations delegate to the Windows CryptoAPI, the cryptographic library in the Windows operating system. However, since CryptoAPI does not expose the raw mathematical operations, the current implementation classes for AsymmetricAlgorithm do *not* implement these raw operations, but provide additional methods that do padded cryptographic operations. Of course, as soon as a developer uses one of these additional methods, his code is no longer provider independent.

Example programs

The following example program illustrates the use of SymmetricAlgorithm and CryptoStream.

Listing 3.2: .NET Crypto Example

```
using System;
using System.Security.Cryptography;
using System.IO;


class ExampleNETCrypto
{
/* ExampleNETCrypto.cs
 * Sample program showing how to use the .NET
 * cryptography namespace
 * This program accepts one filename as parameter.
 * It encrypts the entire file using a randomly
 * generated key.
 * Then it decrypts the file again. The resulting
 * decrypted file should of course be identical to
 * the original file.
 */
static void Main(string[] args)
{
string filename = args[0];
// Create a default implementation object for
// a symmetric encryption algorithm
// It is automatically initialized with a random key
SymmetricAlgorithm cipher = SymmetricAlgorithm.Create();
// Open a file output stream, and wrap an encrypting crypto
// stream around it
FileStream outStream =
  new FileStream(filename + ".enc", FileMode.Create);
```

```
CryptoStream encOutStream = new CryptoStream
  (outStream, cipher.CreateEncryptor(),CryptoStreamMode.Write);
// Open the input stream
FileStream inStream = new FileStream(filename, FileMode.Open);
// And just copy bytes ...
byte[] buffer = new byte[1024];
int bytesread;
do
        {
        bytesread = inStream.Read(buffer,0,1024);
        encOutStream.Write(buffer,0,bytesread);
        } while (bytesread > 0);
encOutStream.Close();
inStream.Close();

// Now we will decrypt the encrypted file again.
// Open a new file that will contain the decrypted output
// and wrap it in a decrypting crypto stream
FileStream outStream2 =
  new FileStream(filename + ".dec",FileMode.Create);
CryptoStream decOutStream2 = new CryptoStream
  (outStream2,cipher.CreateDecryptor(),CryptoStreamMode.Write);
// Open the encrypted file
FileStream inStream2 =
  new FileStream(filename + ".enc", FileMode.Open);
// And just copy bytes ...
do
        {
        bytesread = inStream2.Read(buffer,0,1024);
        decOutStream2.Write(buffer,0,bytesread);
        } while (bytesread > 0);
inStream2.Close();
decOutStream2.Close();
}
}
```

The second example illustrates the use of asymmetric cryptography in .NET. The program in listing 3.3 signs a file and puts the signature in a separate file with .sig extension. The program in listing 3.4 verifies such a signature.

Listing 3.3: .NET Signing Example

```
using System;
using System.Security.Cryptography;
```

```
using System.IO;

class ExampleNETSign
{
/* ExampleNETSign.cs
 * Sample program showing how to use the .NET
 * cryptography namespace
 * This program accepts one filename as parameter.
 * It creates a public-private keypair and outputs
 *  - the private and public key in file key.priv
 *  - the public key only in file key.pub
 *  - the signature of the file in <filename>.sig
 */

static void Main(string[] args)
{
  string filename = args[0];
  // Create a default implementation object for
  // an  asymmetric algorithm
  // It is automatically initialized with a random key
  AsymmetricAlgorithm cipher = DSA.Create();

  // Save public and private key in key.priv
  StreamWriter outStream = new StreamWriter("key.priv");
  outStream.Write(cipher.ToXmlString(true));
  outStream.Close();

  // Save public key only in key.pub
  outStream = new StreamWriter("key.pub");
  outStream.Write(cipher.ToXmlString(false));
  outStream.Close();

  AsymmetricSignatureFormatter asf =
    new DSASignatureFormatter(cipher);
  SHA1 sha1 = SHA1.Create();
  // Open the input stream
  FileStream inStream =
    new FileStream(filename, FileMode.Open);
  byte[] sig =
    asf.CreateSignature(sha1.ComputeHash(inStream));

  // Save the signature
  FileStream outStream2 =
```

```
    new FileStream(filename + ".sig",FileMode.Create);
  outStream2.Write(sig,0,sig.Length);
  outStream2.Close();
}
}
```

Listing 3.4: .NET Verify Example

```
using System;
using System.Security.Cryptography;
using System.IO;

class ExampleNETVerify
{
/* ExampleNETVerify.cs
 * Sample program showing how to use the .NET
 * cryptography namespace
 * This program accepts one filename as parameter.
 * It inputs a DSA public key from file key.pub
 * and a signature from <filename>.sig and verifies
 * if this is a valid signature over the file
 */

static void Main(string[] args)
{
  string filename = args[0];
  // Create a default implementation object for
  // an  asymmetric algorithm
  AsymmetricAlgorithm cipher = DSA.Create();

  // Retrieve public key from key.pub
  StreamReader inStream = new StreamReader("key.pub");
  string pubkey = inStream.ReadToEnd();
  cipher.FromXmlString(pubkey);

  // Retrieve the signature from <filename>.sig
  FileStream inStream2 =
    new FileStream(filename + ".sig", FileMode.Open);
  byte [] sig = new byte[inStream2.Length];
  inStream2.Read(sig,0,sig.Length);

  AsymmetricSignatureDeformatter asd =
    new DSASignatureDeformatter(cipher);
  SHA1 sha1 = SHA1.Create();
```

```
    // Open the input stream and recreate the hash
    FileStream inStream3 =
      new FileStream(filename, FileMode.Open);
    byte[] hash = sha1.ComputeHash(inStream3);

    // Verify the signature
    if (asd.VerifySignature(hash, sig))
      Console.WriteLine("Signature OK!");
    else Console.WriteLine("Signature NOT OK!");
  }
}
```

3.3.4 Microsoft Windows CryptoAPI

[TODO Must still be written]

3.4 Further Reading

Books The "Handbook of applied cryptography" ([8]) is considered the bible of cryptography. If you are working on an application whose security relies on cryptographic technology, you should have this book on your desk. "Applied Cryptography" ([11]) is another high quality book about cryptography. This book is much more pleasant to read, but does not have the same amount of technical detail as the handbook. The "Java Security Handbook" ([5]) has an extensive discussion of the JCA/JCE.

The ".NET Framework Security" book ([7]) has a detailed discussion of the cryptographic API's in .NET.

[TODO Must still add books for Crypto API]

URL's Via `http://java.sun.com/security` you can find the Reference Guide and Programmers Guide for the JCE and JCA. Also the javadoc documentation for the JCA and JCE is extremely useful.

Documentation about the .NET cryptography namespace, and about Windows CryptoAPI is available via the MSDN library ([33]).

Standards TODO PKCS standards

Papers TODO Reference to "Why Cryptosystems Fail"

Chapter 4

Cryptographic Protocols

A protocol is a *distributed algorithm*: a number of concurrently active parties perform local computation and communicate with each other to achieve some common goal. A *cryptographic* protocol is a protocol that uses cryptographic primitives, and typically achieves some security-related goal. Examples are protocols for:

- secure communication: exchanging data between two parties in such a way that it is impossible to eavesdrop on the communication or to tamper with the communicated data.
- entity authentication: verifying the identity of the other party in a communication.
- key distribution: safely distributing a secret key to authorized parties.
- electronic payment: securely transferring an amount of money over a digital communication channel

Being familiar with existing cryptographic protocols is useful for the software engineer for at least two reasons:

1. If you have to achieve a security related goal in your own distributed applications, it is useful to know what existing protocols you could use. From that perspective, it would make sense to study cryptographic protocols as black boxes, in a similar way as we have studied cryptographic primitives.
2. But cryptographic protocols are at the same time good examples of how you can use cryptographic primitives to achieve a higher-level goal. They are a kind of "cryptographic design case studies", and as such give

a good indication of the complexity and subtlety of designing systems based on cryptographic primitives.

Hence, we will study cryptographic protocols in more detail than cryptographic primitives: for some protocols, we will also discuss their inner working. But the reader should keep in mind that the example protocols that are discussed here are only meant to convey the basic principles underlying these protocols. They are by no means "industrial strength" protocols. In the section on real-world protocols (section 4.5), we discuss some of the additional issues that need to be taken into account when building production quality protocols.

4.1 Secure Communication

4.1.1 Message-oriented communication

4.1.2 Stream-oriented communication

4.2 Entity Authentication

Entity authentication is the process of verifying a claimed identity. It is useful to distinguish a number of different instances of this problem, that are similar but still typically require different solutions.

The first instance is the case where a computer authenticates a local user. In this case, the computer can authenticate the user in at least the following ways:

1. by verifying that the user knows a certain secret, e.g. a password.
2. by measuring something that is supposed to be a unique attribute of a person, e.g. a fingerprint.
3. by verifying that the user possesses a certain token, e.g. a smartcard.
4. by measuring something only the legitimate user can do, e.g. measuring a handwritten signature.
5. by a combination of two or more of the above, e.g. a smartcard combined with a secret PIN.

The second instance is the case where a computer authenticates another computer. In this case, authentication almost always relies on the verification of knowledge of a secret. This secret will often be a cryptographic key.

A third interesting instance is the case where a user must authenticate to a remote computer. In this case, the user is typically interacting with a local computer (his 'workstation') that is in turn communicating with the remote computer. How authentication should be done in this case depends heavily on the amount of trust one can have in the workstation. The user will typically need some amount of trust in the workstation, and will have the workstation assist him in proving his identity to the remote computer. Often, the user will temporarily delegate his identity to the local workstation, and allow the workstation to authenticate as the user to remote computers. This temporary delegation of identity is often referred to as *single sign-on*: the user authenticates only once per session, and for the rest of the session, his workstation will take care of authentication in his place.

We will first consider in this section the case of computer-computer authentication, including the possibility that one of the computers is actually a workstation authenticating as its current user. After discussing key establishment (section 4.3), we will return to the case of single sign-on (section 4.4).

In the computer-computer case, authentication is typically achieved by verifying whether the other party knows a certain secret that is only known to the legitimate party. This verification of knowledge of the secret must be secure against eavesdropping: an attacker listening in on the authentication dialogue should not learn the secret. Moreover it should be secure against *replay*: an attacker recording all communication between the two computers should not be able to authenticate succesfully by using ("replaying") some of this recorded material.

Hence, entity authentication is done by computing a message, such that: (1) the message can only be computed if you know a certain secret key but does not leak that secret, and (2) the message varies with every authentication round.

To achieve the first property, one can use a cryptographic primitive, typically symmetric encryption, a digital signature or a MAC. To achieve the second property, one uses a time-varying parameter. This can be:

1. A *timestamp*: include the current time in the constructed message.
2. A *nonce*: a nonce is a number that is never reused for the same purpose, typically implemented as a random number of sufficiently large bitlength to make the odds of repeating very small.
3. A *counter* that increases with each authentication round.

By making different choices for the cryptographic primitive, the time-varying parameter, and possibly additional content of the constructed message, a large variety of authentication protocols is possible, each with its

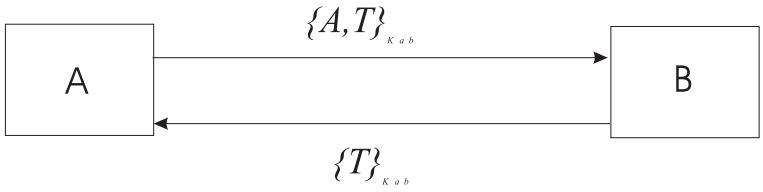

Figure 4.1: Example authentication protocol with timestamps

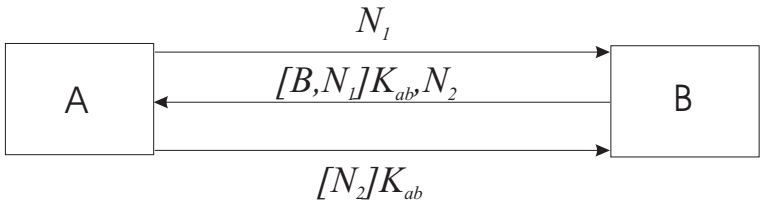

Figure 4.2: Example authentication protocol with nonces

advantages and disadvantages. We will discuss two example protocols in some more detail. For a more thorough treatment, the reader can consult chapter 10 of the *Handbook of Applied Cryptography*([8]).

The example protocol in figure 4.1 uses symmetric encryption and timestamps (recall the notational conventions from the chapter on cryptographic primitives). T stands for the timestamp: it is a representation of the date and time the message was constructed. This protocol assumes that each pair of entities wishing to authenticate each other have agreed on a secret key in advance. We denote the key shared between A and B as K_{ab}. It also assumes that all parties have access to a reliable clock. The protocol authenticates A to B and vice-versa. This is called *mutual* authentication. If it is not necessary that B authenticates to A (i.e. *one-way* authentication suffices), the second message is superfluous.

B can be sure of A's identity, because:

1. only A could have done the encryption (we assume nobody but A and B know the key K_{ab}).

2. the message is fresh: it is not a replay of an old message of A (we assume B checks the timestamp in the message, and rejects it if it is too old).

The second example protocol in figure 4.2 uses MAC's and nonces. It does not rely on synchronized clocks, but requires more messages.

A can be sure of B's identity, because:

1. only B could have computed the MAC.

2. the message is fresh, since it includes the nonce N_1 that A just sent to B.

A protocol of this kind is often called a *challenge-response* protocol: the nonce N_1 is a kind of "challenge" that A sends to B. If B can respond correctly to the challenge, then he really is B.

Keep in mind that these two example authentication protocols are just simple "proof-of-concepts". A real-life authentication protocol is typically more complex. We come back to this issue in section 4.5.

4.3 Key Establishment

The entity authentication protocols discussed in the previous section assumed that any two entities on the network shared a secret key. If there are order n entities, this implies order n^2 shared keys. For large n, this quickly becomes unmanageable. What is needed is some way to establish these keys at run-time, at the moment they are needed.

Key establishment is the process of making available a shared secret key to two parties. Key establishemnt is a complex and subtle process. It can be realized in a number of different ways, but the properties of the shared secret may be different depending on the process used to establish the secret.

Key transport is a kind of key establishment where a key is generated (or obtained in some other way) by one party, and then securely transported to the other party. *Key agreement* on the other hand is a kind of key establishment where the two parties have essential influence on the final established secret. (I.e. it is *not* the case that one of them gets to choose the secret.)

We say the shared secret key is *authenticated* if one entity is assured that only another known entity could get hold of the key. We say the key is *confirmed* if one entity has actual proof that another entity holds the key. The most desirable situation to achieve if you want to establish keys for entity authentication is the establishment of authenticated and confirmed keys. If the key is not authenticated, you cannot use it for entity authentication. If the key is not confirmed, you are not sure that the other party actually has the key.

We will discuss two example key establishment protocols. For a deeper treatment of key establishment, consult chapter 12 of the Handbook of Applied Cryptography([8]).

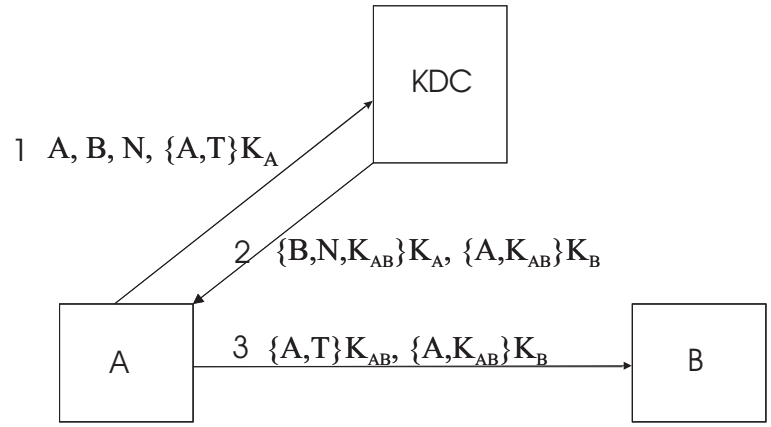

Figure 4.3: Example key establishment protocol with symmetric cryptography

4.3.1 Key establishment using symmetric techniques and a trusted party

The first key establishment technique assumes the presence of a third party, trusted by all entities on the network. The trusted third party (TTP) will generate keys on demand and transport them to the appropriate entities. We call this kind of key transport *key distribution* and the TTP is called the *Key Distribution Centre (KDC)*.

In our example, the KDC will generate keys and transport them to the appropriate entities using symmetric cryptography. To make this work, we must assume that each entity on the network already shares a secret key with the KDC. Let us call the key that entity A shares with the KDC K_A. (Note that this reduces the a priori key distribution burden from order n^2 to order n)

The protocol used to establish a key between entities A and B is given in figure 4.3.

A sends to the KDC a message consisting of:

- the names of the two entities involved: A and B.
- a nonce N that will serve to prove freshness of the distributed key.
- A also authenticates to the KDC using the shared secret K_A.

The KDC now generates a new *session key* K_{AB} that is intended to be used between A and B. It packages the key securely, once for A and once for B, by encrypting it using the keys the KDC shares with A and B. In the

Figure 4.4: Example key establishment protocol with asymmetric cryptography

package for A, the KDC also includes the nonce, so that A can be assured of the freshness of the key. The KDC sends both packages to A.

A now unpacks the first part of the message (the part that is encrypted under K_A), and obtains the session key K_{AB}. A forwards the second part of the message to B, together with an authentication message based on K_{AB}. B can decrypt his package, obtaining the session key. A and B now share a shared secret K_{AB}. This secret is authenticated, because the packages from the KDC carry the names of the other entity owning the key, and each entity trusts the KDC not to cheat on this. After B verifies the authentication message, B also has key conformation: he knows that A actually has the key. A does not yet have key confirmation: an attacker might have destroyed the last message to B. To achieve key confirmation, B could send back an authentication message to A.

