Chapter 5. Application and language security

5.1 Introduction

Programs are the means by which computers are controlled. As such, malicious programs can be used to control computers in ways not intended by the owners of the computers. In addition—since many applications are complex artifacts—applications often have unintended flaws that may be exploited by intruders. This chapter considers software from a security viewpoint.

A variety of attacks occur through the execution of applications due to the characteristics of the languages used. The most common problem is buffer overflow. Attacks from downloaded contents were initially serious but have been controlled with recent advances. Other application-related attacks include misuse of pointers, misuse of data types (e.g., execution of data), and bypassing entry points that have access controls. Large applications may use components, which brings new security problems.

5.2 Malicious software: Trojan horses, Viruses and Worms

The array of malicious software includes viruses, worms, Trojan horses, logic bombs and other artifacts. The dividing lines between these programs are often blurred and often a particular program includes characteristics of more than one form of malicious software. We revisit now the definitions of Chapter 1 and provide more details of their work.

As stated above, the distinctions between these forms of malicious software are not clear cut. To illustrate, viruses spread using viral replication technology. Most viruses also carry a payload intended to perform some action once some condition is met, such as when a certain date is reached. In this case the payload is in the form of a logic bomb that is set to activate on the given date or when some condition is true.

5.2.1 Computer Viruses

```
____ ____
| | | |
| | <- Original entry point ->| |
| | | | |
| | | | |
| | (Host program) | | | (Host program)
| | | | |
| | | | |
| | | | |
---- | |....|
| | |<- New entry point
| | |
| | | (Virus)
```

```
--|JUMP|
----
```
Figure 5.1: Operation of file virus; left: before infection, right: after infection
.
The next time that the infected program is executed, the viral code is activated initially. The viral code executes and, possibly a fraction of a second later, jumps to the original entry point that causes the original program to execute as if nothing as changed. During execution of the viral code the virus can infect other files or, more typically, install itself as a separate thread, process or memory resident code. Now executing separately from the original program, it can infect other files and/or activate its payload using whatever criteria it desires. (However, keep in mind thatproper operating systems control write access to files, the creation of processes and other low-level activities in the system; chapter 4 contains more information about security in operating systems, while section 5.4 below addresses specific issues to counter malicious software.). Boot viruses use a similar strategy to file viruses in that they insert themselves in an execution stream. Rather than using a file, they insert themselves in the boot code. Normally the boot sequence proceeds as follows: When the computer is switched on, code in ROM *(Read-only memory)* is activated. This code reads the code from the boot sector of the boot disk and execute it. The executed code loads the operating system files and activate them. Now the operating system is running and the boot process is essentially complete. A boot sector virus copies the original boot sector's contents to a new location on the disk. Next it writes its own code to the boot sector and patches the newly written code so that it will load and execute the copied (original) boot sector contents. Hence, the next time the computer is booted, the ROM code will load code from the boot sector (which has been replaced by viral code) and executes it. When the virus code executes it can, as in the case of file viruses, copy itself to other disks and/or install itself in memory so that is can later infect disks. (However, note that it cannot create a proper process since the operating system—which manages processes—has not been booted yet.) Once the viral code has completed its installation or replication activity, it loads the original boot sector from the copy made earlier and activates it. This in turn loads the operating system and activates it to complete the bootstrap process. Again the process seems to proceed normally, with the entire process slowed down by a fraction of a second to a few seconds.
Countermeasures that were introduced against virus infections became widespread soon. Since file viruses appended viral code to program files, users were told to watch the size of such files and become suspicious if any executable file increased in size. Virus scanners were usually based on a list of *signatures*, where each signature was a piece of code extracted from a known virus; the pieces of code were selected in a manner that made it unlikely (but usually not impossible) for the piece of code to occur in any other software. By scanning program files for these signatures, infections by known viruses can be detected and reported. In many cases it is also possible to build disinfectors that reverse the infection procedure: for a file virus (see figure 5.1 again) the original entry point of the program is recovered from the virus code and corrected in the header. After this the viral code is stripped from the file. Yet another solution sometimes used was the 'inoculation'

of program files against certain viruses: Many viruses check whether they have already infected a file or a disk. Without such a check, files can be continually re-infected that could make them grow to undesirable sizes where they would eventually cause the infection mechanism to break. (This was especially true at a time when floppy disks with a capacity of only 360 kilobytes were still in use and many viruses were in the region of 2 kilobytes per infection). Inoculation worked by attaching this identifying code that the virus used to check for its own presence to possible target files. When the virus did (eventually) attempt to infect the file, it found the identifying code and skipped the file to avoid a 'second' infection.

None of these countermeasures are, however, without problems. Starting with the latter, the major problem with inoculation is that viruses often share the same location in a file or a boot sector to write their identifying code. When two viruses do this, it is impossible to inoculate against both. One therefore has to 'know' which virus will strike next and inoculate against it. This is, in general, impossible. Even if it were possible to inoculate against multiple viruses, the thousands that are in the wild make it impractical to use inoculation as a serious tool, except in very specific circumstances.

A problem of disinfectors is that they cannot always restore a file to its exact original contents. Some early viruses were known as *overwriting* viruses because, rather than appending the virus, they overwrote part of the executable file. In such cases it is clearly impossible to restore the original contents (apart from reinstalling the software or retrieving it from a backup copy). Perhaps a less serious issue arises when a virus does append itself to a file, but before appending, pads the original file to a paragraph boundary (ie extends the file until its size is a multiple of 16 bytes). The virus can be removed, but it is impossible to tell exactly how much padding has been added. While the padding is harmless, the file is not restored to its original version and a later size comparison of the file with the same application installed on a different machine may give rise to false alarms.

Virus scanners are probably the major remaining tool of those listed here. In practice they have proven to be quite effective. In general, selection of signatures to scan for, have been adequate, so that relatively few false alarms were generated. Most problems associated with virus scanners are the inconvenience of their use: Occasionally scanning an entire disk takes time. And the frequency at which their signature files need to be updated to cater for the latest viruses usually exceed users' conscientiousness.

When considering file sizes, it soon becomes clear that the average user is not aware of the size of program files they generally use. Even when the process is automated, so that file sizes and checksums for the contents of files are calculated and stored by some utility, all problems are not eliminated. Some programs use the (dubious) practice of storing program configuration information in the executable file. While this has the advantage of needing no separate configuration file, it means that an integrity checker will notice the change and, yet again, the average user will not know whether the change in content could have been caused by a virus.