Again, keep in mind that this example protocol is intended to explain the basic principles of key transport using a KDC and symmetric cryptography. A real-life protocol will be much more complex (see section 4.5).

4.3.2 Key establishment using asymmetric techniques and a PKI

Asymmetric cryptography can make key establishment simpler. Under the assumption that public keys of all entities are widely known (e.g. published in some kind of phone book), a symmetric key can be distributed between A and B using the protocol in figure 4.4.

A generates the session key K_{AB} and transports it to B. The session key is encrypted with B's public key, making sure that nobody but B can decrypt it. The encrypted session key, together with B's name, a lifetime for the key (LT), and a nonce are signed by A and sent to B. By verifying the signature, B knows the message originated from A. The lifetime and nonce together guarantuee freshness of the key: if the lifetime has passed, B refuses the key. If the lifetime is still OK, B verifies whether he has seen the nonce N before (B is supposed to store all nonces for the duration of the lifetime). If he has not seen the nonce yet, the message is fresh, and B accepts it.

Figure 4.5: Man in the middle attack

This protocol achieves authentication of A (B can be sure he is talking to A), and authenticated key transport from A to B. It does not achieve key confirmation.

The protocol shows that asymmetric cryptography can make the key establishment process a lot simpler. It relied however on the assumption that public keys of all entities were widely known. This is a quite strong assumption. If an attacker C can trick A into believing that B's public key is PK_C instead of PK_B, then C can of course read the session key generated by A. If C can also trick B in believing that A's public key is also PK_C, then C can perform a man-in-the-middle attack as shown in figure 4.5.

C pretends to be B to A and pretends to be A to B. Neither A nor B will detect this. And C will of course be able to see all traffic encrypted with the session key later on.

Hence, to make public-key based systems work, it is of great importance to be able to distribute the public keys of entities in a secure way. A *Public Key Infrastructure (PKI)* is an infrastructure to ensure secure dissemination of public key information. We discuss PKI's in detail in section 4.6. It will turn out that setting up a reliable and usable PKI is by no means trivial, and hence key establishment remains a hard problem even with asymmetric cryptographic techniques.

4.4 Single Sign On

We have seen how computers can authenticate each other based on the verification of the knowledge of some secret. This secret is typically a cryptographic key, and the proof of knowledge is given by performing some cryptographic operations. It is clear that we can not apply the same idea to computer-user authentication: users cannot remember cryptographic keys (i.e. long random bitstrings), and they cannot perform cryptographic operations. So the typical situation will be that a user authenticates to his local workstation, and (temporarily) delegates his identity to the workstation. All further authentications to remote computers are done by the workstation using techniques as discussed in the previous section. This way of working is called *single sign-on* : the user authenticates once to his local workstation (hence *single* sign-on), and can then access all servers on the network because

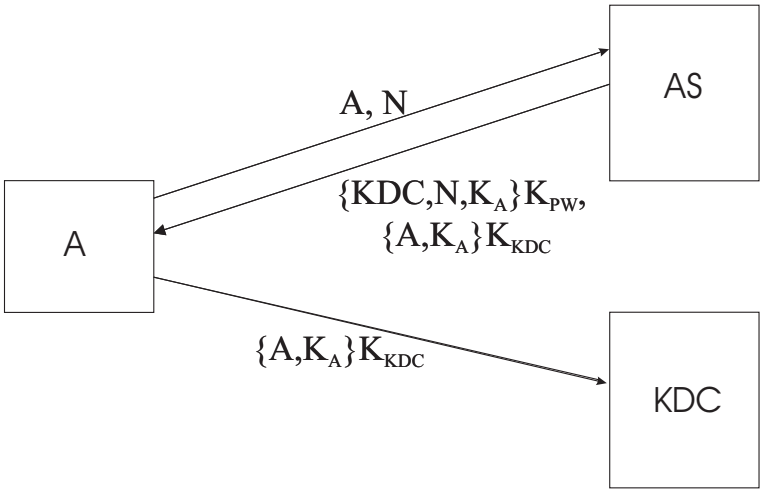

Figure 4.6: Example single sign-on protocol

his workstation will take care of authentication to these servers.

In this section, we discuss one possible way to implement this idea in the context of key distribution by means of a KDC using symmetric cryptography. We will use password authentication for the workstation-user authentication. The goal is to assign to a user A a temporary key K_A shared between the workstation of the user and a KDC. The key K_A will only be decipherable by the workstation if the entered user password is correct. Once K_A is distributed, the workstation can perform authentications of A to other parties.

The idea is the following: at logon time, a trusted third party, the *Authentication Server (AS)* will be contacted, and in a way similar to the key distribution technique discussed in section 4.3.1, the AS will generate a temporary cryptographic key K_A for A, and transport it in a secure way to the user's workstation and to the KDC.

The protocol is shown in figure 4.6. Initially, the AS knows all user passwords, and can derive a cryptographic key from these passwords. Moreover, the AS shares a key K_{KDC} with the KDC.

At logon time, A's workstation sends A's name and a nonce to the AS. The AS looks up A's password and derives a key K_{PW} from it. Then the AS generates a temporary key K_A, and packages it in a secure way, once encrypted with K_{PW} and once with K_{KDC}. A's workstation asks A for his password, derives the key K_{PW} from it, and decrypts the first part of the message it got back from the AS. If the password (and hence K_{PW}) was correct, the workstation now knows K_A. The workstation forwards the second part of the AS's response to the KDC. The KDC can extract the

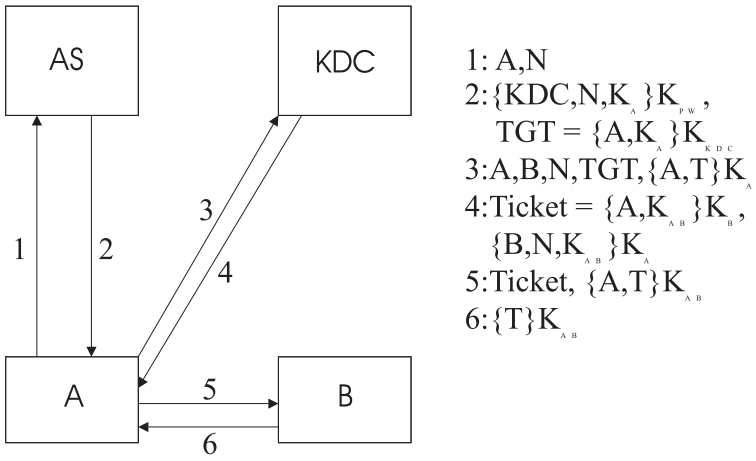

Figure 4.7: Kerberos protocol

K_A using K_{KDC}. As a consequence, K_A is now a shared secret between the KDC and A's workstation. Using the key establishment protocol from section 4.3.1, authenticated session keys can now be established between A (or his workstation) and other entities.

4.5 Real-world protocols

4.5.1 Kerberos

Kerberos is an authentication and key distribution system developed at MIT. It is one of the most widely used symmetric key based systems for client-server based computer systems. Kerberos uses techniques very similar to the symmetric key techniques we discussed in previous sections: it uses a single sign-on protocol based on the protocol in figure 4.6, a key distribution protocol based on the protocol in figure 4.3 and an authentication protocol based on the protocol in figure 4.1.

Kerberos uses some specific terminology to talk about the messages being exchanged in these protocols. An *authenticator* is a message like $\{A, T\}_{K_{AB}}$, used to prove your identity on the basis of a shared secret. A *ticket* from A for B is a message like $\{A, K_{AB}\}_{K_B}$ used to transport a session key to B. The name ticket comes from the specific way in which this message is used in Kerberos. We will discuss this further on. A *ticket granting ticket (TGT)* is a message like $\{A, K_A\}_{K_{KDC}}$ used to transport the key K_A generated during the login procedure to the KDC.

The overall working of Kerberos is summarized in figure 4.7.

Kerberos is strictly client-server based, and authentication or key distribution is always triggered by a client. A client is typically a process running on some workstation on behalf of some user, but a server can in turn act as a client to other servers (e.g. a mail server could be a client of a file server). Let us discuss the Kerberos protocol by following what happens when a user A wants to use a local workstation to read his mail on a mailserver B. When the user logs on to the local workstation, the single sign-on protocol is executed. The workstation, after receiving the reply from the AS, caches the key K_A and the TGT locally. To request a session key to talk to B, the workstation sends (on behalf of A) an authenticator, the TGT, B's name and a nonce to the KDC. The KDC unwraps the TGT to find K_A. The KDC also shares a key K_B with the mail server B. Using these two keys, the KDC sends back a reply to A's workstation, containing among others a ticket from A for B. The workstation unpacks the key K_{AB} and caches the key and the ticket locally. Then A's workstation sends the ticket and an authenticator to B. B unpacks the ticket to find K_{AB}, and using K_{AB}, B can verify A's identity.

The KDC and B do not remember the keys they unpack from tickets: it is a client's responsibility to resend the ticket every time. In that way, servers need to keep less state information. The name 'ticket' comes from the fact that clients need to present a ticket each time they access a server. The name 'ticket-granting ticket' comes from the fact that a client needs to present the TGT whenever he requests a new ticket.

Once A's workstation has established a connection with B, and the ticket and authenticator have been sent and validated by B, A and B share a session key that they can use for other purposes than just authentication. They can use the key to construct integrity-controlled or confidential messages to each other (called *safe* and *private* messages in Kerberos).

The content of messages in the Kerberos protocols have been simplified in the figure above. In reality, Kerberos messages are much more complex. Here are some of the simplifications we have made:

- Tickets always have a finite lifetime in Kerberos (typically a few hours). Hence a ticket has an additional field to indicate it's validity period.
- Messages carry version and message type information with them.
- Kerberos has additional functionality that we have not discussed here, like the possibility to delegate your identity to a server.
- Kerberos allows for more than one AS and KDC, and foresees ways to connect these multiple trusted third parties together.

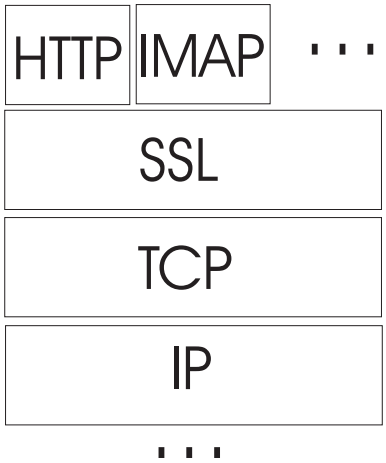

Figure 4.8: SSL in the protocol stack

- ...

For a full discussion of the Kerberos protocol, we point the reader to the Kerberos website ([26]).

4.5.2 SSL/TLS

The Secure Socket Layer (SSL) protocol was originally developed by Netscape to secure communication between a web browser and a web server. Earlier versions of the protocol contained serious flaws, but after public review, version 3 of the protocol was considered reasonably secure. The Internet Engineering Task Force started from SSLv3 to produce the Transport Layer Security (TLS) protocol, an Internet Standard for secure communication at the socket level. SSL and TLS are very similar protocols.

SSL fits in the TCP/IP protocol stack between TCP and application protocols like HTTP or IMAP as shown in figure 4.8.

Setting up an SSL connection happens in two phases. In the first phase, the handshake protocol is executed to establish a *security context* at client and server. This context encapsulates all information necessary to communicate securely, like session keys, algorithms in use, etc .... The handshake protocol is a complicated protocol (for a detailed description, see [13] or the TLS standard [**?**]). It starts of with a negotiation between client and server, to decide on the key establishment method that will be used, and the cryptographic algorithms that will be used. It then proceeds to establish, (possibly) authenticate and confirm a so-called *master secret*. This master secret is a 48 byte secret shared between client and server, and used to derive

further session keys from it. SSL uses different keys for integrity control and encryption, and has the ability to change session keys after a certain time has passed or a certain amount of data has been sent.

In the second phase, SSL accepts data from the application protocol above it (e.g. HTTP), and sends it to the other side in a secure way using the established security context. The application data is first fragmented in fixed size blocks. Then a MAC is added for each block and the block plus the MAC are encrypted. Finally, an SSL header is prefixed, and the resulting message is sent to the other side, where it is decrypted, the MAC is checked, and the blocks are reassembled before being passed to the application layer.

SSL supports a number of key establishment methods, but the most widely used method is just key transport using RSA. Just like the protocol in section 4.3.2, this key establishment method is vulnerable to a man in the middle attack if public keys are not disseminated in a secure way. In other words, to be secure, SSL requires a good public key infrastructure. In the absence of a good PKI, man in the middle attacks are possible against SSL, and tools to launch such attacks are available on the Internet.

4.6 Public Key Infrastructures

We have seen in previous sections that protocols using asymmetric cryptography often rely on a secure public key directory: they assume that the correct public key for any entity is easily available. A *Public Key Infrastructure (PKI)* is an infrastructure designed to realize this assumption. In this section, we will discuss different ways in which a PKI can be set up.

4.6.1 Basic PKI concepts

Essentially, a PKI has to make sure that every entity can get hold of another entity's public key, and have a reasonable amount of trust that this key is the correct one. In other words, it should not be possible for an attacker to trick somebody into using an incorrect public key to talk to somebody else.

Locating somebody else's public key can happen in a number of ways. For example, the other entity's public key can be sent to you by the other entity. Or you can find the key in a public directory of some kind.

Making sure that you can trust the correctness of the key is harder. A first possibility is that you have *direct trust* in a key, because you used some out-of-band mechanism to verify the key (e.g. you call the owner of the key, and verify a hash of the key over the phone) or because you got the key over a trusted channel (e.g. you get the key on a floppy or CDROM that is sent

in a physically secure way). Establishing direct trust in a key is possible for a limited number of keys, but it does not scale very well.

Therefore, most PKI's use some kind of trusted third party that tells you that the key is good. The task of this third party is to *certify* bindings between public keys and their owners. Hence, the third party is called a *Certification Authority (CA)*. The CA certifies a binding by issueing a *certificate*. A certificate is in its simplest form a data record consisting of a public key and the name of the owner of that public key, signed by the CA. The signature by the CA is supposed to guarantee the integrity of the binding. But of course to verify a signature, you need the public key of the signer. So you need the correct public key of the CA to verify a certificate. Typically, you need to establish direct trust in the public key of the CA.

In practice, the role of CA is played by commercial organisations like VeriSign or GlobalSign, and the public keys of these commercial CA's is baked into your operating system or browser software. Hence, you have direct trust in these public keys, because you assume that nobody can tamper with the contents of, for instance, your Windows 2000 CDROM.

It is important to realize that certificates can be distributed (e.g. through an online directory) in an insecure way: if an attacker would tamper with a certificate, this would be detected, because the signature of the CA would no longer be valid.

Hence a very simple PKI could consist of one globally trusted CA, and an untrusted directory that contains all certificates issued by the CA. Introducing a new entity in this system requires interaction with the CA. The entity should present its public key to the CA. The CA should authenticate the entity, should verify that the entity indeed owns the corresponding private key, and can then issue a certificate for the new entity.

Lookup and validation of a certificate does not require interaction with the CA: any entity can access the directory and retrieve certificates, and any entity has the CA public key to verify retrieved certificates.

This simple example of a PKI with one CA illustrates the essential concepts underlying a PKI. But real-world PKI's are much more complex. We discuss some of the issues in the following sections.

4.6.2 Trust models

The assumption of a single CA trusted by everybody does not scale very well. Hence, a real-world PKI will have to deal with the fact that multiple CA's exist. There are a number of ways in which a PKI can support multiple CA's.

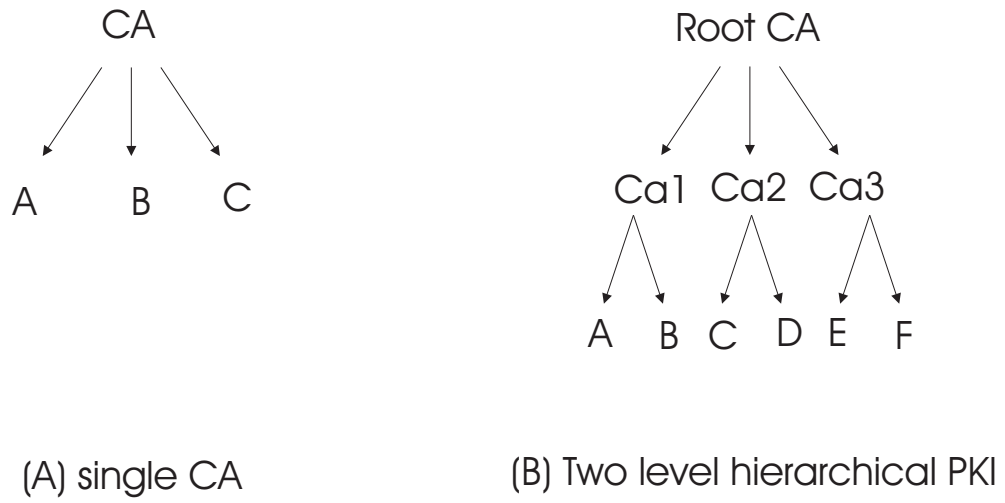

Figure 4.9: Hierarchical PKI

Hierarchical trust model

A straightforward way to deal with multiple CA's is to use a two level hierarchy: there are multiple CA's each certifying a group of entities, and there is a single root CA certifying each of the other CA's. If we indicate the fact that X certifies Y by drawing an arrow from X to Y, figure 4.9 shows the differences between a single CA and a two level hierarchy.

Such a two level PKI could be used in a large company: there is one CA per department to certify members of the department, and there is one company wide root CA certifying the departmental CA's.

It is interesting to look in more detail at the trust issues arising when going from a single CA to a two level hierarchy. If a single CA certifies all entities, the entities need to trust the CA for only certifying correct public key-owner bindings. Hence the CA will need good procedures for enrolling for a certificate: it should not be possible to fool the CA into signing an incorrect binding. If we move to the two-level hierarchy, the amount of trust needed in the root CA is much larger: all entities need to trust the root CA for:

- never certifying an incorrect public key - owner CA binding: this is similar to the one CA case.
- never certifying an entity that cannot be trusted to be a good CA

This second item actually represents a great amount of trust: all entities must assume that the root CA can correctly evaluate whether another entity will be a good CA!

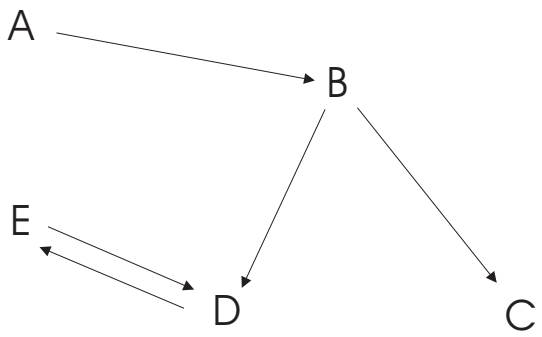

Figure 4.10: Network PKI

Hence, when using multiple levels of certification, it is reasonable to allow the root CA to limit the power of the CA's it is certifying. This can be achieved by putting *constraints* in the certificate. An example constraint could be: the CA certified by this certificate can only certify members of the finance department.

The idea of a two level hierarchy can easily be extended to an arbitrary tree of CA's. In this *hierarchical model*, there is still a single root CA, that certifies a number of second-level CA's. The second-level CA's can certify third-level CA's and so on, leading to a hierarchical tree of CA's. The leaf CA's in this tree will certify public keys of ordinary entities. In a hierarchical PKI, it is even more important to include appropriate constraints in certificates: if a CA certifies an end-entity, it should not be possible that this entity decides to become a CA himself, and starts certifying other entities (thus deepening the tree). Hence certificates will include constraints saying things like: this is a certificate of an end-entity, or this is a certificate of an intermediate CA.

The advantage of this hierarchical model is its relative simplicity, the main disadvantage is that the root CA represents a single point of failure: if the root CA private key is compromised, the entire PKI collapses.

Network trust model

Another way to deal with the scalability of the single CA solution is to abandon the idea of a CA and just allow entities to certify other entities. This leads to a certification structure as shown in figure 4.10.

In the example of the figure, if E already has A's public key, and wants to get C's public key, C can send to E his key certified by B, and B's key certified by A. E can verify both certificates to gain trust in the correctness of C's public key. If A and B have issued their certificates correctly, E will

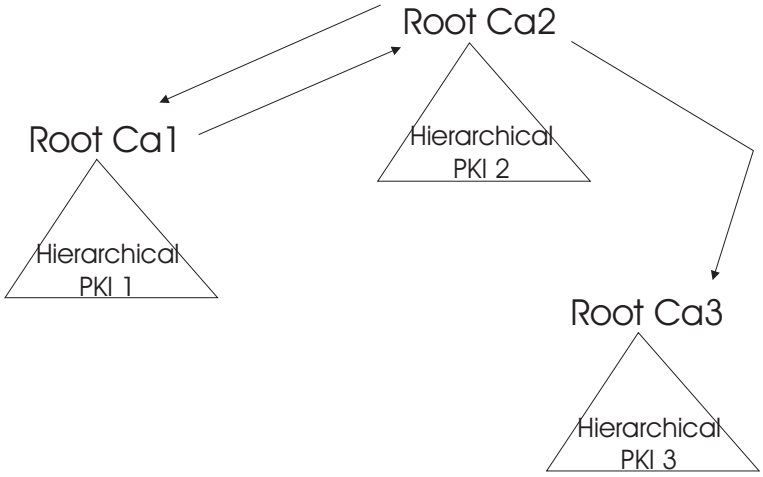

Figure 4.11: Hybrid PKI

indeed get the correct public key for C.

It should be clear that the trust issues in this network model are quite complex. In the example above, E needs to trust both A and B to correctly certify other entities. But the trust in A is higher than the trust in B, because A is higher in the certification chain.

The network model in its purest form is used in PGP. We refer the reader to [29] for more information.

Hybrid model

The network and hierarchical trust models can be combined as shown in figure 4.11. There are a number of separate hierarchical PKI's, each with their own root CA. These root CA's now certify each other as in the network model. These kinds of certifications are called *cross certifications*. They can be one way or mutual.

4.6.3 Revocation

A second problem with the simple PKI of section 4.6.1 is that it cannot handle the situation where someones private key is compromised: if the CA issues a certificate binding a public key to A, and A's private key is stolen subsequently, the attacker that stole the key can impersonate A forever.

To solve this issue, certificates typically carry a validity period: they expire after a certain time. If we keep the validity period very short, the compromise of a private key is not so bad anymore: the window of vul-

nerability after key compromise is very short. However, very short validity periods have as a side-effect that certificates must be renewed very often.

Hence, the validity period is typically rather long, and an additional mechanism is introduced to deal with the case of compromised keys: a certificate can be *revoked*. The two most widely used ways to implement revocation are:

- *Certificate Revocation Lists (CRL's)*: A CRL is a list published regularly by the CA that contains (the serial numbers of) all certificates that are still valid according to their validity period, but that have been revoked. The entire list is signed by the CA.

- the *Online Certificate Status Protocol (OCSP)*: during validation of a certificate, the CA (or some delegate) must be contacted to check whether the certificate is still valid.

To support revocation, certificates can carry an attribute indicating where revocation information about the certificate can be found, e.g. a URL where the most recent CRL can be downloaded. The CRL will not contain full certificates but serial numbers of certificates, so each CA should include a unique serial number in each certificate it issues.

The possibility of revocation considerably complicates the certificate validation process: where a CA signature can be checked offline, getting recent certificate status information requires the validator to be online, either to download a recent CRL, or to contact an OCSP responder.

4.6.4 Certificate validation

While the basic notion of a certificate and its validation are quite straightforward, the issues of the trust model and revocation complicate the matter. As we have seen above, a certificate will contain:

1. name of the owner
2. public key of the owner
3. serial number of the certificate
4. name of the issuer
5. a validity period
6. information on where revocation information can be found

7. various kinds of constraints on the usage of the (public key in) the certificate, e.g.

   - the certificate can not be used to certify other certificates
   - the certificate can be used to certify other certificates but only of entities within a constrained namespace
   - the key in the certificate can only be used for signature verification and not for encryption
   - ...

As a consequence, the process of validating a certificate also gets more complicated. Typically, one validates a *certificate chain*. The verifier has a secure copy of the public key of some root CA, and needs to validate the certificate of an entity, with the possibility of having an arbitrary number of intermediate CA's between the root and the end-entity. The verification of such a certificate chain includes the following steps:

1. Verify whether the end-entity certificate contains the expected name (e.g. if you surf to website www.xyz.com over SSL, the certificate that the site sends you should contain the name www.xyz.com).

2. Verify whether the current date falls within the validity period of the certificate.

3. Verify the signature over the certificate using the public key of the issuer (i.e. the public key contained in the next certificate in the chain that is to be verified).

4. Check whether the certificate has been revoked

5. Check all following certificates in the chain in a similar way, until you arrive at a certificate that was signed by a CA who's public key you knew beforehand.

6. Verify all constraints on the certification path: e.g. if there was a constraint in one of the certificates that the certificate could only be used to certify entities within a certain name-space, then make sure that the constraint is satisfied.

As you can see, validating a certificate is a complex and error-prone process.

4.6.5 Real-world PKI's

Most real-world PKI's are based on the X.509 standard for certificates. Version 3 of this standard includes fields for all the information we discussed in previous sections and more. The X.509 standard has been criticised extensively for being too vague: it leaves many issues unresolved, leading to serious interoperability problems between X.509 based products. An in-depth discussion of the pitfalls in the X.509 standard can be found in [28].

An X.509 profile is a refinement of the standard, making choices for some of the open issues in the standard. A profile that may gain importance in the future is PKIX, the PKI for the Internet. The PKIX standards specify a detailed X.509 profile as well as the corresponding validation algorithm. More information on PKIX can be found on [**?**].

An important issue with setting up a PKI that we have not discussed here are the logistic and administrative aspects. This includes setting up and maintaining a certification authority, and defining secure procedures to enroll for a certificate, to renew a certificate or to revoke a certificate. Since the CA is such an essential link in the security chain, it must be well secured. This means physical security of the machine running the CA, as well as secure procedures for personnel operating the CA. In practice, the process of verifying the identity of entities applying for a certificate is often separated from the process of creating the certificates. The identity verification is then done by a so-called *Registration Authority (RA)*, and the actual signing of the certificates is then done (possibly in batch) by the CA. In that way, the CA can be offline most of the time, and only needs to communicate now and then with the RA.

4.6.6 Conclusion

While asymmetric cryptography promised to make the key establishment process simpler, the truth is that even a public-key based solution for key establishment tends to become very complex when scaling it up to a large number of users. Setting up and maintaining a PKI is an expensive endeavour, and many PKI projects have failed because their complexity was underestimated.

4.7 Advanced protocols

We have discussed protocols for entity authentication, key establishment and single sign-on. While these are probably the most widely used kinds of protocols, there are many other kinds. Some examples:

- protocols for *digital payment or digital cash*
- protocols for *electronic voting*
- protocols for computing with secret data, *secure distributed computing*
- ...

For a good overview of the kind of security goals that can be realized with cryptographic protocols, we refer the reader to the excellent book of Bruce Schneier ([11]).

4.8 Software interfaces to cryptographic protocols

4.8.1 Introduction

A software interface to a cryptographic protocol will of course often depend on the kind of protocol that one is interfacing to. Hence, in this section, we will discuss a number of quite different API's. We will discuss two example secure communication API's in more detail. Finally, we will discuss an interface to single sign on protocols: the authentication part of the JAAS API.

4.8.2 Java Secure Socket Extensions

Description

The *Java Secure Socket Extensions (JSSE)* API is a provider based API that offers authenticated confidential and integrity checked communication over sockets. Providers for the API are typically based on SSL/TLS like protocols. Although it is in principle possible to put a protocol like Kerberos under JSSE, such a provider is not yet available.

The basic idea of JSSE is that it offers secure sockets as decorations of normal sockets. The API is structured as shown in figure 4.12: an application uses the API for secure communication and the API internally uses an existing unsecure communication mechanism.

Example code

TODO

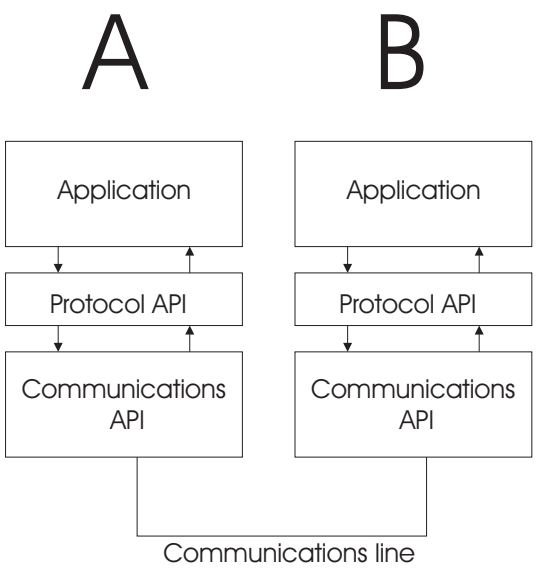

Figure 4.12: JSSE API

4.8.3 Java GSS API

Description

The *Generic Security Service (GSS)* API is an old C API to mechanisms for entity and message authentication and message confidentiality. The Java GSS API is an object-oriented rewording of the GSS API in Java.

The Java GSS API is a provider based API under which different security mechanisms can be plugged. The GSS API is a *token-oriented* API: it provides methods for constructing cryptographic tokens, i.e. opaque messages, and the application is responsible for getting these tokens to the other side. Hence, in contrast with JSSE, the JGSS API is not a wrapper around a communication system. It is structured as shown in figure 4.13.

Example code

TODO

4.8.4 Pluggable Authentication Modules

TODO

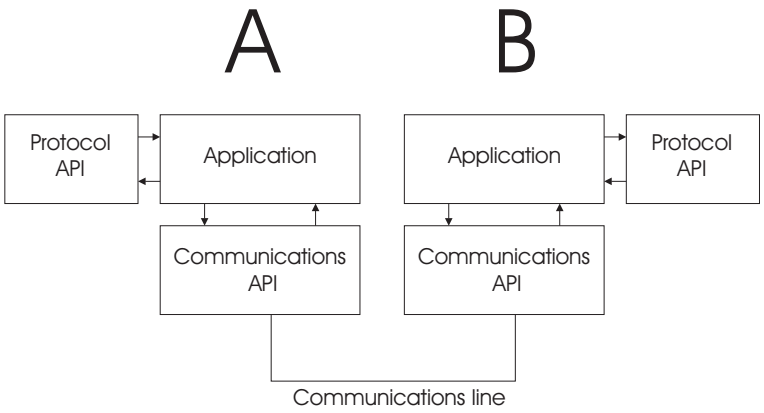

Figure 4.13: JGSS API

4.8.5 JAAS API, Authentication part

Description

The *Java Authentication and Authorization Services (JAAS)* consist of an authentication part, that is essentially an interface to single sign-on or authentication protocols, and an authorization part that performs access control based on the authenticated identity. In this section, we focus on the authorization part, and we come back to the authentication part later.

Example code

TODO

4.9 Further Reading

TODO

Chapter 5

Access Control Fundamentals

Cryptographic methods are about protecting access to data in the case where an attacker has access to the raw bits representing the data. For example, if you want to protect the confidentiality of data while sending it over a channel that is insecure (i.e. an attacker can read all bits travelling over the channel), you will resort to cryptographic algorithms.

In computer security, it is often the case that certain data is not directly accessible to an attacker: the attacker can only get access to the data via a certain software layer. The most typical example of such a software layer is an operating system. If an attacker is hacking his way into a remote computer, he does not have direct access to the bits of the files on this remote computer: there is a software layer (the operating system) between the attacker and the data. Other examples of such software layers that protect access to data are: web servers, databases, and application servers.

Access control techniques are about how to build such a software layer. This turns out to be much harder than it would appear at first sight.

First, one has to decide precisely what will be allowed by the software layer and what not. These rules about what is allowed and not are written down in a more or less formal *security policy*. Writing down a consistent and unambiguous security policy from scratch is hard, but a number of *security policy models* that are useful in many situations have been developed and studied in computer security research. These models typically focus on access control, and are therefor often referred to as *access control models*.

Secondly, once the security policy is made precise, it has to be implemented correctly. Any bug in the implementation can lead to a potential security breach. Moreover, many bugs that would typically not show up in non-security related software will become very important in access control software, because attackers will do the most unlikely things to get such a piece of software to break. Robustness and correctness are very important

for an access control layer to be secure.

In this chapter we will look at the first aspect: we study some example security policy models for access control. We will focus on policy models for applications that let multiple users share data in a controlled way. These cover a very wide class of applications, including operating systems, databases, web servers and web applications. For each of the policy models, we discuss both the model and possible implementation structures.

In more complicated scenarios (especially scenarios involving applications built out of partially trusted software components) more expressive policy models are necessary. Access control in this setting is discussed in one of the next chapters.

The second aspect, implementation correctness, will be covered in chapter 7 where we will see some detailed examples of how bugs in an implementation can lead to security breaches. By discussing a taxonomy of common implementation errors, we hope to make the reader aware of these errors, and help him to avoid these in his own software. An implementation error that can potentially lead to a security policy breach is often called a *software vulnerability*.

This chapter finishes with an overview of the Windows 2000 access control system.

5.1 System model for access control in multi-user software

We start by defining a system model: an abstract model for the system that needs to be secured. Then we go on to describing different ways of defining a security policy for such a system.

A system that needs to be secured manages *objects* that contain data or represent resources. Examples of objects could be files, or network sockets, or semaphores. An object could also be an application-level object, e.g. a shopping cart in an e-commerce web application. An object is accessed by means of a predefined set of *operations*. What operations are available may depend on the type of object under consideration. A file for example might support read and write operations, whereas a semaphore might support P and V operations.

Users will, in the course of performing useful work on the computer system, try to access objects: reading files, adding items to shopping carts, etc. But the user, as a human being, cannot directly invoke the operation on an object: he will have to log on to a computer system, and start a process or

thread that acts on his behalf. We call the active entity (process, thread) that performs the operations on behalf of the user a *subject* or a *session*. A user can have many different subjects (sessions) running on his behalf, even at the same time. It is possible in our system model that subjects or users are themselves objects too. For example a process could be considered as an object with operations for suspending or killing the process. Or a user could have an operation that changes his password.

In typical operating systems, it will often be the case that all processes and threads actually run with the privileges of the user on whose behalf they are running. Under this simplification, it is not useful to distinguish subjects and users: anything that happens on the computer system happens on behalf of some user. We don't care what process or thread is doing the work: all processes or threads running on behalf of the same user are considered equivalent from the point of view of access control.