The countermeasures considered here are mostly intended as specific antivirus tools. Section 5.4 below will consider more generic countermeasures that

are not only intended to protect users against computer viruses.
More interesting for the current discussion is the effect that these countermeasures had on virus writing. The fact that viruses could be detected by a known signature, by a change in the size of the executable file and/or a change in the con8 tents of the file, lead to the development of stealth viruses. Stealth viruses work by installing a process in memory that disinfects a file when the file is opened, but re-infects it when the file is closed. This means that any type of scanner that systematically
works through a disk by opening one file after the other and checking it for a signature or for changes will see the disinfected file — in other words, it will find no trace of the virus. However, when each file is closed the file will be re-infected (or infected if it has not been infected previously). Similarly, any calls to determine file length are intercepted and decreased by the length of the virus if the file is indeed infected. Thus stealth viruses are able to avoid detection by normal means. For anti-virus software writers this had the implication that their software should check for the presence of the virus in memory before attempting to scan any disk. (A still better solution is to boot from a known uninfected boot disk on a read-only medium before scanning. This practice has, however, become less prevalent over the years. Perhaps this is due to the fact that targeted operating systems have become larger and more complex and licensing conditions make it harder to construct suitable versions of commercial operating systems.)
Another impact that anti-virus tools had on virus writing was the development of polymorphic viruses. In the case of a polymorphic virus, the virus is encrypted with a random key and a small decrypter is inserted and patched to use this key. This means that the bulk of the virus will be different for each infection and this will make it harder to detect it using a signature-based scheme. Note that the encryption used does not hide the content of the virus—the key forms part of the virus and is very easy to extract. This (very weak form) of encryption is simply used to vary the content as it occurs in infected files and on disks. While it is still possible to scan for the decrypter, the decrypter can be so small (and can itself be modified randomly by inserting No-operation instructions and other superfluous instructions — such as adding zero to some register) that makes it impractical to simply scan for it.
This section has described computer viruses in the form in which they originally appeared in the wild. This was a good starting place for the discussion, because it highlighted the 'pure' notion of a virus. While these viruses are still found in the wild, they are not the major form of virus problem found today. However, before newer 'virus threats' can be considered, other malicious software has to be described.

5.2.2 Worms

Worms are similar to viruses in the sense that they include their own distribution mechanisms (and do not need a human to install them on a system or copy them from one system to the next). However, unlike viruses they do not need a carrier program to infect to spread. Instead, a worm uses vulnerabilities on a target system to copy itself to the target system and, once copied, to activate the copy. The

copy now finds new targets and continues the replication effort, while the original, independently, continues its own replication effort.
To illustrate, it is possible for a worm to use a dictionary-based attack (see ...) to attempt to log on to a different system via the network. Preferably the worm wants to log on as the root or administrator user. Once the worm succeeds in logging on, it copies itself (using the password it has discovered) to the target system. Once copied, the copy is activated on the target system, and the source system logs out of the target system.
This example demonstrated the four steps that are required:
1. Identifying a target;
2. Compromising the target;
3. Copy the work code to the target; and
4. Activating the copy on the target.
Note that the steps are sometimes interchanged. A very early example of a worm will illustrate this: In 19... the IBM Christmas Tree worm spread internationally amongst many networked IBM mainframes. After selecting a target (see below for how this was done), a copy of the worm was attached to an e-mail message and sent to potential victims. The e-mail instructed the user to execute the attached program to see a Christmas message. (The worm spread during Christmas time, so the message would seem appropriate to many recipients.) When the recipient executed the attached program, a rudimentary Christmas tree was drawn on the screen, with good wishes. (Remember, this was in the days when character-based terminals were used to access mainframes.) More important than the picture and the message, however, was the fact that the worm at this time also scanned the user's e-mail address book and started mailing copies of itself to everyone in the address book.
This old example illustrates a number of points. Firstly, the copy is sent to the target using a legitimate channel before the target is compromised. Secondly, the vulnerability that is exploited to compromise the target is the human element in the process. (It was reported that many users, after being warned to delete any such e-mails they received, intentionally opened and executed them to see what the fuss was about.) Finally, the example illustrates that many modern worms (often referred to as e-mail 'viruses') tend to use a strategy that is, in fact, very old in the history of malicious software, and are still able to exploit it successfully.
One major difference between modern worms and the Christmas Tree, is the degree of skills a user needs to participate in the distribution chain. Whereas in the case of the Christmas Tree, the user had to extract and store a program file and then execute the program, the same operation in today's graphical operating systems is often accomplished by (double) clicking on the name of the attachment when reading e-mail. Moreover, in the case of at least one such operating system, the types of well-known files (such as executable files) are, by default, not displayed to the user. The user may therefore get a message with an invitation to look at the attached `funnypicture.jpg`. While looking at a JPEG image seems safe enough, it is possible that the real name of the attached file is `funnypicture.jpg.exe`, with the operating system hiding the technical details (that the file is actually an executable file) from the user to make the system

more ‘user-friendly’. Being an executable file means that instead of (only) displaying
a picture, it is a program that may execute with all the permissions the
user has.
Worms discussed thus far required some level of user participation. It is, however,
also possible to write worms that spread without any human interaction. To
illustrate this, consider a worm that uses a dictionary-based attack (see chapter
...) to log on to a remote system masquerading as a legitimate user. It is not hard
to find usernames: e-mail addresses and URLs for Web pages belonging to users
often can be parsed into a username and a hostname. If the dictionary attack for
any such user succeeds, the worm can log onto the remote site, upload a copy of
itself and activate it, before logging of again. (Various mechanisms exist that a
worm might use to hide the uploaded code from the user.)
Whereas a dictionary-based attack will not work against a user who has selected
a proper password and kept it secret, another vulnerability is often exploited
that works—often even in the case of a user who attempts to take reasonable steps
to secure his or her system: buffer overflows. The buffer overflow problem is discussed
below. Note that many worms use it to operate, but it is applied far wider:
—it forms the basis of many successful cracking attempts and is often ‘packaged’
into a script for attacks used by the so-called script kiddies (see section ???).