The notion of subject or session only becomes important when a user wants to voluntarily limit his access rights, perhaps to protect himself against mistakes, or perhaps because he wants to run programs that he does not fully trust. By logging in with less access rights, he can limit the damage that could be done.

We assume that users are properly authenticated when starting a session (i.e. logging on as a certain subject), and that the user can indicate in some way how much access rights he wants to grant the newly created session. How this is indicated will depend on the security policy model and on implementation details.

In many cases a user will start many sessions with the same access rights. A collection of all sessions with the same rights is called a *protection domain*.

The level of access that a subject (or protection domain) has is often expressed by stating the *permissions* for the subject. A permission is a statement that something is allowed. In its simplest form, a permission can encapsulate the right to access a specific operation on an object. But permissions can be more generic: e.g. the permission to perform all operations on a given object, or the permission to perform the read-operation on an entire collection of objects.

The *access control policy* essentially assigns a specific set of permissions to each protection domain.

A final concept that must be discussed is the notion of a *reference monitor*. The reference monitor is the part of the software and/or hardware that actually enforces the security policy. It is invoked on every object access, and denies or allows this access depending on the current access control policy. In operating systems, the reference monitor is a part of the kernel.

This system model is summarized in figure 5.1.

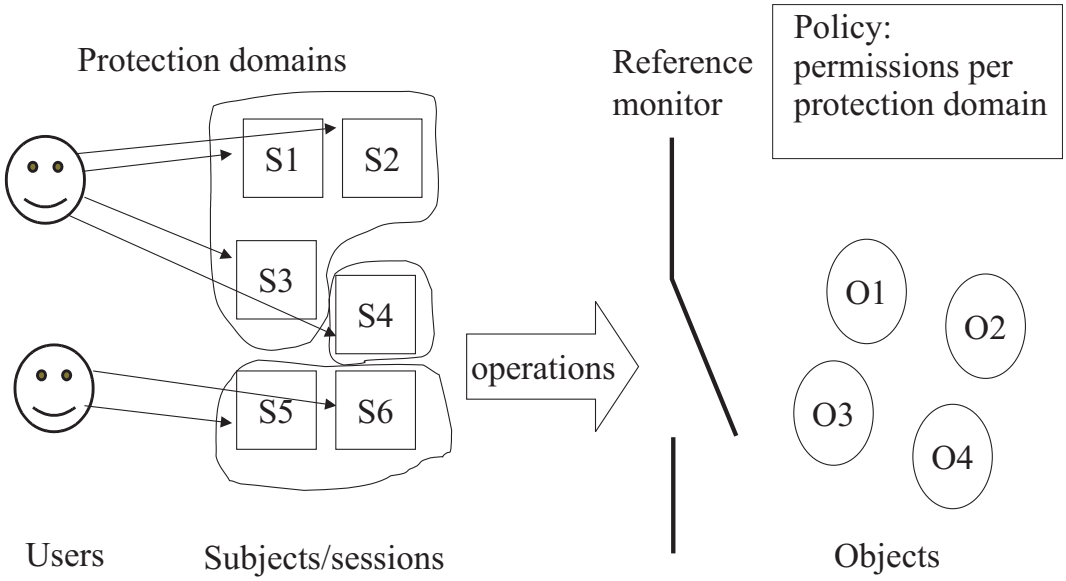

Figure 5.1: Access control system model

The security policy at a given point in time can be conceptually represented by an *access control matrix (ACM)*. The rows of the ACM are indexed by the subjects and the columns by the objects. Each entry in the ACM (say indexed by subject s and object o) contains the operations that s is allowed to perform on o.

5.2 Classic discretionary access control

5.2.1 Objective and model

The most well-known access control model is the model that is implemented by most popular operating systems. Its objective is to enforce creator-controlled sharing of objects.

In this model, every object in the system has a user that is the *owner* of the object. When a new user is created (registered with the system), it owns nothing (or only some newly created objects such as a home directory). When a new object is created, the owner is the creating user. Every user can autonomously decide what access to its objects it gives to other users. In other words, who gets access is at the discretion of the owner. Hence the name *discretionary* access control (DAC).

There exist different variants of DAC: some variants will, for instance, allow users to pass on the ownership to another user while others will not

allow this. Some variants will allow a user to delegate his rights to grant access while others will not allow this.

The advantage of the discretionary access control model is that it is conceptually simple and intuitive. That is also the reason why it is so widely implemented.

But this model also has a number of important disadvantages.

In the first place, discretionary access control is difficult to administer. The fact that users own objects makes it difficult to deal with the situation of a user being deleted. An example would be an employee leaving a company. The files owned by the employee have to be transferred in some way to one of his colleagues or to a new employee. Also, all access rights that others have granted to the deleted user must be removed. While possible, this is tedious and error-prone work. Also, when a new user is added to the system, the default of having no access to existing objects is seldom appropriate. So time must be spent thinking about what access the new user should have, and the owners of the respective objects have to give the new user appropriate rights.

Some of these problems can be solved by giving the set of users more structure. A typical example is to allow grouping of users. Access rights can then be given to a group instead of to an individual user. When adding a new user, appropriate rights can be granted to the user by adding him to the correct groups. When deleting a user, his rights can be revoked by removing him from the groups he belongs to. However, as long as it is possible to grant rights to individual users, the administration of a discretionary access control system remains a challenge. This problem will be handled better by role-based access control systems.

As a second disadvantage, discretionary access control is not very secure for at least two reasons.

- In the first place, the fact that users can decide for themselves about access rights leaves them vulnerable to so-called *social engineering* attacks, where a malicious user convinces another user to grant him unnecessary rights. (E.g. "I cannot solve your problem unless you give me full access to all your files")

- In the second place, discretionary access control is very insecure when users run malicious programs. Since a program run by a user can do everything the user can do, a malicious program could give access rights to attackers without the user being aware of it.

A malicious program that does something unexpected without the user noticing it is called a *Trojan Horse*. Discretionary access control does not handle

Trojan horses well. A Trojan Horse, since it runs with the access rights of the user, can grant access to anybody.

There are at least two ways to deal with the Trojan horse problem:

- We no longer give users (and hence the programs they start) the right to change access rights. This leads to mandatory access control, discussed in the next section.
- We no longer consider all programs equal: a program that is not trusted by a user is run in a restricted environment. This leads to the more complex access control model for untrusted code discussed in chapter 6.

  A simple implementation of this idea is that a user always has two sessions open: one in which all his access rights are enabled, and one with a minimal set of access rights enabled. Untrusted programs can then be executed in this second session.

5.2.2 Implementation structures

TODO
ACL's, Capabilities

5.3 Mandatory access control

5.3.1 Objective and model

The main objective concerning data security for the military, is strictly controlling information flow. The problem of Trojan horses circumventing intended access control was considered such an important problem by the military that they designed another access control model that (at least partially) solves the problem. This model has as its prime purpose the strict control of flow of information, even in the presence of malicious programs.

In their *mandatory access control* model, there is no notion of the "owner" of an object. Instead, there is some system-wide rule that states what kind of accesses are allowed and what not, and users (subjects) just have to abide by these rules. The most well-known example of a mandatory access control model is the lattice-based access control (LBAC) model. In this model all objects are labelled with a sensitivity level (e.g. Top Secret, Secret, Confidential or Unclassified), and all users are cleared to a specific sensitivity level. When a user logs on and starts a subject, he has to indicate a sensitivity level for the new subject. A user can start subjects with any level lower than or equal to his clearance level.

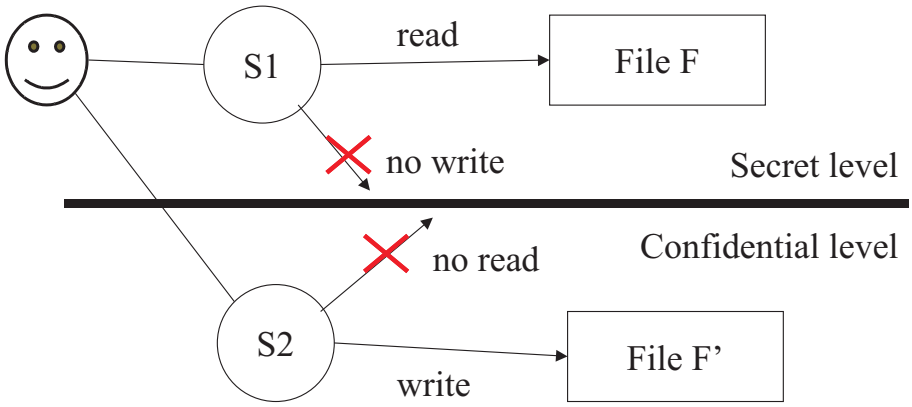

Figure 5.2: LBAC prevents incorrect information flow

The system wide access control rule has two parts:

no read up A subject labeled with level l_s can only read from objects with a sensitivity of level l_s or below. For instance, a subject cleared for confidential documents can never read top secret files.

no write down A subject labeled with level l_s can only write to objects with a sensitivity of level l_s or higher. For instance, a subject cleared for the secret level can never write information to confidential files.

Together, these two rules prevent Trojan horse programs from leaking information to a lower sensitivity level. This is illustrated in figure 5.2. If a user cleared for Secret data logs on to a computer, he must indicate a sensitivity level for the subjects he starts. If he starts a subject S1 with level Secret, the subject will be able to read a Secret file F, but it will not be able to write the contents of F to a file at the Confidential level, not even if the programs run by the subject contain Trojan horses: the no-write-down rule is always enforced. If the user starts a subject S2 with level Confidential, he will be able to write to Confidential files, but he will not be able to read the Secret file F: the no-read-up rule is always enforced.

The general LBAC model allows an arbitrary lattice of sensitivity levels. Two example lattices are shown in figure 5.3. The left lattice is the linear order Top secret > Secret > Confidential > Unclassified. The right lattice shows how Confidential data can be further horizontally subdivided according to, for instance, the project that the data belongs to.

While very important in military applications, the LBAC model did not capture very well the access control requirements of commercial businesses, and hence, it is not very widely implemented outside the military. Moreover,

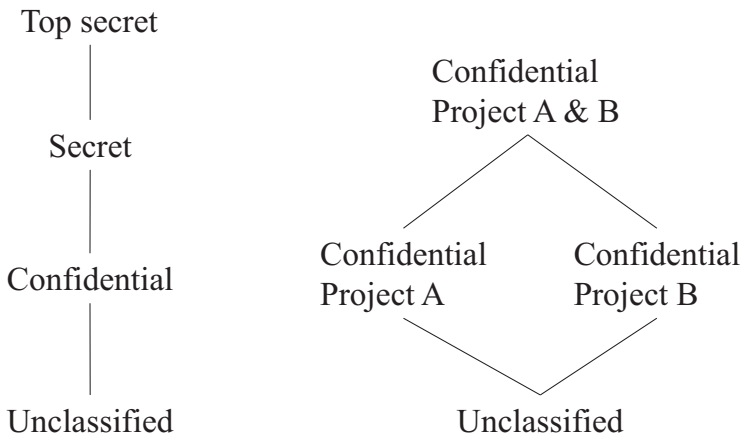

Figure 5.3: Some lattices of sensitivity levels

practical implementations of the LBAC system have suffered from various problems:

- It turned out to be unrealistic to enforce the two rules consistently. In practice, trusted programs had to be introduced that could downgrade information (i.e. that can violate the "no write down" rule).
- Even where the rules were imposed, some level of information leak was still possible through so-called *covert channels*: a Trojan horse at the secret level could send information to another program at the confidential level by modulating some system parameter (like cpu-utilization).

5.3.2 Implementation structures

TODO
labels, security kernel

5.4 Role based access control

5.4.1 Core Role based access control

Role based access control (RBAC) is essentially an attempt to better capture typical business access control requirements, and this with less administration overhead than the discretionary access control model.

In RBAC, the central concept is the notion of a *role*. A role represents a certain function, job or responsability (e.g. the role of clerk at a bank).

A role has a number of access rights or permissions associated with it: the rights necessary to perform the function represented by the role. Users are then assigned to one or more roles. When a user logs on and starts a session, he indicates which of his assigned roles he wants to activate and attach to the session. A session has the right to perform a given operation if one of its roles has the corresponding permission.

RBAC is a kind of mandatory access control in the sense that there is no notion of "owner" of an object, and hence access control decisions are never left to the discretion of individual users. The system-wide access control rule is the rule stated above that a session has access when one of its activated roles has access.

With RBAC, adding or deleting users become quite straightforward operations. Hence, the administrative overhead disadvantage of the discretionary access control model goes away. Moreover, since RBAC is mandatory, it suffers less from social engineering attacks.

RBAC is becoming more and more popular in operating systems and databases, and is the most important access control model for application level access control (for instance in web-based applications).

The core RBAC model has been extended in numerous ways, to make it easier to manage, or to make it more expressive. We discuss two popular extensions below.

5.4.2 Hierarchical Roles

Roles represent responsabilities, jobs or functions in an organisation, and often the various existing roles are related to eachother. A role *hierarchy* makes it possible to explicitate some of these relations between roles, and as such also makes administration easier. A role hierarchy defines a partial order between roles, called the *seniority* relation. A senior role inherits all the permissions from its junior roles, and a junior role inherits all the users from its senior roles. An example role hierarchy is given in figure 5.4. The Engineer role groups all the permissions given to engineers. The Project A Engineer role inherits all permissions from the Engineer role and adds some permissions specific to the engineers working on Project A. The Director of Engineering role inherits all the permissions from all engineering roles, and adds some permissions specific to the director role. A user added to the Project A Engineer role is automatically also added to the Engineer role. In summary, permissions are inherited upward (highest role has most permissions) and users are inherited downward (lowest role has most users).

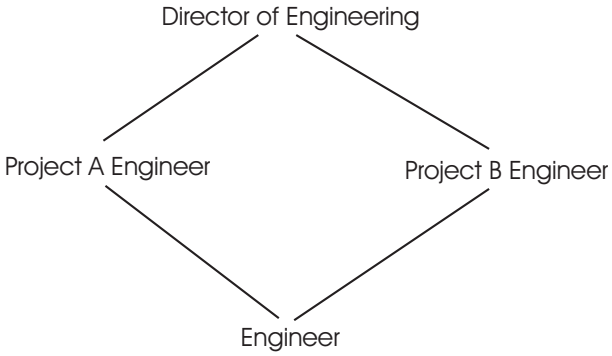

Figure 5.4: An example role hierarchy

5.4.3 Constraints

Another extension of RBAC allows for the specification of constraints on the assignment of users to roles, or on the activation of roles.

Static constraints are constraints on the assignment of users to roles. For instance, a constraint could say that a Judge role and a Jury role are mutually disjoint: it is not allowed for anybody to be both Judge and Jury. The addition of static constraints to RBAC makes the formalism more expressive: for instance, security policies can now enforce the *separation of duty* principle: assign permissions to users only in such a way that no single user can commit fraud without being caught. Separation of duty is an important security principle: experience shows that the risks of fraud are substantially reduced if fraud requires collaboration among different users.

Dynamic constraints are constraints on the activation of roles. For instance if a user has both the Project A Engineer role and the Project B Engineer role, a dynamic constraint could say that the user is only allowed to activate one of these roles at a time. Dynamic constraints help to enforce the *principle of least privilege*: make sure that you do not have more permissions than you need for the job you are currently doing.

5.5 Other access control models

DAC, MAC and RBAC are certainly not the only access control models. In the literature, a wide variety of other models is described.

The *Biba model* is a lattice-based access control model designed to protect integrity instead of confidentiality. Objects and subjects are labeled with "integrity" levels, and information is only allowed to flow down the lattice of integrity levels.

The *Chinese wall* model is a dynamic access control model that reduces the permissions of users in the course of time. Such a model is important for instance in consultancy firms, where a consultant, after seeing data from client A, may be forbidden to see data about client B (a possible competitor of client A).

For an overview of existing access control models, the reader is referred to [].

5.6 Example: access control in Windows 2000

TODO

SID's, ACL's, ACL inheritance, simulating RBAC in Windows 2000 using groups.

5.7 Further Reading

TODO

Chapter 6

Untrusted Code Security

In the system model of chapter 5, it was assumed that any process or thread running on behalf of a user, should have all access rights granted to that user. This assumption makes sense if the user can be sure that all programs that he starts will behave as he expects them to behave. If the programs just perform the actions that the user wants them to perform, programs can be considered to be an "extension" of the user, and there is no need to model them separately in the system model.

The assumption that the user could trust all software on his computer to behave as he expects might have been reasonable some decades ago, when installing new software on a computer was a relatively rare event, and only software from trusted sources was ever installed. But today, new software can be installed on a PC just by clicking a link in a web browser. It is clear that in this new context, it is no longer reasonable to trust all software on your computer. Moreover, many applications can be extended at run-time with new pieces of code: a browser for instance can be extended with plugins, applets, ActiveX controls and so forth.

Hence, there is a need for access control technologies that take into account the fact that users might trust certain (parts of) programs more than others, and hence might wish to grant certain (parts of) programs all their rights, while granting others only a very limited subset.

One possible technology (and a very important one) is called *sandboxing*: untrusted code is put in a (configurable) sandbox, where it can do no harm. Sandboxing was first implemented in Java virtual machines, to support the secure execution of *mobile code*, in particular of applets. Such downloaded components are prototypical examples of components that would be less trusted, and hence sandboxing is often referred to as a mobile code security technology, even though it can be just as useful in applications that are statically built from several components where some of these components

are more trusted than others.

In this chapter, the technique of sandboxing is discussed in detail. It is shown how the combination of a type-safe language, stack inspection and a suitable class library can realize configurable sandboxing of untrusted components securely, with relatively little effort for the developer. Also, the implementations of these ideas in Java and .NET are discussed.