5.3 The buffer overflow problem

In a nutshell, a buffer overflow attack operates as follows: the attacker sends more
data than a buffer has space for. If the program accepting the data assumes that
the data will fit into the buffer and therefore does not check this before it starts
copying the data to the buffer, it can ‘overflow’ the buffer. The data that is written
beyond the end of the buffer will typically overwrite some other data. This can
cause the program to malfunction, the system to crash, or, in the ‘ideal’ case, for
the perpetrator to gain control of the machine. Exactly how this is done will be
described below.
The solution to the buffer overflow problem is very simple: before copying
data to a buffer, check that sufficient space is available. If not, abort the program
or raise an exception so that the program can deal with the situation. Many
programming languages will do this without any additional action by the programmer.
Many programmers, however, prefer to disable such automatic checks or use
programming languages that do not support such checks for the sake of speed.
The C programming language should be mentioned explicitly in this context.
C stores strings as arrays terminated by a null character. Often such strings are
not manipulated as arrays, but through so-called pointer arithmetic: rather than
manipulating the array as a data structure in its entirety, a pointer is used that
contains the address of the element of interest. To move to a new element, an
appropriate value is simply added to or subtracted from the pointer address. To
illustrate, consider how strings are copied. A function to copy a string simply
needs pointers to the starting point of the source and destination strings, and, perhaps,
the number of characters to copy. (If the number of characters to copy is
not specified, copying proceeds until the end of the source string, as indicated by
the null character.) The algorithm to implement this function is very simple: copy

the character pointed to by the source pointer to the position pointed to by the
destination pointer. Increment both pointers. Repeat this process until you have
just copied the null character or until you have copied the specified (maximum)
number of characters. This function is not only simple, but can be implemented
very efficiently, with many processors providing hardware support. The Intel 8086
family of processors (and all compatible processors) can implement this by loading
the (maximum) number of characters to be copied in the CX register, loading
the address of the source in the SI (*source index*) register and that of the destination
in the DI (*destination index*) register.[1] After this the following instruction

[1]For the sake of simplicity, the role of the segment registers are not discussed here. Similarly

does all the copying required: `REPNZ MOVSB` (*repeat while not zero, move string of bytes*). The time this instruction requires to execute depends on the length of
the string to be copied; however, it only needs to be fetched and decoded once and
then copies at the highest speed the internal microcode of the processor allows.
The appeal of the technique clearly lies in its simplicity and efficiency. The
problem lies in the fact that the copying function has no information about the
structure it is copying to and therefore cannot automatically check that is does not
overflow the destination variable. This leaves it to the programmer to add code to
do the check. However, now the programmer not only has to remember to add the
code, but the added code will also slow the program down: to check the length
of the source string, the program will have to scan the source string until it finds
the null character, before it scans the string to do the actual copying—effectively
doubling the time required to complete the operation. If one keeps in mind that
benchmarks are often used to compare products it is probably of little surprise
that new buffer overflow vulnerabilities are found so often—even in well-known
products.
The question about whether checks for buffer overflows are intentionally omitted
(for speed or other reasons) or whether they are omitted simply because programmers
are unaware of the consequences, is not as important as the question
about what the fact that the frequent discovery of new buffer overflow vulnerabilities
has to say about professionalism in IT. The buffer overflow problem was
already known in the 1960s. Now, several decades later, the problem is still disrupting
computing regularly and reports about the millions of dollars it costs to
patch systems appear on a regular basis—before one even begins to consider the
potential harm when systems containing confidential or private data, or critical
systems are compromised. One cannot help but wonder in how many professions
such a recurring error would be endured without serious consequences for the responsible
individual and/or vendor. Clearly, this is an area that IT as an industry
will have to address if it intends to be viewed in a professional light.
Before the operation of the underlying mechanism of a buffer overflow vulnerability
is described, consider an example of how such a vulnerability could be
exploited in real life. Suppose one writes a web server. The web server will accept
URLs and then serve the requested pages. Suppose a programmer "knows"
that a URL would never exceed a few hundred characters and allocate the buffer
based on that "knowledge." Such an approach may well work for all real URLs.

we ignore the manner in which the registers and instructions have been extended for the 32-bit architecture introduced by the Intel 386. These details have little impact on the current discussion.

However, once the fact that this assumption has been used becomes known, it is
relatively easy to fabricate a URL that overflows this buffer.
Finally, consider the details of how an unchecked buffer may lead to the effects
of exploiting it mentioned above.
The vast majority of current programming languages allocate local variables
on a stack. When a procedure (or method in an object-oriented system) is called,
the parameters to be sent to the procedure are pushed onto the stack. Next, the values
of some registers might be pushed onto the stack so that they can be restored
to their current values when the procedure returns. In particular, the program
counter (or instruction pointer) points to the instruction following the procedure
call. Since it is important to know which instruction to execute once the procedure
returns, this value needs to be preserved and is therefore pushed onto the stack at
this point. The very first thing that the called procedure does, is to allocate space
on top of the stack for its local variables. This data structure (including the parameters,
return address, local variables and other data), is known as a *stack frame*
or *activation record*.
While it is entirely possible that the stack can grow from low memory to high
memory, let us consider the case where the stack grows from high memory to
low memory. This situation is depicted in figure **??**. Since the local variables
were allocated on the stack last, they are sitting at the low addresses — lower,
in particular, than the address where the return address is located. If one were
able to store an arbitrary string to one of the local variables as described above,
one can achieve a malicious effect as follows. Suppose firstly that the string is
simply long enough to overwrite the return address. When the procedure returns,
the return address will no longer be valid and the program will be 'resumed' at the
new address on the stack. In all likelihood this will cause the program to crash.
A more effective attack will include code in the string that is used to overflow
the buffer. In addition, the return address will be set to point to this code. Therefore,
when the procedure returns, it will 'return' to the code in the buffer. Clearly,
when this code starts to execute, it will do whatever the attacker instructed it to
do. It is important to remember that the operating system will think that it is the
original program that is still running and it will still have all the permissions it had
prior to the attack. If it were running with the permissions of a specific user, it will
still be limited to those permissions. If it were running with superuser privileges,
the attacker has just taken over control of the machine.
If the stack is arranged from low to high, the return value may be changed by
writing an over long parameter; the remainder of the procedure is the same as in
the previous case.