The chapter is structured as follows: section 6.1 and 6.2 motivate the need for a technology for securely loading partially trusted code. Code mobility is used as a motivating example. Current applications are often built as extensible frameworks that can accept code extensions from anywhere on the network, and current operating systems do not provide adequate support for securing against malicious extensions. In section 6.3, we define the notion of component more precisely, and show that limiting rights of components is useful in other scenarios than code mobility alone.

The remainder of the chapter focuses on the technology of *sandboxing* to support the secure loading of partially trusted extensions. An important building block of this technology is *type-safety* and this is discussed in section 6.4. Next, in sections 6.5 and 6.6, we describe how a security policy for partially trusted code can be expressed using the concept of *permissions*, and how this policy is used to tag components with their permissions at load-time. Section 6.7 discusses stack inspection, another key ingredient of sandboxing. In section 6.8it is shown how a sandboxing based security architecture can be made extensible. Sections 6.9 and 6.10 discuss briefly how sandboxing is implemented in the Java platform and in the .NET platform.

6.1 Mobile or partially trusted code

The term *mobile code* can be used for any kind of executable content that is transmitted over a network. As such, it is a very broad concept: PostScript for instance can be considered executable content, since it is a programming language designed to describe the contents of a page. In a similar way, macros in a document or spreadsheet are executable content and hence could be considered mobile code when sent over a network. More obvious examples include applications or device drivers that are downloaded from the Internet, or applications that travel as attachments to an e-mail.

Code mobility is by no means new: since the early days of the Internet, the network has been used to transport executable content. But an important and relatively recent trend is to build applications that themselves are extensible by code that is downloaded. The application is built to accept extensions from a variety of sources. Some examples:

- a multimedia player can be extended at run-time with new codecs, so that it can decode and play new kinds of audio or video files.
- a web-browser can be extended with plug-ins to deal with specific kinds of content, e.g. a viewer for a specific document format.
- the same web browser can be extended with new code automatically if the user visits a page that contains executable content. This executable content can be interpreted by the browser (e.g. JavaScript), or it can be native code that is actually linked to the browser code (e.g. ActiveX controls).

  Applets lie somewhere inbetween: a browser implemented in Java can load and link them to the browser code, but many browsers implement a well-isolated virtual machine that could be considered an interpreter for the applet.
- also web servers are extensible: the server can be extended at run-time with new functionality to render specific dynamic web pages. Examples of technologies that do this are Java Server Pages and ASP.NET, the .NET variant of the older Active Server Pages (ASP) technology.
- many applications today support automatic updating over the Internet: bug-fixes or product updates can be downloaded and applied with little or no user intervention.

What interests us in this paper is not so much the fact that mobile code is used, but the fact that depending on the source the code came from, the user might want to enforce some security restrictions on what the code can do. In other words, the user might not fully trust the downloaded code: what is needed is a technology to support constrained execution of partially trusted code.

6.2 Security aspects

Securing code mobility can be considered from a number of different viewpoints. On the one hand, the code producer might want the code to be protected, e.g. to be sure that no unlicensed copies can be made. On the other hand, the code consumer might want some guarantees that the downloaded code cannot do any harm on his computer system. In this paper we will only focus on this second concern.

There are two reasons why standard operating systems like Unix or Windows do not provide adequate support for this:

1. An operating system assigns privileges to a process based on the user that started the process. What program is started is typically not taken into account. So for instance if a user clicks on an executable attachment to an e-mail, the attachment will be run with the privileges of the user. This is one of the main causes why e-mail borne viruses have been able to propagate so easily.

   Both Windows and Unix have some mechanisms to side-step the user-based privilege assignment. You can start for instance a process as another, more restricted user. Or you can start a thread in a process with more limited permissions than the process itself. Using these mechanisms, some form of protection against partially trusted executables would be possible.

2. However, another important restriction in current operating systems is that the operating system considers all code in the memory space of a process to be equally trustworthy. As a consequence, changing the privilege level of a thread depending on the code that it is currently executing would be hard to realize. But of course, if an application consists of different parts that differ in trustworthiness, this adaptation of the privilege level of a thread is exactly what one would want to do.

The virtual machines for modern programming languages provide solutions for these two issues. We will discuss a way in which an application platform can support applications consisting of many components where not all components are equally trusted. Moreover, secure run-time extensions of applications with new components, where the extensions are only partially trusted will also be possible. The approach we discuss relies both on properties of the programming language used to develop the application, and on the runtime-system used to execute it. As such, it is not something that can be retrofitted on legacy applications. But the two most important new application development platforms, the Java platform and the .NET platform, both support this approach.

6.3 Applications and Components

If one wants to grant different rights to different pieces of code, an important choice is the level of granularity with which one can distinguish pieces of code. One extreme is to take complete applications as the unit: different applications can be given different access rights, but within one application, all code has the same rights. This is certainly useful and is supported in a limited way in current operating systems as discussed above in section 6.2.

But software applications are not monolithic pieces of code. A single application is usually built from a collection of components, and it is often not the case that these components all come from the same software producer. Applications that accept plugin extensions at run-time with new, downloaded code are just one, very extreme, example. Even applications that are built statically from a number of components with different origins can benefit from a technology that can limit the amount of damage that one of these components can do. As a consequence, if one wants to limit what code can do, applications are not the most interesting unit of code to consider.

We will use the term *component* when we talk about a piece of software (code and possibly resource files such as graphics) that is:

- a *unit of deployment*: components are not deployed partially.
- third-party composable: components can be composed to form applications by anybody, not only the developer of the component.

Examples of components are DLL's (Dynamic Link Libraries), Java class files, and .NET assemblies. For the rest of this chapter, we will consider applications to consist of a collection of such components. Different components can be given different access rights, but within one component, all code has the same access rights.

6.4 Safety and type soundness

6.4.1 Safe programming languages

Many program fragments only make sense if certain conditions hold in the current execution state. Consider for instance the following statement:

```
a[i] = (int) d;
```

The effect of this statement is only well-defined under a number of conditions: the variable a should be bound to an array, i should be an integer and should lie between the bounds of the array, d should be bound to a value that can be cast to an integer (say a double), and the cast should not result in an overflow.

When defining the semantics of a programming language, one can take two different options:

1. One can force an implementor of the language to check all the conditions. The statement will only be executed if it makes sense, and some

well-defined error-reporting (e.g. terminating the program or throwing an exception) will be done otherwise.

This option makes it harder to implement a language efficiently: while a compiler can check some of the conditions statically, other conditions will have to be checked at run-time, leading to a less efficient execution.

2. One can put the burden of ensuring that the conditions are satisfied on the programmer. The definition of the language can then say that the behaviour of the program is *undefined* in such erroneous cases, and the programmer should make sure his programs never run into such undefined behaviour.

   This option allows for a more efficient implementation with less run-time checking, but a bug in the program (i.e. if the programmer fails to ensure all conditions are satisfied) can lead to completely unpredictable behaviour.

We call a programming language *safe* if the first option is chosen. For a safe programming language, the behaviour of any program is specified in the definition of the language. Most declarative languages (such as ML or pure Prolog) are safe.

If a language is unsafe, a specific implementation of the language could still choose to detect all cases where a program runs into undefined behaviour, and terminate the program when this happens. However, such a safe implementation will typically be much less efficient. Hence, this text will assume that unsafe languages are also implemented in an unsafe way.

The programming languages C and C++ are examples of unsafe languages. The ANSI C standard specifies in many places that the behaviour of C is undefined. Examples include the behaviour upon integer overflow, or the behaviour upon dereferencing an invalid pointer.

Modern object-oriented languages sit somewhere in between: languages like Java and C# have large subsets that are (at least intended to be) safe. But both Java and C# also include features that break safety. These features are present at least in part for efficiency reasons. In Java, an example of such a feature is the Java Native Interface, that allows Java programs to call arbitrary assembly code. In C#, there is even a language keyword **unsafe** that can be used to annotate a block. Such an unsafe block is allowed to perform operations such as pointer arithmetic that can break safety.

Whether a programming language is safe or not has some important security implications.

In the first place, safety can guarantee the absence of certain kinds of errors that could lead to security vulnerabilities. In particular, a safe language

can guarantee *memory safety*: a program fragment can only write to valid memory locations explicitly made accessible to that fragment by the compiler. In particular, it cannot interfere with run-time data structures such as the call stack. This avoids bugs like buffer overflows.

A second reason why safety is interesting from a security point of view, is again a consequence of memory safety. Because of memory safety, the run-time environment can partition one single memory address space in a secure way and assign the different partitions to different programs. Such a subdivision is currently not supported in a Java virtual machine, but the .NET Common Language Runtime (i.e. the .NET virtual machine) supports such partitions and calls them "appdomains". If two appdomains run only safe code, the separation between the two appdomains is similar to the separation between two processes that is provided by the operating system.

A third reason why safety matters for security has to do with run-time extensions. The fact that any program fragment has well-defined behaviour allows one to draw conclusions about an arbitrary program fragment, because all program fragments will have to abide by the rules of the programming language. In other words, it is not possible to break out of the abstraction offered by the programming language from within a program fragment. This is particularly interesting in the case where an application consists of various components, and some of these components are trusted more than others. By structuring the trusted components well, one can limit what less trusted components can do. Understanding how this works will be easier after a discussion of types and type soundness, so we will return to this point later. Note that, when using safe languages for security in this way, it is important to ensure that the components are safe at load-time: if checking would only be done at compile-time, a malicious unsafe component could be hand-crafted, or could be made by modifying a component after compilation (e.g. by removing the run-time checks that the compiler has inserted to guarantee safety).

6.4.2 Types and type soundness

As indicated above, the compiler for a safe language may need to include run-time checks. Static analysis by the compiler can reduce the amount of run-time checking needed, thus leading to more efficient execution. Also, we discussed how it is important from the point of view of untrusted code security to be able to reason about arbitrary program fragments. For both these issues, a *type system* can help.

Types are essentially annotations of programs. They make assertions about invariant properties of the execution state of the program, such as

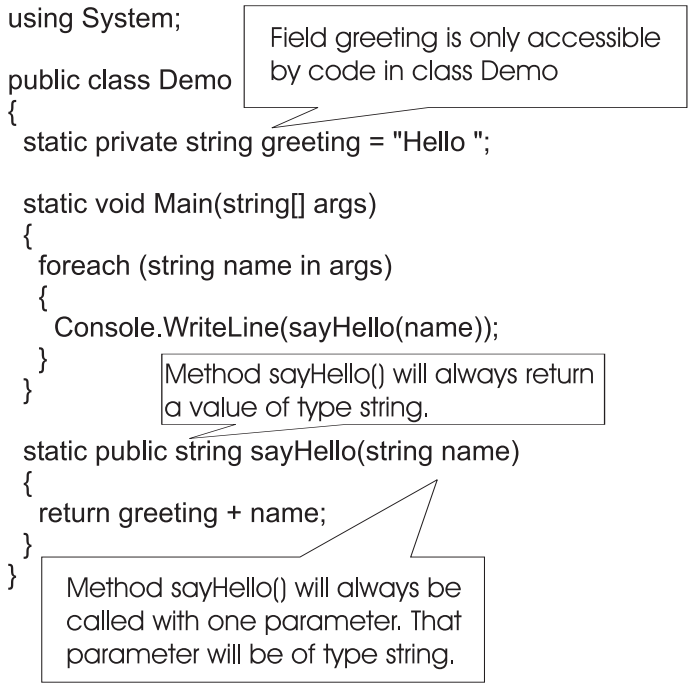

Figure 6.1: Typing information as assertions.

"This reference will always refer to an object of a class that implements interface I" or "The first formal parameter of function f will always be bound to an integer".

This is illustrated informally in figure 6.1. In the figure, a C# program is shown, informally annotated with some of the assertions made by the typing information in the program.

Typechecking is the process of verifying the assertions. This can be done at compile-time (static typechecking) or at run-time (dynamic typechecking). If the compiler and the run-time environment of the program use a combination of static and dynamic typechecking to *guarantee* that the invariants asserted by the types will hold, the language is called *type sound*.

Type soundness and safety are related in the following sense: an unsafe language can never be type sound, because if it runs into undefined behaviour, anything can happen and hence any assertion made by the type system could be broken. In other words, type soundness implies safety. Moreover, since types make static analysis of a program easier, typing information and type soundness can help reducing the amount of run-time checking that is needed to ensure safety.

C and C++ are examples of typed languages that are not type sound. In a language like C, types are informational: they document the intention of

the programmer, but they are by no means strong guarantees. Typechecking is "soft": it will detect many type errors, but a programmer can easily bypass the type system, intentionally (e.g. by using casts), or by accident (e.g. by accessing an array with an index that is out of bound).

Some declarative languages (e.g. ML) are safe and type sound: both the meaning of types and the meaning of programs is defined mathematically, and it can be shown by mathematical proof that a program that passes the typechecker will at run-time always satisfy the assertions made by the types. Other declarative languages (e.g. Prolog) are safe but do not use a type system.

The fragments of Java and C# that are safe are also believed to be type sound, but there is no mathematical proof. Moreover, using any of the unsafe features in these languages will definitely break type soundness.

Types can also help in reasoning about untrusted components. If the type system allows assertions like "This field is only accessible by code belonging to this class" (e.g. the private modifier in Java or C#), then the type system can be used to guarantee confidentiality of certain information with respect to untrusted code: if some trusted component stores information in for instance a static private field in one of its classes, then no newly loaded untrusted component can access this field directly.

Hence, a safe, type sound language (often called a *type-safe* language) is a convenient vehicle for addressing the untrusted code security problem. Note however that, when using type systems for security in this way, it is important to typecheck components at load-time.

Checking safety and type soundness at load-time is very hard for components distributed in native machine language. That is one of the main reasons (next to portability) why modern languages are typically compiled to an intermediate language. Java for instance is compiled to bytecode, C# to IL (Intermediate Language). These intermediate languages are also typed, and again, large subsets are believed to be type-safe. Both for Java bytecode, and for Microsoft IL, there are even proofs of type-safety, but such proofs are either for subsets of the languages, or under the assumption that certain parts of the standard class library are not called.

The process of verifying type-safety for the intermediate language (either at load-time or at JIT compile-time) is often just called *verification*. Intermediate code that verifies as type-safe can be loaded in the runtime environment with certain security guarantees: the code can, for instance, not access private fields of objects instantiated from other classes. Both the Java virtual machine and the Microsoft Common Language Runtime perform this kind of verification, and build an entire security architecture to protect against untrusted or partially trusted code on top of the basic security guarantees given

by type-safety. Essentially, such a security architecture tags every piece of code loaded into the runtime with information about what that piece of code is allowed to do. Type-safety ensures that a piece of code cannot change its tag. Then, all system code that accesses resources such as files, or the network will call the security architecture before giving access to the resource. The security architecture determines what code is active by looking at the control stack of the runtime, and decides on the basis of that information whether the access can proceed or not. In the following sections, we will describe each of these steps in more detail.

6.5 Security policy: what is code allowed to do?

6.5.1 Permissions

In a scenario where an application consists of components coming from different sources, we need a way to tell the runtime environment which components we trust and how much we trust them. The typical approach taken to express this policy information is to assign *permissions* to components.

A permission is a representation of the right to do a possible action on a specific resource. For instance, a permission could represent the right to do a read action on a particular file. The security policy assigns such permissions to components. A component will be allowed to read a particular file only if the security policy assigns that particular component the permission to read that file.

Permissions are typically represented as objects. Some examples are:

- FilePermission, possibly with a name parameter and a mode parameter. FilePermission(name, mode) represents the right to open the file with the given name, in the given mode (for instance read-only mode or read-write mode). The name parameter could be allowed to contain wildcards, so that a single permission object can represent the right to access many files.

- NetworkPermission, possibly with a host name and port number parameter, representing the right to do network communication to a given host on a given port.

- WindowPermission, representing the right to open windows on the user's screen.

Since a single permission object can represent many rights at once (see the FilePermission example with wildcards above), permissions can be given a set-like structure: permissions can have subpermissions. For instance FilePermission("f1","read") is a subpermission of FilePermission("*","read"), or in other words, the latter implies the former. Permissions will usually support an operation that checks if one permission implies another permission.

It is important to note that Permission objects are *not* capabilities: the fact that a piece of code owns a Permission object (i.e. has a reference to it) does *not* mean that that piece of code has the right represented by that permission object. The piece of code has that right only if the security policy (see later) assigns that permission to it.

6.5.2 Evidence and policy

Since there are a huge number of components and a huge number of permissions, the security policy is usually implemented as a configurable function that maps information about a component on the set of permissions assigned to that component. The information about a component that is used to compute appropriate permissions can take many forms. Typical examples of such information include:

- Was the component digitally signed by the publisher of the code, and if so, who was that publisher?

- Where did the component come from: was it downloaded from the Internet, or was it loaded from the local filesystem?

Evidence is the general term for information about components that is used to compute appropriate permissions (this terminology was introduced by .NET). So essentially, the security policy will be a configurable function that computes permissions based on evidence. In its simplest form, configuration could be done as follows. The administrator indicates in a table what kind of evidence leads to what set of permissions. For instance, code signed by code publisher X gets the set of permissions P. Or code downloaded from URL U gets the set of permissions P'. If a component has both pieces of evidence (i.e. is both signed by X and downloaded from U, it would get the union of the sets P and P').

In real life systems, the configuration mechanism for the security policy can be substantially more complicated. We will see an example of a more elaborate system in section 6.10.

6.6 Loading components

A prerequisite for enforcing access control on untrusted components is of course that the run-time environment knows what permissions are assigned to each component. When loading a new component (either at application start-up, or at run-time when the application loads a new component), the runtime will first collect all evidence about the component and verify the evidence if necessary. It will feed this evidence to a policy evaluator, that will return the permissions assigned to the loaded component. Depending on what the policy says, the runtime can decide to verify the component for type-safety, or to skip that verification. It should be clear that if a component is allowed to skip verification, no further security guarantees can be given. Only highly trusted components should be allowed to skip verification. The runtime will then tag the component with the resulting permissions in such a way that the component itself cannot change this tag. This is possible because of type-safety, for instance by putting this information in a private field of a system-provided class. From that moment on, the newly loaded component can become active. The run-time can for instance call the main() method on some loaded class. Or the code that initiated the loading of the new component can call a method on one of the loaded classes.

Remember from section 6.3 that a component is a unit of deployment. For Java, this unit is a class: the virtual machine can load individual classes on demand, and will assign permissions to classes. For the .NET platform, the unit of loading is an assembly. The .NET platform assigns permissions to assemblies: all classes in an assembly get the same permissions.

6.7 Stack inspection: what code is requesting access?

6.7.1 Basic architecture

One of the basic assumptions of the stack inspection technology is that resource access must always be done via some trusted (usually system-provided) component. For instance, to open a file, you must always call a method on the trusted, system-provided File class. There is no way for a partially trusted component to open a file "directly" without going through the File class. Because of this assumption, access control to the resource can be enforced in that trusted component. For instance, the open() method in the File class will implement the necessary access control checks. This access control check essentially checks if the current thread has the permission to open the

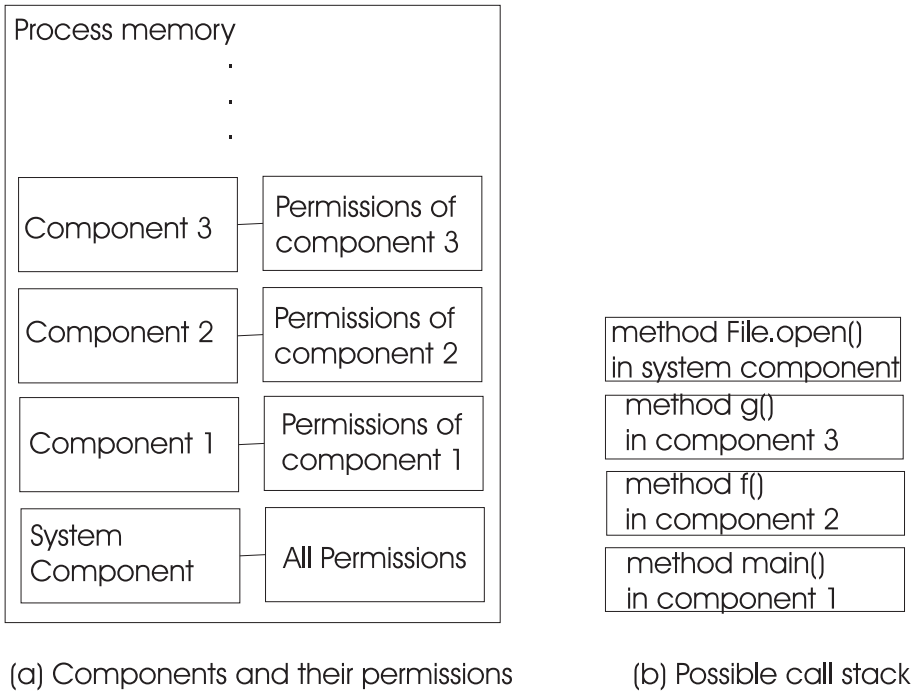

Figure 6.2: State of the run-time environment

file. Such access control checks will be done throughout the system class libraries (e.g. the Java class library or the Microsoft .NET Framework class library). Let us assume that the method that implements the access check is called demandPermission(P). Basically, demandPermission(P) will check if the currently executing thread of control has the permission P, and will return silently if it does, or throw an exception if it does not.

Hence, in the File.open() method, a demandPermission for a FilePermission will be done, and the open() method will abort with an exception if the current thread does not have the permission to open files.

How does demandPermission() make its access control decision? A possible state of the run-time environment is shown in figure 6.2. The runtime contains a collection of components, where each component can have a different permission set assigned to it. A thread that is executing in the runtime, can crosscut the various components: the thread can start for instance in some main() method of a class in component 1, then call a method f() of some class in component 2, which in turn calls the method g() in some class in component 3. If g() now calls into a protected resource (e.g. tries to open a file by calling File.open()), an access control check will be performed by calling demandPermission(). If the components 1, 2 and 3 have been assigned

different permissions at load-time, the question about what permissions the thread has is not so straightforward, and the algorithm to determine whether access is allowed or not is non-trivial. This access control algorithm is based on *stack inspection*: the call stack of the currently executing thread will be inspected to determine the current set of permissions. This call stack is also shown on figure 6.2. Note that, in the figure, the call stack grows upward. The runtime will associate with each activation record on the call stack the set of permissions assigned to the component that contains the method corresponding to the activation record. On the basis of all that information, demandPermission() decides if the current access is allowed or not. The details of the algorithm are intricate and are discussed in a separate section.

6.7.2 Details of the stack inspection algorithm

The implementation of demandPermission(P) will look at the call stack of the current thread (hence, stack inspection, also called *stack walking*) to determine if all calling code has been granted the demanded permission. The default behaviour of demandPermission is to inspect the entire call stack, and to check for each activation record that the code corresponding to that activation record has been granted the permission P. If one of the activation records on the stack fails that test, demandPermission fails (i.e. throws an exception) and access to the resource is not granted.

This default behaviour is secure: it is impossible for a component that has not been granted the permission to access files to access any file, even if the component tries to access a file via a number of "intermediary" more trusted components. This prevents so-called "luring attacks", where an untrusted componente "lures" a more trusted component into doing something bad.

However, in some circumstances, this default behaviour is too restrictive. For instance, a trusted component might wish to do something on behalf of a less trusted component (perhaps after some checking of the request) that the less trusted component is not allowed to do itself. A concrete example is the creating (and showing) of a window by an applet: applets might not be trusted to open arbitrary windows, because they might deceive a user that way (e.g. trick the user into typing a password in such a window). But some restricted kind of window-creation could be offered by a trusted component: the trusted component will open a window on behalf of the applet, but will make sure that this window is always labeled with a warning that this is an applet window. With the very restrictive stack walk algorithm that we have discussed upto now, it would be impossible to write such a trusted component: since the applet code is still on the stack when the trusted code opens the window, the default behaviour would be to deny opening the

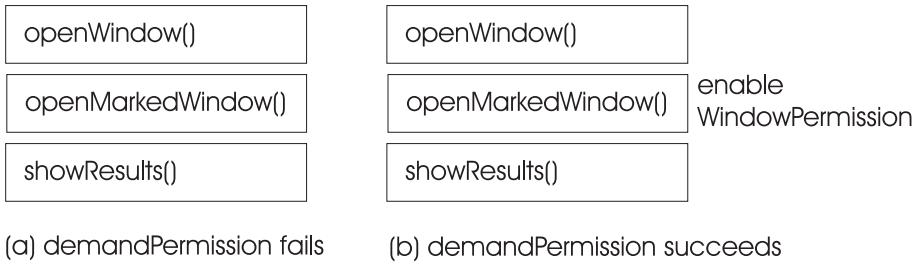

Figure 6.3: Illustration of the enable modifier

window, even to the trusted code.

To deal with such scenarios, the behaviour of demandPermission can be influenced by using *stack walk modifiers*. Two modifiers are possible.

- If a method calls the method enablePermission(P), the current activation record will be marked with an enable(P) modifier. The meaning of this modifier is that the current method takes responsibility for all its callers. In other words, during stack inspection, all activation records below the modified record need not be checked if the permission P (or some sub-permission) is being demanded.

  Using the enablePermission modifier, a piece of more trusted code can perform work that requires certain privileges on behalf of less trusted code. For instance, the piece of trusted code that opens windows on behalf of applets will enable the permission to open windows. The access check for opening a window will now pass.

- If a method calls the method disablePermission(P), the current activation record will be marked with a disable(P) modifier. The meaning is that the current method does not want to be granted permission P during its further execution, even if the security policy did grant it that permission. In other words, during a stack inspection for permission P, if a disable(P) modifier is encountered, demandPermission will fail.

  The disablePermission modifier allows you to dynamically narrow your privileges before you start work where you will rely on possibly less trusted code. Also, if trusted code is interpreting some (less trusted) script, it might be hard to validate that the script does not perform any prohibited actions, and disabling all unnecessary privileges can help sandboxing the execution of the script.

In figures 6.3 and 6.4 some example stacks with modifiers are shown.

In figure 6.3, the use of the enablePermission modifier is illustrated. Suppose an applet has a showResults() method that opens a window. Under the

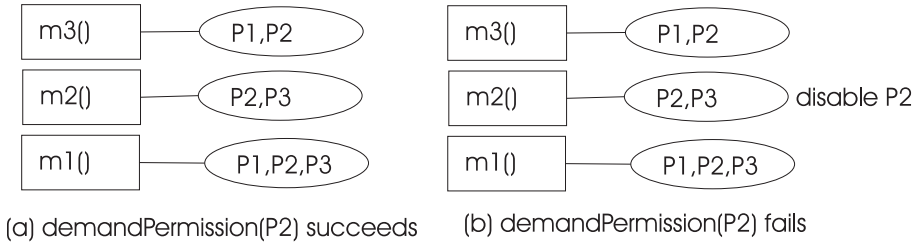

Figure 6.4: Illustration of the disable modifier

assumption that applets do not have the permission to open new windows, the applet asks a trusted component to open a marked window for it using the openMarkedWindow() method. This method will in turn call openWindow() and will make sure an appropriate warning label is attached to the newly created window. If the trusted component does not enable the WindowPermission, the request of the applet will fail: this is shown on the left of the figure. The demandPermission call fails because not all methods on the stack have the appropriate permission. If the trusted component does enable the permission (shown on the right of the figure), the demandPermission call for WindowPermission will succeed.

In figure 6.4, the effect of a disablePermission call is illustrated in a similar way. In this figure, two call stacks are shown, and each activation record on the call stacks is decorated with the permissions assigned to the component containing the method corresponding to the activation record. For instance, method m2() belongs to a component that has been assigned permissions P2 and P3 by the security policy. In case (a) demandPermission(P2) succeeds. In case (b), it will fail because of the disable modifier.

There are some further complications in the stack inspection algorithm. If a component starts a new thread, this thread will of course start with a new call stack, and hence all information about the current call stack is not available any more on the new thread's stack. Hence, the algorithm will not only check the current call stack, but will also check an inherited access control context. This inherited context will be attached to any new thread, and will essentially summarize the access control information on the call stack of the thread that created the new thread.

To summarize, the algorithm implemented by demandPermission(P) looks something like this:

```
void demandPermission(Permission P) {
  for each caller on the stack, from top to bottom {
    if (caller lacks Permission P)
```

```
      throw exception;
    if (caller has disabled Permission P)
      throw exception;
    if (caller has enabled Permission P)
      return;
  }

  check inherited access control context in a similar way
  return;
}
```

6.8 Extensibility

A stack inspection based security architecture can be made extensible in many ways.

6.8.1 Developer defined permissions

A first axis of extensibility is allowing developers to define new kinds of permissions. If you write code that encapsulates a resource of some kind, and you want to be able to limit access to that resource for partially untrusted code, you can define a new permission class, and put demands for that permission in all methods that access the resource. Of course, this only makes sense if the policy system is sufficiently flexible to allow for assigning these new permissions to components.

Recall for instance the example from section 6.7.2 where a browser implements an openMarkedWindow() to give applets the possibility to open windows that are clearly marked as applet windows on the users desktop. If you want to be able to selectively grant this right to open marked windows to some applets and deny it to others, you could define an OpenMarkedWindowPermission, and through the policy system assign this permission to some applets. Of course, the implementation of the openMarkedWindow() method should include a demandPermission for an OpenMarkedWindowPermission.

6.8.2 Developer defined evidence

A second axis of extensibility is to have the security architecture recognize new kinds of evidence. Mobile components could carry new kinds of evidence

and the policy system should be extensible to recognize these new kinds of evidence, and base permission assignments on them.

An example of such a new kind of evidence could be a third-party certification that the code complies to a given safety or security standard.

6.9 Example: The Java Security Architecture

The Java Security Architecture was the first security architecture using stack inspection. Since Java 2, a security architecture based on stack inspection has been part of the Java platform. The java.security.AccessController.checkPermission() method initiates a stack walk.

The implementation in Java is in some places a simplification of the model we have described in this paper, and in other places an extension of it.

In the first place, Java does not support all stack walk modifiers: the disable modifier is not supported, and only a simplified version of the enable modifier is implemented: the AccessController.doPrivileged() method enables all permissions of the current caller. There are however possibilities to implement enabling of permissions in finer granularity, but they require more work from the developer.

Secondly, Java does not support extensible evidence out of the box: only code location and digital signatures can be used as evidence.

Java does however provide a way to make checkPermission() calls also do user based access control. Since one of the things to make the stack inspection system work is the assumption that all code that exposes a resource performs an explicit call to a checkPermission(), it is certainly reasonable to make use of the presence of this call also for user-based access control. After authentication of a subject, the permission sets assigned to the activation records on the stack can be dynamically recomputed based on the subject running the current thread. In the security policy, permissions can be granted based on the two kinds of evidence (location and signatures) *and* on the principal currently executing.

For more details about the Java Security Architecture, consult section 6.11 for references.

6.10 Example: Code Access Security in the .NET platform

The Microsoft .NET platform has a complete implementation of the stack inspection based access control infrastructure described in this paper. Microsoft calls it Code Access Security. Demanding permissions is done by calling the Demand() method on a Permission object. In a similar way, calling Assert() on a Permission will enable that permission and calling Deny() will disable it. Both Permissions and Evidence are extensible.

The .NET platform goes a significant step further by allowing the specification of each of these three security actions in a declarative way: a developer can attach attributes to methods that will trigger demanding, asserting or denying permissions at run time. The advantage of this approach is that it leads to a cleaner separation of the security concern from the application code, and that tools can inspect components and find out what permissions they actually need.

The .NET Code Access Security system also has a richer but more complex notion of permissions. Permissions can reflect the right to access a resource, as we have discussed, or they can directly reflect evidence. A URLIdentityPermission for instance reflects the fact that a component was downloaded from a specific URL encoded in the permission. Demanding such permissions allows a developer of a method to restrict the callers of that method on the basis of evidence.

.NET also supports a more efficient but less secure kind of demand, a so-called link demand, that only checks if the immediate caller of your method has the demanded permission. Using link demands and permissions that reflect evidence, a developer can specify with fine granularity what code is allowed to link to his methods. In that way, link demands and permissions that reflect evidence can be seen as a very expressive kind of visibility modifiers of methods.

In addition the .NET platform comes with a well thought out policy system, based on the concept of *code groups*, that allows administrators to specify the code access security policy in a way that is similar to the way they specify access control policy for users. A code group has a membership condition, a boolean function that tests evidence of components. All components that are members of a certain code group (i.e. pass the membership condition) are assigned a code group-specific (and possibly dynamically computed) set of permissions. Code groups can be arranged hierarchically in a tree to form a policy level, and the .NET platform supports multiple such policy levels (e.g. a user policy level, machine policy level and enterprise

policy level). The effective set of permissions granted to a component is the intersection of the permissions granted by each policy level. In this way, the enterprise policy level can set a baseline policy, and the machine and user policy levels can further restrict this (but cannot assign more permissions than permitted by the baseline policy).

The platform does not have a mechanism to reuse the permission demands for user based access control as Java does. It does support another kind of integration with user based access control: demanding a so-called PrincipalPermission will check the identity of the current principal instead of initiating a stack walk.

The .NET implementation of stack inspection includes many other extensions. For the full details, check section 6.11 for some references to more detailed documentation.

6.11 Further Reading

The notion of component that is used in this chapter is closely related to Szyperski's definition of component. His book ([14]) is an excellent reference for a detailed discussion of components.

Types and type-safety of programming languages have been studied for a long time in the programming language community. A very good introductory paper on types and type-safety has been written by Cardelli ([17]). Readers interested to learn more about the type-safety of Java can consult [19]. For the .NET platform, a good reference is [18].

The idea of stack inspection was first implemented by Netscape in Netscape Navigator 4.0, and soon thereafter, Internet explorer and the Sun Java Development Kit also implemented a stack inspection based security architecture. The history of stack inspection, and an in-depth analysis of the technology can be found in the PhD thesis of Dan Wallach ([20]). In one of his papers ([21]), he compares stack inspection to other possible technologies for securely loading mobile code.

If you want to know more about the Java Security Architecture, the definitive reference is the book ([4]), co-authored by Li Gong, the architect of the Java Security Architecture. A more superficial description, but perhaps easier for a first encounter is the book by Scott Oaks ([9]).

The security features in the .NET platform are discussed in detail in the book by LaMacchia et al. ([7]). There are also some papers about Code Access Security in the documentation of the .NET Software Development Kit.

Recently, some researchers have proposed to move from stack inspection

based access control to the more general idea of history based access control. A good discussion of the disadvantages of stack inspection, and how history based access control can alleviate them is in [16].

Part III

Secure Software Applications

Chapter 7

Software Vulnerabilities

7.1 Introduction

In previous chapters, we have discussed different technologies for making software secure. All this technology has existed for many years, and yet a lot of software developed today still contains a lot of vulnerabilities. It is interesting to look at how software actually breaks in practice.