5.4 Countermeasures to malicious software

Malicious software can only be countered using a comprehensive strategy focussing
on as many of the following lifecycle stages as possible:
1. Ensure that software is developed in a professional manner;
2. Inspect software for malicious content before installation;

3. Install software with the least level of trust possible;
4. Monitor the behavior of the software;
5. Monitor the appropriate forums to determine whether vulnerabilities have been discovered; and
6. Patch the software when necessary, applying the same caution that should be applied when new software is installed.

Each of these steps will now be considered in more detail.

It is relatively easy for a programmer to build code into a system that was not called for in the original specification (or omit code that is indeed required, for example checks that prevent buffer overflows). As an example consider the flight simulator included in some versions of Microsoft Excel (assuming that a flight simulator did not form part of the original specification). Many other examples exist where some sequence of keypresses and/or other actions by the user will activate such an undocumented (and indeed unnecessary) ‘feature’ of a program.

One of the best-known examples of (potentially) malicious code is backdoors that are included in software to allow the programmer to bypass the normal security checks to access the system. Often backdoors are useful during development, but are then forgotten when the system is deployed and awaiting discovery by a cracker (or open to abuse by the original programmers).

Many anecdotes also exist about logic bombs installed by programmers and set to activate when some logic condition occurs. According to one of the most popular versions the programmer usually inserts a check that causes the program to malfunction whenever the name of the programmer is removed from the payroll. The programmer then, according to the story, has to be re-hired at a significantly higher salary before he or she agrees to “fix” the problem. The point is not how many of these anecdotes are true, but the fact that it is indeed rather easy for an individual programmer to include such unwanted code in a software system (and make it relatively hard to locate later).

In fact, the extent to which such code can be introduced and disseminated through a system has been illustrated in the early 1980s by ... during his Turing award acceptance lecture. In this scenario malicious code is introduced in the compiler and over time permeates to almost every program compiled by that compiler [Get reference and verify accuracy of this paraphrasing].

Whether software that does not check for buffer overflows should be classified as malicious software is open to debate, but, given the remarks in section 5.3 above, a case for this can certainly be made. We will use the phrase *malicious negligence* as one form of *malicious intent* below.

Given our wide definition of malicious software, it is clear that no simple, fool-proof countermeasures during the software development phase will solve the problem. Here it needs to be addressed by a range of (non-technical) precautions, such as proper screening of applicants when hiring programmers, clear policy guidelines expressly prohibiting inclusion of such malicious aspects in any software developed, regular review of any new code and the use of teams in a manner that would need collusion to get malicious parts included in code.

Code not developed by the enterprise, but by an external party, or code acquired as a consumer product clearly cannot be controlled during development as can in-house code. Here it is important to ensure that one only acquire software from well-known companies that are known to have proper development practices in place. However, given the number of incidents reported in software of well-known companies, it is important for the community at large to impress the importance of good development practice on such companies.

It should further be remembered that not all malicious software are in the form of *programs* — often malicious software is written as macros or scripts. Usually macros and scripts are not subjected to the same development efforts that application and utility software is; in fact often such pieces of code are developed by users and administrators to assist with once-of tasks. Moreover, such code is usually embedded in files that are often perceived as data files. This clearly holds implications for organizations relying on professional development practices to counter the threat of malicious software.

The second countermeasure mentioned was inspection. The option to inspect source code is often touted as one of the major benefits of open source software. Source code inspection is, obviously, also an option for software developed inhouse. And, in the case of some commercial software, it is possible to acquire the right to inspect source code. However, in many cases the sheer size of such an effort makes it impractical. In the case of open source, the assumption is that there are so many programmers out there tinkering with the code to adapt it to their own needs, that any code with malicious intent (again including malicious negligence) will be noticed and reported. This assumption has not yet been tested (and is certainly questionable for many of the very small, specialized open-source projects).

Inspection of object code has turned out to be useful. Many virus scanners scan for ‘signatures’ of known viruses, as explained earlier in this chapter. Such scanners clearly have three major implications:

1. Since only known malicious software is detected, it is necessary to update the signature database regularly;
2. The fact that malicious software may be ‘concealed’ in various forms (such as when it occurs in compressed folders, or in MIME-encoded format), means that scanners need considerable intelligence to locate all malicious software; and
3. If all executable content is to be scanned before it can do any damage, considerable resources will be consumed by the scanning process.

The third countermeasure against malicious software above was the practice of installing software with the least level of trust possible. Installing software in this manner clearly limits the degree of damage it can do, irrespective of whether it is intentionally or negligently malicious. This may be done by creating one or more special users (such as *nobody* or a user specifically for a given application), assigning very limited rights to those users, and then executing the application with those rights. While this requirement seems obvious, it was the practice for many years (and in some cases still is) to execute dæmons (or servers) with the privileges of the superuser.

The next countermeasure is monitoring software when it executes. This may be done, for example, by installing a personal firewall that monitors the program's attempts to communicate with other computers. Often the user's permission is asked when an attempted connection is made. Personal firewalls differ in this respect from normal firewalls by knowing the identity of the application that attempts the connection. It is (in theory) simple to extend this idea to letting an individual program execute in a *sandbox*: A sandbox is an environment where *all* accesses to resources are intercepted and then allowed to proceed or not, depending on the needs of the given application. In practise, most programs use such a variety of resources that it becomes impractical for the user to exercise a specific choice for each attempted use. In fact, whether a choice only needs

to be made for attempted network connections or for all resource usage, such a choice is often too technical to expect the average user to make the choice. This is often dealt with by defining categories of software, assigning permissions to access resources based on these categories and then classifying software in terms of these categories. Unfortunately, this assumes that users (or installers) of software should understand these categories. In a world where time is money, users (and installers) often soon learn that installing software in the most trusted category, allows the software to execute without security software 'interfering' when they want to use the software.

The final two countermeasures (monitor the appropriate forums to determine whether vulnerabilities have been discovered, and patch the software when necessary) are self-explanatory and will not be discussed here in detail. However, note that such monitoring (and even patching) may be very time consuming and are candidates for automation. A scanner such as `nmap` will actively search for vulnerabilities and alert the user of new vulnerabilities and point the user to more information. Similarly, a number of current operating systems have options to allow updates to be installed automatically. Arguing whether this can indeed be done with the same caution that should be applied when new software is installed, is left as an exercise for the reader.