In this chapter, we give an extensive overview of real-life software vulnerabilities. Experience shows that a majority of these software vulnerabilities can be traced back to a relatively small number of causes: software developers are making the same mistakes over and over again.

Hence, it is useful to try to survey and classify the most frequently occurring types of vulnerabilities. In the following sections, a number of categories of software vulnerabilities are identified, and extensive examples of each of these categories are given. A software engineer familiar with these categories of problems is less likely to fall prey to these same problems again in his own software.

We also look in detail at the technical aspects of one of the most important vulnerabilities: the buffer overflow vulnerability.

7.2 An illustrated survey of software vulnerabilities

Any attempt to classify causes of software vulnerabilities must make some more or less arbitrary choices. There is no strict hierarchy of "types of mistakes" one can make. We base the top-level of our classification on whether the flaw was introduced at implementation time, design time or deployment

time.

7.2.1 Implementation flaws

Detecting and fixing (implementation) bugs in large software systems is known to be very hard. Security related bugs are typically even harder to detect, the reason being that security-testing is typically a creative form of testing: the tester has to think of the various ways in which an attacker might try to invade the system. Also, an attacker will often study source code to see how he can bring the software system in an inconsistent state, whereas automated testing systems typically consider more routine usage of the software system.

We discuss a number of categories of causes for implementation level vulnerabilities.

Insufficiently defensive input checking

A developer regularly makes (often implicit, and at first sight very reasonable) assumptions about the input to his programs. An attacker can invalidate these assumptions to his gain. It is important to realize that *input* should be interpreted in a broad way: input could be given to a program through files, network connections, environment variables, interaction with a user etc... If method calls in a program can cross protection domains (as is the case for instance in Java), even method parameters should be considered as non-trustworthy input that needs careful checking.

Examples of this category include: buffer overflows and weak CGI scripts.

Example: Buffer overflows Programming languages like C or C++, that are tuned for highly efficient execution and allow for low-level interaction with the computer's hardware, are typically not memory-safe. In other words, it is not guaranteed that a program can only access memory locations that are actually allocated to the program. For example, when accessing an array, no run-time check is made whether the index is out-of-bound, and using such an out-of-bound index will result in accessing memory that is not part of the array.

An array (typically a byte or character array) that is used to collect input is often referred to as a *buffer*. A *buffer overflow* is the situation where input is copied in such a buffer, but the actual input is bigger than the space allocated to the buffer. The consequence of a buffer overflow is that memory cells situated logically after the memory allocated to the buffer are overwritten. This overwriting can have a number of consequences:

- if the overwriting happens by accident, random errors or a program crash can occur: e.g. if the memory addresses written to are invalid for the process executing the program, the program will crash.

- but if the overwriting process is guided by an attacker (i.e. if the attacker constructs the input in a clever way), the program can be made to misbehave in a guided way. Typically, an attacker will try to have the program spawn an interactive shell, or will try to have the program install another more malicious piece of software on the computer.

The fact that such guided misbehavior is possible has to do with the way in which languages like C or C++ are executed. The technical details of how a buffer overflow attack works are discussed in the next section (section 7.3).

Even though they are easy to avoid (just perform bound checks on all array accesses), buffer overflows are one of the most frequently occurring and commonly exploited vulnerabilities. Two important causes for this are: (1) developers are not aware of the threat of buffer overflows and (2) the C runtime library contains a large number of string (char array) manipulating functions that do not check array bounds.

To prevent buffer overflows in their software, programmers can use a type and memory safe language like Java. If you have to work in a non memory safe language, you should program defensively, and carefully check sizes of all inputs. Also, a number of static analysis tools exist to assist you in finding buffer overflow vulnerabilities in legacy code.

Example: Weak CGI scripts CGI, an acronym for Common Gateway Interface, is an old but still widely used technology to produce dynamic web pages. The basic idea is the following: instead of returning a static html-file in response to a request by a browser, the web server will start an external process, possible pass it parameters included in the browser request, and return the output of the external process to the browser. Typically the external process will dynamically compute a result html-page based on the input given to it, and will possibly also cause some side effects, like sending an e-mail, or updating a database.

CGI programs are often written in a scripting language like Perl, Tcl or the Unix shell (in that case they are often called CGI scripts). These scripting languages have very powerful features and insufficiently defensive input checking in such languages can lead to disasters.

A typical example is the case where the script wants to send a mail to an e-mail address entered by the user of the browser. For example, the user

of the browser can submit his e-mail address via a form, and the result of posting the form is that some information is sent to the given e-mail address. Implementing such behavior in a CGI script written in the Unix shell, might be done as follows:

```
mail < infofile $clientadress
```

where `$clientadress`is the e-mail address sent via the form. The idea is that the contents of infofile should be sent to the given address.

If the format of this address is not thoroughly checked by the CGI script before executing the above line, an attacker can abuse this in many ways. If, for example, the attacker gives the following string as his e-mail address:

```
user@xxx.com | rm -rf /
```

then the line that will actually be executed by the CGI script is:

```
mail < infofile user@xxx.com | rm -rf /
```

In other words, the infofile will be sent, and afterwards, the entire file system will be removed.

Instead of wiping out the file system, an attacker could use the same technique to install a backdoor into the system (e.g. by adding a line to the password file).

Non-atomic check and use

A typical scenario in a security relevant part of a program is: check if some condition is ok, and if it is, perform some action. Often attacks are based on invalidating the condition between the check and the action.

A typical example is a so-called *race condition*. For example, a program checks to see if a certain filename in the temporary directory is available (i.e. no file with that name exists already), and if it does not exist, it opens a file with that name and starts writing information to it. An attacker can try to create a link with that specific name to a file he wants to alter between the check of existence and the actual opening of the file. As a consequence, the attacker causes the program to inadvertently add information to an existing file, where the program tried to enforce that it was really opening a new file.

A real-life example of such a race condition occurred in older implementations of the mkdir program in Unix. This program ran with root privileges (i.e. it is a so-called *set-user-id root* program, meaning that, even if it is started by a non-root user, it still has root privileges). Hence, it is interesting for a non-root user to try to make the program misbehave. The following race condition existed in the program. The creation of a directory was implemented in two non-atomic steps:

1. the program checked whether the name given as a parameter to the program did not yet exist as a file or directory, and if not, it created a new directory with the given name.

2. the ownership of the newly created directory was changed: the caller of mkdir was made owner.

Hence, the second step implicitly relies on the non-existence check done in the first step. If an attacker would succeed in having the second step applied to an existing file or directory, he could take ownership of any file or directory, including the password file. Because of the non-atomicity of the two steps, this attack was actually possible. An attacker would initiate the mkdir program and suspend it after the first step. With mkdir suspended, the attacker removes the newly created directory, and instead creates a link with the same name to the password file. Finally, the mkdir program is reactivated, and it makes the attacker the owner of the password file.

It may seem difficult to suspend the mkdir program exactly between the two steps, but this could easily be achieved by running mkdir as a low priority background process, and having a high priority process check for the existence of the directory to be created. As soon as the directory appears, the high priority process suspends mkdir.

Non-atomic check and use is not always an implementation flaw. It can also be built into the design of a system. A very simple example is the attack of simply typing commands at an unattended terminal: the operating system only checks the identity of the user at login-time, and from that moment on assumes that all commands from that terminal come from the authenticated user. The decision of only checking identity at login-time is a design decision and not an implementation aspect.

Programming bugs

Finally, ordinary programming bugs, i.e. flawed algorithmic logic in security sensitive software, are much harder to detect during testing than bugs in the functionality of the software. Security related bugs only show up in

the presence of malicious adversaries and hence can not easily be detected using automatic testing procedures. Moreover, the inherent complexity of cryptographic algorithms and other security related code makes it very hard to understand all relevant details and unfortunately wrong assumptions or small programming errors often introduce big security holes.

A first class of examples of this category have to do with incorrect implementations of cryptographic primitives. The security of a cryptographic system very heavily relies on the correctness of the implementation. Many details have to be taken care of. A real-life example is the weakness in the random number generator of older versions of Netscape, where random numbers where based on the current time (which is not random at all!), and this made it possible to break the keys used in secure connections within seconds.

Another example is known as the Java DNS bug. Here, the implementers of the applet execution environment tried to enforce the rule that an applet could only connect back to the site from which it was downloaded. Since sites can have more than one IP address, and more than one DNS name, the algorithm to check equality of two sites was:

1. Lookup the names of the two sites in DNS, resulting in two lists of IP addresses

2. The two sites are considered equal if the intersection of the two resulting lists is non-empty.

This logic is flawed, because an attacker that owns a certain domain name can choose the IP addresses that DNS returns for his site. Hence, he could cheat the equality test, and break the rule that an applet could only connect back to the site from which it was loaded.

It should be clear that the given example bugs would *never* be detected by an automated testing procedure: the bugs only show up in the presence of an active malicious adversary.

7.2.2 Design flaws

Relying on non-secure abstractions

Many abstractions offered by a programming language or by an operating system are *complexity-hiding* abstractions more than *tamper-proof* abstractions. Software developers often (implicitly or explicitly) assume that these abstractions are tamper-proof anyway, leading to security breaches. Examples include: type confusion problems in Java, attacks against smartcards, considering TCP/IP connections as reliable communication channels, unanticipated object reuse, etc... In all of these cases, an abstraction turns out

to be non-secure. And hence, if your design relies on the security of the abstraction, your application can be attacked by attacking the abstraction.

We discuss two examples in more detail.

Example: Type confusion in Java To allow untrusted code limited access to objects, it seems reasonable to use object oriented access specifiers (like private or protected) on methods or fields that should not be accessible to the untrusted code. The programmer relies on the information hiding aspects of the object oriented language he is working in to achieve a security related goal.

However, it is important to realize that access specifiers are by no means tamper-proof in most object oriented languages. For example, in C++, untrusted code can scan the entire memory range in use by a process by casting integers to pointers, and hence untrusted code can also access the private fields of any object. In other words, the OO abstractions offered by C++ are not secure, or C++ is not memory safe or type safe.

The designers of Java tried to make the Java Virtual Machine memory safe and type safe by disallowing pointers, and by checking casts even at runtime. But for many versions of the virtual machine, bugs in the implementation have led to breaches in memory and type safety. For example, the well-known classloader-attack (see section 7.4 for pointers to more details of this attack), breaks the type safety of all JVM's upto version 1.1.8.

Even in the absence of type safety problems, untrusted code may try to access private fields of an object by serializing the object, and reading the resulting file as a byte array.

Example: Unanticipated resource reuse The problem here is that the software developer disposes of some object or resource (e.g. deletes a file), and assumes that by disposing of the object, its information content becomes inaccessible. In many cases the object will only be *logically* deleted, and the actual content is still retrievable by an attacker.

Trading off security for convenience or functionality

It is well-known that there is a trade-off between security and convenience (i.e. functionality or user-friendliness of the software). Most security measures tend to add some user-annoyance, and often very powerful and convenient features are easy to abuse. Software developers, tending to think of functionality in the first place, usually emphasize convenience over security. This problem is often an attitude problem: software developers tend to spend a lot of time thinking about how to make things possible. From a security

point of view it is as important to spend time thinking about how to make certain things *impossible*.

A typical example of this class of vulnerabilities are e-mail clients that allow executable attachments. It is clear that the ability to execute the attachment to an e-mail by simply clicking on it, or even allowing attachments to be started automatically as soon as a user opens an e-mail message is convenient for the user. However, it also carries a number of substantial risks. As the waves of e-mail viruses of the past few years have shown, executable attachments make virus-spreading trivial.

Another example is the inclusion of powerful scripting or macro languages in applications like word processors, spread sheets, etc... Although such scripting languages can be very convenient to support customizable or dynamic content, they also represent a serious security risk. If the application is used to open documents coming from an untrusted source, the results can be disastrous.

Unanticipated (ab-)use of services and feature interaction

Highly successful services are often used (and abused) in ways never imagined by the designers of the service. Hence, the designers failed to provide safeguards for these abuses.

A typical example is e-mail. The Internet e-mail system, based on SMTP, was designed to provide a simple electronic messaging service for a relatively limited group of people. The unforeseen success of TCP/IP and the Internet has made SMTP a standard for a worldwide electronic mail system. Since sending e-mail is typically much cheaper than sending paper mail, advertisers have been abusing the e-mail system since many years, sending out advertisements to millions of addressees at once. Because the designers did not anticipate the enormous success of their protocol, they did not think of safeguards for protecting against such spam e-mail.

A special case of unanticipated abuse is *feature interaction*. As more and more features are added to a software product, they start interacting in unforeseen and insecure ways. An example is the telephone network, where the introduction of new services, like call-forwarding, conference calls and ringback have led to numerous security breaches.

7.2.3 Deployment flaws

Reuse of software in more hostile environments

Software written for use in a relatively friendly environment (like a mainframe or an intranet) is often reused later in a more hostile environment (like the Internet). Because the software developer made certain assumptions about the environment in which his program would be running, this change in environment can lead to major security holes. Typical examples include programs using password authentication, and document processing software reused as content viewers on the Internet.

Example: Password authentication Under the assumption that passwords are well chosen, and well guarded by their owners, password authentication is relatively secure in an environment where the communication between the user typing in the password and the computer verifying the password can not be eavesdropped on by attackers. Typically, for a terminal connected to a mainframe by a dedicated line, a password mechanism is sufficiently secure.

However, in a context where the password is communicated over the Internet, password authentication is extremely weak: eavesdropping on connections is commonplace on the Internet, and once a password has been seen by somebody else, the security of the mechanism is completely broken. Still, many popular programs like telnet and ftp rely on this mechanism to authenticate connections over the Internet.

Example: Document processing software reused as Internet content viewer If word processing software is used to create, edit and view documents authored by the owner of the software, or a small set of trusted colleagues, then the security requirements of this software are relatively low. If the software contains buffer overflow problems, it might crash occasionally, but it does not represent a major security problem.

This situation changes completely if the same word processing software is reused as a viewer for Internet content. The same buffer overflow problem can now be maliciously exploited: an attacker places a carefully constructed document on the Web, and tries to lure victims into viewing this document. By exploiting the buffer overflow problem, the attacker can do anything he wants on the victims computer.

Since it is difficult to predict in advance in which contexts your software will be used, it is good practice to strive for secure software development even for software that will initially only be used in a friendly environment.

Insecure defaults and difficult configuration

The default configuration of general purpose software is often not secure to guarantee that the majority of customers is able to use it without experiencing too many restrictions. This used to be common practice for operating systems, for example. Clearly, the users impression about the system is important and security restrictions might annoy him.

Even if the configuration options are clearly described in accompanying documentation, system administrators typically do not take the time to read this documentation. They tend to make a default install, and if that works leave it at that. Hence, if the default configuration is an insecure one, many installations will be in an insecure state.

Another reason why configurable security is often a cause for vulnerabilities are so-called *social engineering* attacks. In such an attack, one tries to convince the victim, e.g. over the phone, or via e-mail, or via a web page, to change his configuration. (E.g. "To run this fancy demo, you should change these and these settings in your configuration") A user without security experience will typically follow the directions given to him, thus opening up security holes.

7.3 Technical details of buffer overflow vulnerabilities

Buffer overflow attacks are always based on overflowing allocated memory: an attempt is made to write data past the bounds of an allocated chunk of memory. A program can allocate memory in a number of different ways: statically (the memory will be allocated at program load time), or dynamically (the memory is allocated at run time). Dynamic allocation of memory can be stack-based or heap-based. Stack based dynamic allocation is done for allocating space for local variables on invocation of a function or method. Heap based allocation is used for explicit allocation requests like the creation of a new object.

In this appendix we discuss the technical details of overflowing stack based dynamically allocated buffers. Stack based buffer overflows are much more common than heap based buffer overflows.

7.3.1 Stack based execution of imperative languages.

Function calls or method calls in imperative languages like C or C++ are executed on a typical modern computer by allocating the necessary memory

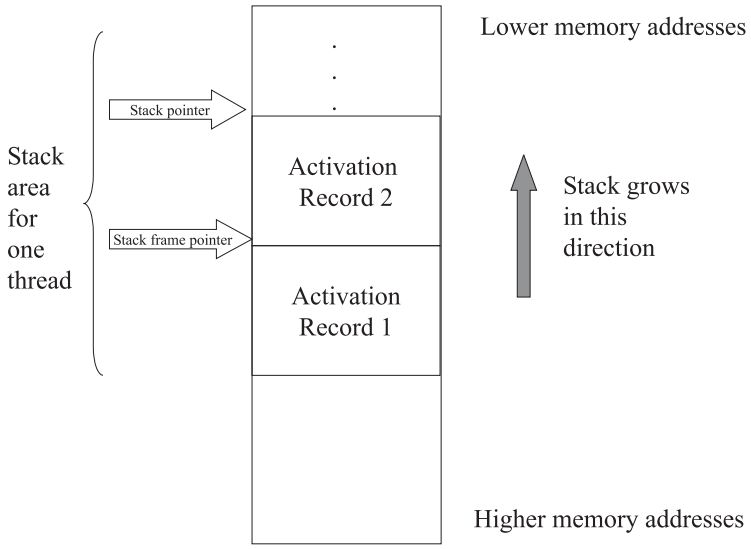

Figure 7.1: Organization of stack memory area

for the invocation on a stack. An area in the virtual memory space of a process is designated to serve as the stack for (a thread of) the process. For each thread, the operating system remembers the bounds of the memory area designated to the stack of that thread. The actual stack is then implemented using a stack pointer, a register of the processor pointing to the first free memory cell on the stack.