5.5 Covert channels

Some of the solutions against malicious software described above assumed that malicious software will use an overt (or explicit) channel to communicate confidential information with the outside world. A personal firewall, for example, can easily prevent an application from connecting to port 25—the port normally used for SMTP — if that program is not supposed to send e-mail. However, matters are significantly more complex if the application does not use such an explicit channel.

A *covert channel* is a means of communication that is not easily identified as a channel — or easily identified as a channel that carries the information that it is in fact used to carry.

To illustrate, a *timing channel* modulates information on the time it takes to produce its normal output. Suppose a web server is modified such that it still provides the usual information it is supposed to provide, but handles some URLs in a special manner. Suppose, for example, that someone knows that the stock price of a given company tends to increase at the moment it introduces new products to

the market. Now suppose that this someone is able to monitor database updates
using software on the web server the company uses and use that information to
slow down serving of one of the company's web pages (that is totally unrelated to
products). Note that we assume that the database contains a date after which the
entered data becomes public, but the perpetrator knows about the updates before
the public knows about the products by monitoring the speed at which the covert
channel page is served. The perpetrator is therefore able to buy the stocks with
'inside' knowledge, without necessarily being an insider.
While the previous example illustrated the notion of a covert channel, it is
not very realistic, because a programmer able to monitor database updates will
in all likelihood be able to gather more detailed information and devise ways of
supplying more details via a covert channel than what a timing channel allows.
For a more realistic example, consider malicious software in a multilevel secure
database. Here software is indeed allowed to manipulate highly sensitive
data, but a monitor prevents that software to communicate what it knows to a program
running at a lower clearance or to write any such information to a file at
a lower security layer. However, if software that is running at a lower clearance
level can observe the time the process at the higher level takes, a covert channel
can indeed be used to communicate information across levels. Similarly, if the
process at the higher layer is written to consume storage space dependent on what
sensitive information it learns, the lower process that works in conjunction with
it to establish a covert channel, can receive the communication by monitoring the
available storage.
Covert channels are clearly mechanisms with a relatively low bandwidth (although
it is possible to increase the bandwidth significantly above what the simple
examples here show—see the exercises below). The significant point that covert
channels demonstrate, is the following: Security mechanisms are often primarily
designed to protect the obvious access paths to information; perpetrators are by no
means bound to use those obvious paths. The security architect should therefore
attempt to establish what alternatives exist for a perpetrator to access information
and protect those access routes as appropriate.
What are the solutions to covert channels? Covert channels always employ
malicious software to establish the channel. Therefore, all the countermeasures
against malicious software mentioned above, apply here as well. What makes
covert channels harder to detect than most other classes of malicious software
is the fact that it is usually hard to establish what the sender is doing, because
nonobvious ways of communicating information may be used and it may be hard
to identify the receiving end of the channel to determine that something is indeed

being monitored. In addition to the steps outlined for malicious software in general
above, it may be necessary to ensure that software cannot access the same
resources. For the first example above, running the database and the web server
on different machines may solve the problem—assuming that the malicious software
cannot learn what it used to learn from now sniffing network traffic.

5.6 Theory of program protection — information flow and program analysis

The previous section outlined the problems associated with covert channels. However, why, if a malicious programmer can access the code, doesn't this programmer simply modify a program that processes confidential material so that the program simply uses a standard channel (such as writing to a file) to disclose the confidential information to the programmer?

One could obviously inspect the code to determine whether the program outputs any information to a file or other channel that it should not. Such an analysis would also have to verify that information simply based on sensitive information should also not be exported to untrusted destinations. While manual inspection is a possibility in theory, the size of even many straightforward programs would make this impractical. What is needed is a tool that can automate this process.

Such an automated program analyzer has to be based on information flow between containers. Such flow control is very similar to flow control discussed in the context of multilevel databases (see chapter **???**) but differs from it in two significant respects: (1) most of the containers in this case would be inside the program (as opposed to database fields or other objects outside the program); and (2) the analysis can be performed at compile and link-time (as opposed to runtime in the earlier case).

The majority of containers to be considered when analyzing a program would naturally be variables. In addition to variables, external files and database fields may indeed be employed as containers by the program. The first step to analyzing a program is to categorise all such containers into a security class (such as *confidential*, *secret* or *top secret*). In general, it is assumed that the security classes form a lattice.

Two types of flow need to be considered: direct and indirect. Assignment, input and output statements clearly cause direct information flow to occur. Conditional statements may cause indirect information flow to occur, and will be dis20 cussed below. However, first consider direct information flow.

Consider the assignment statement `destination := source`. Clearly, information flows from `source` to `destination`, and this may by indicated by writing `source )destination`. Whether this should be permitted depends on the security classes of `source` and `destination`; if the two classes are the same, or `destination` is in a higher class than `source`, the information is allowed to flow; it is not allowed to flow from a higher class to a lower class.

More formally, if information is allowed to flow from `source` to `destination` it may be indicated by writing `source !destination`, where the underscores are read as *'the class of'*; therefore the preceding rule reads: *information may flow from the class of* `source` *to the class of* `destination`*'*.

Some assignment statements, however, are not a simple copy operation. The statement `destination = source1 + source2`, for example, combines two containers and, as was the case for an MLS database, the classification of the two is the least upper bound of the two containers. In this context, _ is often used to indicate least upper bound. Therefore

```
source1 + source2 = source1 _ source2
```

A similar argument applies to other arithmetic and logic operators.
Now consider indirect information flow. To illustrate, consider the statement

```
if s=1 then u=1 else u=2
```

Clearly, the value of the (unsensitive) variable `u` depends on that of the (sensitive) variable `s`. Therefore `s ) u`. Unless `s ! u` this should not be allowed.
The classic paper on this topic is that by Denning and Denning [Dorothy E Denning and Peter J Denning, Certification of Programs for Secure Information Flow, Communications of the ACM, **20**, 7, 504–513, 1977]. The notation used in this book corresponds to that used in the paper by Denning and Denning. That paper treats other aspects of the problem as well—notably complex data structures, procedure calls and exception handling.

5.7 Protection in Java: Sandbox model and the Security Manager

One of the primary applications of Java was to be a language that could be used to write client-side applications that would be downloaded from the web. It was clear that executing foreign code on a routine basis would require some guarantee that the code would not compromise one's computer or local data in any way.
The first solution was to let such code execute in a sandbox: as already described above, a sandbox 'wraps' the executing program in such a manner that it does not have any direct access to any resources. The original Java sandbox did not allow a Java applet, downloaded from the web, access to any local resources, apart from that portion of the screen assigned to it when it was started. It was allowed to establish a connection to the outside world, but only to the site from which it was downloaded. In particular was any access to local files or disk drives prohibited.
While this sandbox approach ensured that local resources were not compromised, it turned out to be too restrictive for many applications. It was, for example, impossible to store a local configuration file that restored the applet's behavior to the user's preferences whenever the applet was executed. The absolute rules of the sandbox were therefore extended by the introduction of a security manager.
This security manager can allow an applet access to local (and other) resources, if configured by the user to allow such accesses.
The public methods of the Java `SecurityManager` class are given in figure **??**. It should be clear from the name of each method, which resource it protects, and the details are outside the scope of the current text. However, note that any method that manipulates, say, a thread will call the appropriate method in the installed security manager. If that method allows access it will return without doing anything, and the original method manipulating the thread will continue. However, if the `SecurityManager` method does not want to allow the operation to proceed, it will throw an exception that will cause control not to return to the method manipulating the thread.
Security may now be managed in one of two ways. Firstly, it is possible to develop an application that uses an access control list that governs the operation of the application. One may then give given portions of code the right to use specific

resources for specified purposes. It is, for example, possible to specify

```
grant {permission java.io.Filepermission "abc.dat", read}
```

to enable a program using this policy to read the file `abc.dat`. In addition to this basic specification, one may also add specifications about who should have signed the code for it to be trusted to perform the granted actions. Typically the policy file (containing the permissions) will be associated with a program at execution time. Most browsers, however, install a security manager that simply disallows any action that foreign code should not be allowed to activate. The rights granted by the major browsers' security managers are summarised in table [Should we quote?: Van der Linden, Just Java 2, 4th ed, Prentice Hall, table 14-1, p.358].
The second way in which a security manager could be used to manage security is by defining a subclass of the default `SecurityManager`. Methods that

should apply security different than the default case, should simply be redefined in the subclass to override the default method. Clearly, the range of checks that may be performed is as wide as the programmer's imagination and can use any pertinent information. As explained above, the method simply has to return if access should be allowed, or to throw a security exception if access is not to be allowed.

5.8 Components: J2EE and JAAS, .NET

5.9 Copyright protection

Whereas the rest of this chapter has focussed on the security of programs, this final section will briefly consider the use of mechanisms to prevent programs from being copied illegally.
The techniques that have been used over the years to prevent such copying can be classified into four categories: (1) Nonstandard use of storage media; (2) Dependance on a hardware device; (3) Customization and (4) The need to register software online.
In the early days of personal computing a great number of techniques fitting in the first category were used. On media that normally had 40 tracks, information was, for example, written on media with 41 tracks and special information was written to track 41. When the program was executed, it would attempt to read the information recorded on track 41. Since normal copying programs only copied the standard 40 tracks, it was possible to distinguish a copy (copied with a standard copying program) from the original.
In a similar manner, some software was distributed on disks where the sectors were not assigned the usual numbers. On a disk with nine sectors, eight of the nine sectors could be numbered from 1 to 8, and the ninth sector could be numbered, say, 77. The program itself would then check for information present on sector 77, but normal copying programs would only copy the standard nine sectors (and

would typically report a that a sector was not found).
The first weakness of these copy-protection mechanisms was that it was easy
to write bit-level copiers that copied the information on one disk to another with
out even considering aspects such as sector numbers. And they continued copying
as long as they found more sectors to be copied. These copy-protection mechanisms
were therefore easy to bypass. (Hardware copying devices were also built
that were able to copy some aspects of disks that were hard to copy using only
software.)
The second weakness of these copy-protection mechanisms was the fact that
they prevented non-technical owners of such software to make (legitimate) backup
copies of their software. Because the mechanisms went beyond the normal parameters
within which disks were intended to be used, they sometimes were simply
unreadable on some devices. And these mechanisms assumed that the software
would be executed with the distribution media present — which made it cumbersome
to use when installed on a hard disk. It therefore turned out that these
mechanisms were a major inconvenience to many legitimate users of software,
while it offered virtually no protection against software pirates.
The second alternative mentioned above was the dependance on a hardware
device. Often the device is attached to an input/output port (such as the printer or
serial port) and the program simply checks for its presence when it is executed.
Usually such devices are known as *dongles*. It is entirely possible to use a special
device into which a token may be inserted to vouch for the validity of the licence
to execute the program: a smart card reader may, for example, be used and a smart
card inserted that contains details of the licence and that may be queried by the
program at runtime. On the positive side, it is possible to build dongles and other
tokens that are impractical to forge. One of the main drawbacks of dongles and
similar tokens is the fact that they occupy a limited number of ports or slots. Many
dongles are transparent to normal printing or serial communications — in other
words, a dongle may be attached to the printer port and the printer attached to
the dongle. This alleviates the problem somewhat, but it is often not possible to
use one dongle through another dongle. If one were to use many programs that
require dongles, one would either have to go to the expense of installing many
ports (where the required number of ports is supported) and/or unplug dongles
and plug in other dongles. Again this may be quite a nuisance to the legitimate
user of the software, while the pirate may work in comfort after bypassing this
mechanism— as described below.
One solution that straddles the two categories described above was in use in the
1980s: A laser was used to physically destroy a sector on the disk. The program,
at runtime, checked for the presence of the destroyed sector by attempting to read
from it and expecting a hardware failure. While it suffered from many of the
drawbacks mentioned above, it was not possible to copy the disk using a special
copying program. This particular protection scheme was often bypassed by using
a special device driver that simulated the hardware failure whenever the program
attempted to read from the ‘damaged’ sector of a copied disk.
The third possibility is that of customization. The theory is that if the details

of the person who bought the software occur throughout the program, that person
is unlikely to share copies of the program with others. It is indeed possible to
make it very clear who owns the program by, for example, displaying the name
of the owner during a splash screen when the program is activated, on the *About*
details that are displayed when the appropriate menu option is selected, by including
these details as part of the header or footer of any reports that are printed, and
so on. Note that this approach does not require shipping a customized version to
each purchaser: it is possible to ship a generic version and expect the user to buy a
licence, that is then supplied with an activation code. The activation code then depends
on entering the exact owner information that was entered when the licence
was bought. Often such customization is quite effective where the software is
bought with honest intentions. However, suppose one person with enough money
buys the product (or a few colluding parties who together are willing to spend the
money, purchase a single copy). In this case the intentions of the purchaser are
dishonest and a false name is supplied. This single copy can now be copied by as
many individuals who can lay their hands on this copy, without publicly affecting
anyone's reputation.