The stack starts off empty, with the stack pointer equal to the upper address of the designated area for the stack, and as the stack grows, the stack pointer moves to lower and lower memory addresses. If it reaches the lower address of the designated area, a stack overflow runtime error will occur (or the operating system will automatically increase the designated area). The organization of the stack memory area is shown in Figure 7.1.

The stack consists of stack frames or activation records. The Stack Frame Pointer (SFP) register contains a pointer to the start of the current stack frame. One such frame contains all information concerning one specific function invocation. It contains:

- the actual parameters of the current function call
- the local variables of the current function call
- the return address where execution should continue after return from the function

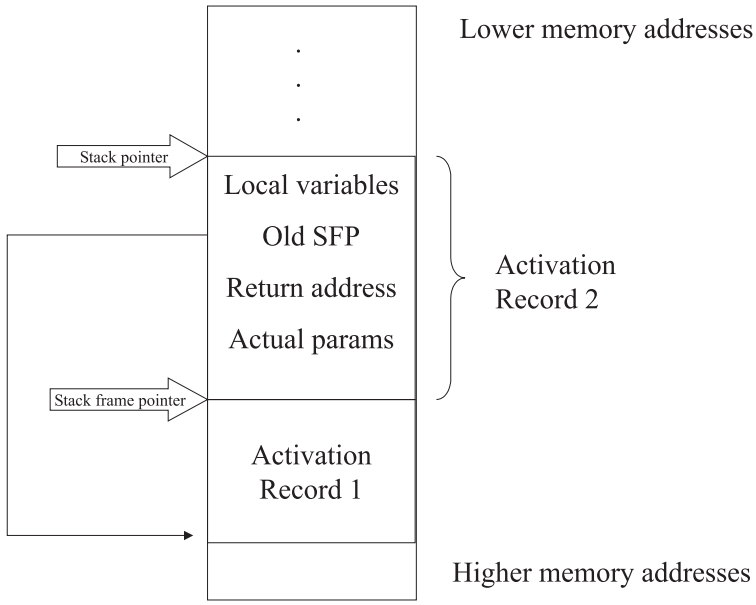

Figure 7.2: Contents of an Activation Record

- the previous stack frame pointer (that will be restored after the current function terminates)

The contents of an Activation Record are shown in Figure 7.2.

For example, consider the following C program:

```
void f(int a) {
  char x[10];
  printf("f invoked\n");
}

void g(int b, int c) {
  int d;
  int i[5];
  printf("g invoked\n");
  f(4);
}

void main() {
  g(12,13);
}
```

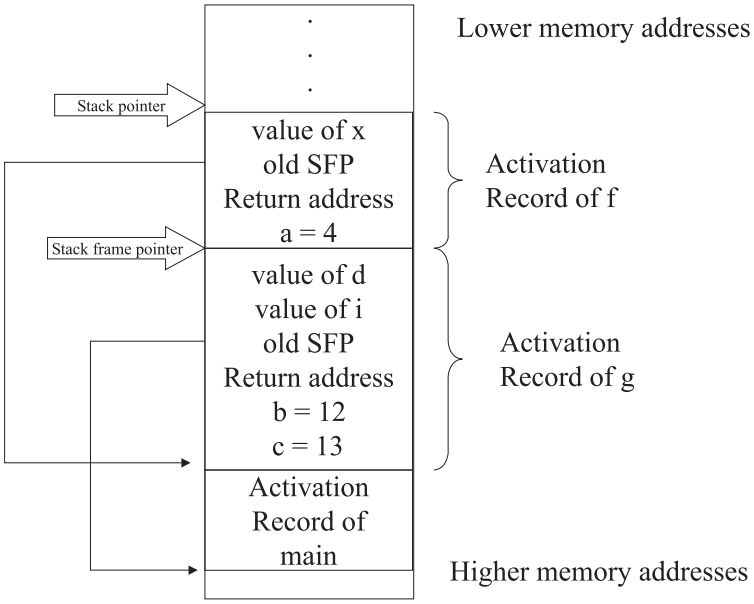

Figure 7.3: Example stack

Right before the execution of `printf("f invoked\n")`, the stack consists of 3 activation records as shown in Figure 7.3.

7.3.2 Overflowing a stack based buffer

A *buffer* is just another word for an allocated chunk of memory, typically a dynamically allocated array. If no bound checks are performed when writing to such an array, it is possible to overwrite memory cells in the vicinity of the buffer. This process of writing data to a buffer beyond its bounds is called *overflowing* the buffer. In particular, when overflowing a stack based buffer, it is often possible to overwrite other local variables, function parameters, or - and this is extremely interesting from an attack point of view - the return address of a function call.

C and C++ do not perform automatic array bounds checking like Java, and hence if the programmer does not write explicit code to check size of indices when writing to an array, a C program will typically be vulnerable to buffer overflows. Moreover, the C runtime library contains a large number of string (i.e. char array) manipulating functions that do not perform any bounds checking. E.g. the gets() function that reads a string from standard input does not check array bounds. Hence, the following C function is vulnerable to buffer overflow:

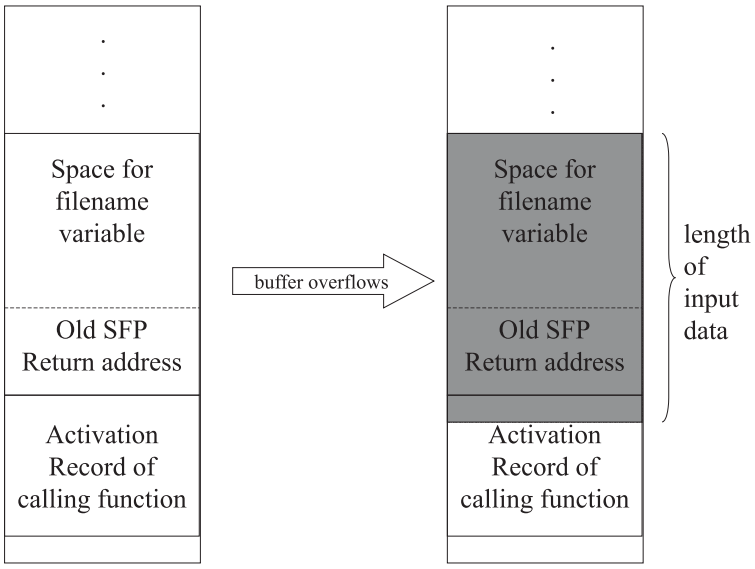

Figure 7.4: Overflowing the filename variable

```
void f() {
  char filename[20];
  printf("Enter filename:");
  gets(filename);
  /* Do something with filename */
}
```

If the user of the program types a filename of more than 20 characters, the filename buffer will overflow, and the part of the stack right above the filename buffer will be overwritten, as shown in Figure 7.4.

7.3.3 Exploiting a buffer overflow

A buffer overflow vulnerability can be exploited in two ways: by using the vulnerability to overwrite other security-sensitive variables of the program, or by overwriting the return address of a function.

Overwriting security-sensitive data

A typical example of this first kind of exploit was possible against old implementations of the login program of Unix. Essentially this login program reads a username and a password, and verifies if a hash of the read password is equal to a previously stored hash. If so, the user is allowed to log in, otherwise an "invalid login" message is generated.

The vulnerable implementation looked more or less like this (simplified):

```
int login() {
 char username[8];
 char hashed_pw[8];
 char password[8];
 printf("login:"); gets(username);
 lookup(username,hashed_pw); /* Put stored hash in hashed_pw */
 printf("password:"); gets(password);
 if (equal(hashed_pw, hash(password))) return OK;
 else return INVALID_LOGIN;
}
```

Clearly, both the username and the password buffers can be overflowed. By overflowing the password buffer, the hashed_pw variable can be overwritten. Hence, an attacker could type in a string that consisted of an 8 character word (say "xxxxxxxx") concatenated with the 8 character hash of that word. This 16 character password would overflow the password buffer, and hence would overwrite the hashed_pw variable with the correct hash of the first 8 characters. I.e. password would contain the string "xxxxxxxx", and hashed_pw would contain the hash of "xxxxxxxx" (see Figure 7.5). Hence, the login would succeed and the attacker is given access.

This first kind of exploit, overwriting security sensitive data, is very application specific: an attacker needs a bit of luck to pull this off: there must be a stack based security sensitive variable that can be overwritten in such a way that program flow is diverted to the advantage of the attacker. Since this is not a very likely situation, this kind of attack is quite rare.

Overwriting the return address

The second kind of exploit on the other hand is very common. The basic idea of this attack is to overwrite a return address, and since return addresses are always present on the stack, almost any stack based buffer overflow vulnerability can be exploited in this way.

When you overwrite the return address, you must choose what new address to write. Typically, an attacker wants to execute code to spawn an interactive shell, or code to install a Trojan horse or backdoor on the system. But of course, such code will typically not be present in the virtual memory of the attacked process. Hence, a buffer overflow attack of this second type, will typically also install the code to be executed in the buffer that is overflowed. That this is possible is caused by the fact that many modern

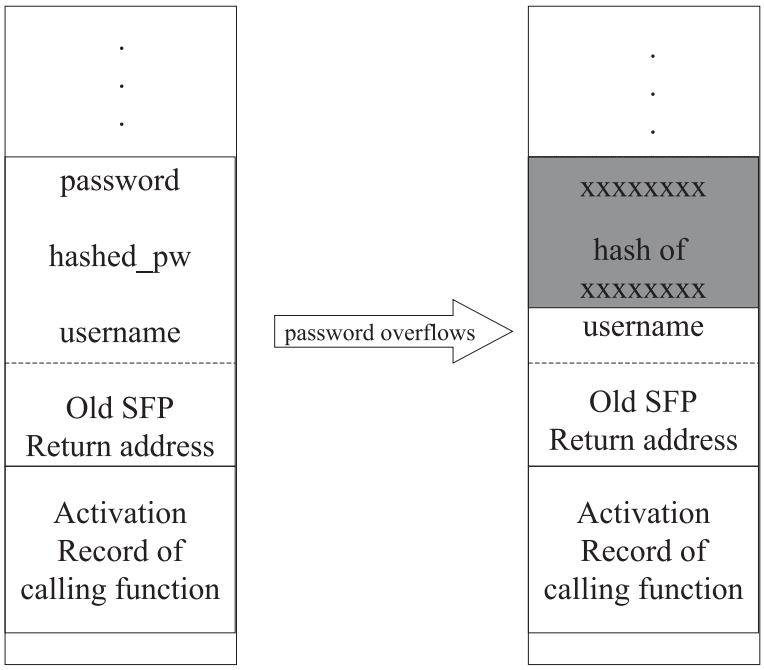

Figure 7.5: Buffer overflow attack against login.

computer architectures don't differentiate between code and data memory: any memory location can be considered to contain machine code, or data.

Hence, an attacker can put machine code in a data buffer, and at the same time overwrite the return address of a function to point to that machine code (see Figure 7.6). The consequence will be that the inserted machine code will be executed after return of the function who's return address was overwritten.

To succeed with this kind of attack, a lot of details have to be gotten right:

- an attacker needs machine code to spawn a shell that can be represented as a string (so that he can give it as input to the program),
- he needs to know the address at which the buffer will be stored, so that he can compute the correct value to write over the return address,
- he needs to make sure that he does not overwrite other critical data or the program could crash before returning from the current function,
- etc...

Getting all these details right requires some work: a detailed description can be found in the classic text of the hacker Aleph One (see section 7.4 for

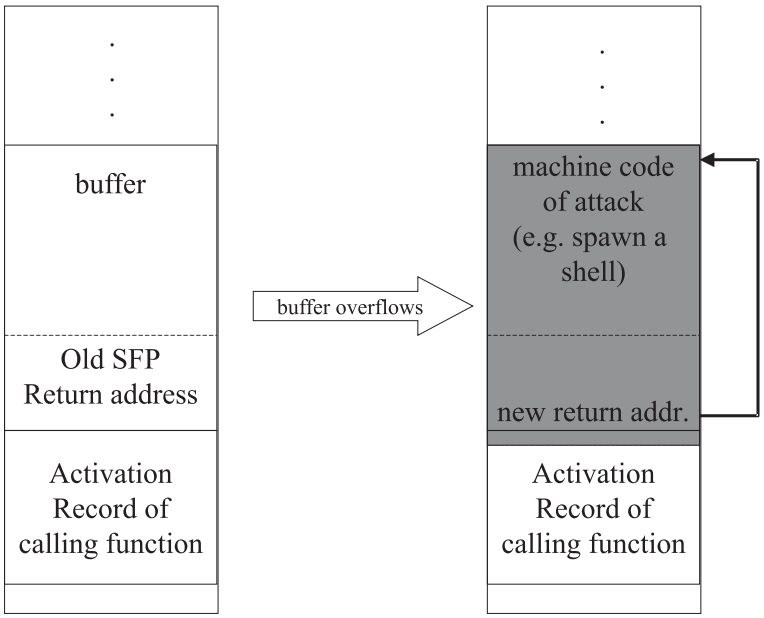

Figure 7.6: Overwriting the return address

pointers to this text). But once you have mastered the skill, implementing this second kind of buffer overflow attack becomes quite straightforward.

7.3.4 Finding buffer overflow vulnerabilities

If you have source code, finding buffer overflow vulnerabilities is relatively easy: you go over the source code looking for unchecked buffers, or for uses of C library functions that are known to do no bounds checking.

If you don't have source code, you can try to find buffer overflows by feeding the program very large strings everywhere the program expects input. If you can make the program crash in that way, you have probably uncovered a buffer overflow vulnerability. You can then proceed by trial and error to find which input you can use to overflow the buffer, and how to exploit it.

7.3.5 Conclusion

Buffer overflow vulnerabilities in server processes are one of the most commonly exploited vulnerabilities on the Internet. Yet, they have a surprisingly simple cause: a programmer forgot to do range checking for one of his arrays. The programmer often does not care too much about such bounds checks because he is not aware of the disastrous consequences that forgetting them can

have. Therefore, raising awareness of developers for this kind of vulnerability is extremely important.

It does take some technical knowledge to exploit a buffer overflow for the first time. But the typical situation on the Internet is that once somebody has done this difficult work, the exploit is posted to various hacker sites as a script that anyone can run against the vulnerable program.

7.4 Further Reading

TODO

Bibliography

Books

[1] Ross Anderson, *Security Engineering, A Guide To Building Dependable Distributed Systems*, John Wiley & Sons, 2001.

[2] Paul Garrett, *Making, Breaking Codes, An Introduction to Cryptology*, Prentice Hall, 2001.

[3] Dieter Gollmann, *Computer Security*, John Wiley & Sons Inc, 2000.

[4] Li Gong, Gary Ellison, Mary Dageforde, *Inside Java 2 Platform Security, Second edition: Architecture, API Design and Implementation*, Addison-Wesley, 2003

[5] Jamie J. Jaworski, Paul J. Perrone, *Java Security Handbook*, Sams Publishing, 2000.

[6] Jonathan B. Knudsen, *Java Cryptography*, O'Reilly, 1998.

[7] LaMacchia, B., Lange, S., Lyons, M., Martin, R., Price, K., *.NET Framework Security*,Addison Wesley, 2002.

[8] Alfred J. Menezes, Paul van Oorschot, Scott A. Vanston, *Handbook of applied cryptography*, CRC press, 1996.

[9] Scott Oaks, *Java Security, 2nd ed.*, O' Reilly, 2001.

[10] Charles Pfleeger, *Security in Computing*, Prentice Hall, 1996.

[11] Bruce Schneier, *Applied Cryptography*, John Wiley & Sons Inc, 1995.

[12] David A. Solomon, Mark E. Russinovich, *Inside Microsoft Windows 2000, Third Edition*, Microsoft Press, 2000.

[13] William Stallings, *Network Security Essentials*, Prentice Hall, 2000.

[14] D. G. Clemens Szyperski and S. Murer. *Component Software: Beyond Object-Oriented Programming, 2nd Ed.* Pearson Education, 2002.

[15] John Viega, Gary McGraw, *Building Secure Software*, Addison-Wesley, 2002.

Research papers

[16] Abadi, M., Fournet, C.: Access control based on execution history. To appear in the Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03), Internet Society. (2003)

[17] Luca Cardelli, Type systems, In Tucker, A.B., ed.: The Computer Science and Engineering Handbook. CRC Press, Boca Raton, FL (1997)

[18] Gordon, A.D., Syme, D.: Typing a multi-language intermediate code. ACM SIGPLAN Notices **36** (2001) 248–260

[19] Hartel, P., Moureau, L.: Formalizing the safety of Java, the Java Virtual Machine, and Java Card. ACM Computing Surveys **33** (2001) 517–558

[20] Wallach, D.: A New Approach to Mobile Code Security. PhD thesis, Princeton University (1999)

[21] Wallach, D.S., Balfanz, D., Dean, D., Felten, E.W.: Extensible security architectures for Java. In: 16th Symposium on Operating Systems Principles. (1997) 116–128

Standards

[22] National Institute of Standards and Technology, "Common Criteria for Information Technology Security Evaluation", ISO IS 15408, Version 2.1, August 1999.

[23] International Standards Organization, "Information Technology – Code of practice for information security management", ISO IS 17799, First edition, December 2000.

Websites

[24] http://java.sun.com/

[25] http://java.sun.com/security/

[26] http://web.mit.edu/kerberos/www/

[27] http://www.cert.org

[28] http://www.cs.auckland.ac.nz/ pgut001/pubs/x509guide.txt

[29] http://www.pgpi.org/

[30] http://www.sans.org

[31] http://www.cs.princeton.edu/sip/

[32] http://www.ibm.com/developerworks/security/

[33] http://msdn.microsoft.com/library/