Finally, the need to register software online was listed as a mechanism against
piracy. In this case, when the software is installed, it collects information about
the system on which it has been installed and hashes this information to a single,
large number. To activate the software, it is necessary to register the software,
using this number. The software vendor performs a cryptographic calculation that
depends on this hash and the licence number of the owner. The installed software
performs the same (or a related) calculation so that, when the vendor's result is
entered, the software can determine that it has been registered. If the software is
later installed on a (somewhat) different machine, the calculated values will be
different and the owner will have to contact the vendor again. The vendor is now
in a position to limit the number of times it will provide an activation code for a
single licence.

This approach also suffers from a number of drawbacks. Firstly, it requires
that the vendor is available to calculate an activation code whenever any user
wants to install the software. It also requires the vendor to be reachable from
any location where a user is likely to install the software. Furthermore, it is hard
to distinguish between a pirate and user who, for example, tests hardware and

therefore continually changes the configuration of his or her machine. In fact,
any such scheme should expect that most users could change the configuration
of their computers over time and should therefore allow the same licence number
to be used a number of times. The vendor cannot distinguish between legitimate
requests due to configuration changes and cases of piracy. However, it is possible
for a vendor to limit the number of times a particular copy is pirated.

Perhaps the biggest problem with this technique is the fact that it is cumbersome
to use in a corporate environment: often corporations have thousands of
personal computers; they often install a master image on a single machine and
then *cast* the image to all other computers. For them to be expected now to go to
every individual machine to get it registered is perhaps too much of a burden that

will cause them to consider alternative products. Users of this scheme therefore often do not use it for corporate products. This means that a pirate simply needs to get hold of a corporate distribution disk and related key and can then pirate as many copies as required.

Above, weaknesses in the individual schemes have been pointed out. However, they all share a common, major weakness: the code that checks whether a copy of software is legitimate, has to be embedded in the software. It is therefore possible to reverse engineer the application, determine how it checks its own validity and then modify that code to always think that it is indeed a valid copy. As the size of software increases, so this task becomes harder and harder. However, it is a task that may be performed in parallel — and as long as enough crackers are willing to collude using a worldwide medium like the Internet, any such technique may be bypassed. The only exception is the rare cases where some processing can be delegated to a secure piece of hardware (such as a smartcard) in a manner that such functionality cannot be emulated by software that does not know secrets stored on the device.

While technology may be used to help alleviate the problem of piracy, there clearly will be no final solution. The problem is ultimately one of ethics. One is certainly permitted to sell the fruits of one's labors at whatever price one chooses and expect others to conform to one's requirements. However, there are also a few cases where one has *no choice*, but to use some software. Note the emphasis on '*no choice*'. And in those cases it also becomes a matter of ethics to consider whether the exorbitant prices charged by some, are justified.

Buffer overflow attacks

This is an old problem that keeps recurring. It is mentioned in the early writings of Multics around 1970 and was already used in the Morris worm of 1988.

Programming languages implement their activation records using stacks, one of whose locations is reserved for the return address. A common way of attack uses an array that has a fixed and predefined size. The attacker puts in this array a larger amount of data than it can hold and makes it overflow. If the attacker knows the details of the hardware and language used he can overwrite the return address in the stack. Most systems perform the return operation in supervisor mode and the attacker gets his program to run in supervisor mode. Operations that can be used to do this include C constructs such as strcopy, strncopy, sprintf, strcat, memcpy, and others.

Some proposed solutions to the buffer overflow problem include [Cow00]:

- Prevent this problem of happening by testing the code [How00]. This is the approach taken by Microsoft. However, it is hard to detect overflow conditions in complex and large code.
- Use a language that checks bounds when copying data such as Java. PL/I had a special compiler that could generate array boundary checks.
- Add code to programs to do integrity checking of their activation records. This is laborious.

- Use a non-executable stack. Architectures that have stack segments such as Intel's can make these segments non-executable.
- Use a segmented address space. Now the compiler can allocate data structures such as arrays to their own segments. The instruction microcode then checks for operands being within the size of the segment. Segmented memory architectures are used in Intel and IBM machines.

The last two approaches show the need for appropriate hardware architectures. Obviously, having omnipotent execution modes is another problem. If the supervisor and similar processes have well defined rights according to a need-to-know policy, damage is more limited.

Attacks from downloaded contents
Downloaded contents pose security risks and there has been a large amount of work on their security aspects.
Some attacks that are possible are [McG97]:

- File reading and corruption
- Take over the operating system
- Denial of service
- Mail forging
- Leaking of client information

Figure 11.1 shows how the overall picture of these attacks. The downloaded applet may try to take control of the host or may download a collaborator to leak information through it or for help in its attack.

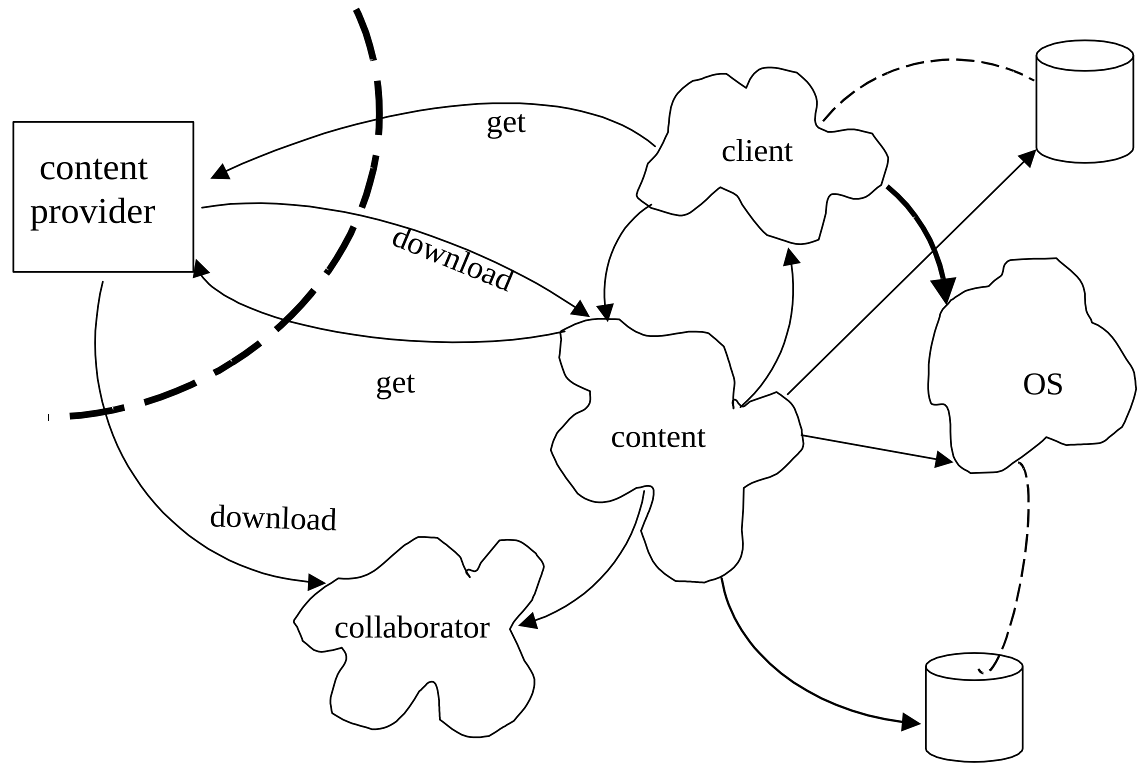

Figure 11.1 Active contents execution

The secure execution models that have been used for Java are [Har97a]:

- Sandbox. This was the first attempt at security. It restricts an applet to access only its own data and data from the place in the Internet where it came from.
- Signed software. A certificate guarantees that the software comes from a responsible institution.
- Authorization system. An access matrix model is enforced through the security manager or other mechanism (see later).

Microsoft never used the sandbox approach. Microsoft and Sun have similar approaches to signed software, using certificates. Microsoft's client side authorization uses a predefined set of zones as subjects. Sun uses protection domains. The two are restricted versions of the access matrix model. C# also uses a virtual machine and follows a similar approach.

A typical enforcement approach is the one used in IBM's Flexxguard (Figure 11.2). This applies a Reference Monitor (Chapter 3) that intercepts access to resources. A more specific approach, based on its execution architecture, is used by Sun. Another interesting approach is used my Microtrend [Che99].

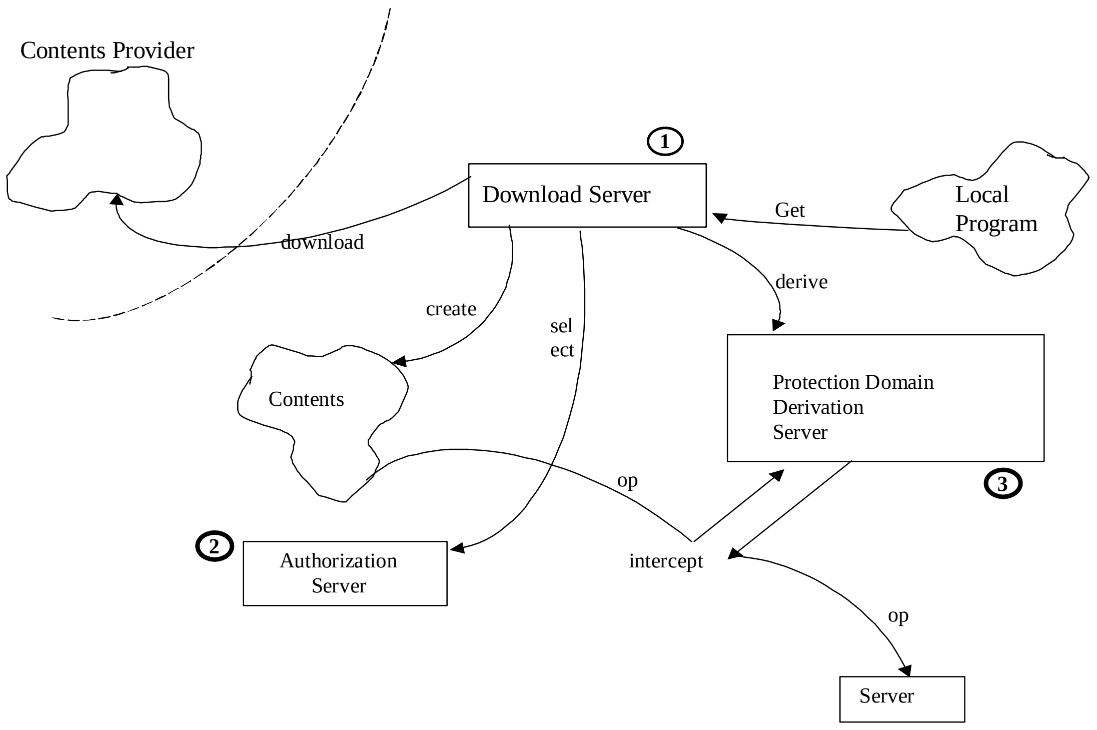

Figure 11.2 Enforcement of security constraints in Flexxguard

Programming languages and security
Languages can be used for security in three ways:
- To prevent actions that can be used for attacks, e.g., misuse of data types, manipulation of pointers.
- To reveal intention if the program with respect to data access to allow further access checking by other mechanisms [Sum75].
- To directly enforce security restrictions through the program constructs [Anc83, McE83]. For example, [McE83] describes a system that performs compile-time checks based on program analysis that enforces information flow policies.

The first approach is the most practical and used in languages such as Java and C#.

Java security
Java is strongly typed language that restricts the use of pointers and is in general a relatively simple language. Its restrictions are enforced through its virtual machine (JVM).

Figure 11.3 shows how a Java application is executed. Java programs are converted into Bytecodes that are interpreted by the Java Virtual Machine (JVM). The JVM is then a Single Point of Access for security enforcement [Yod97]. The security functions of the JVM include [Kov98, Oak01]:

- The ByteCode verifier--Enforces the language rules, e.g., no illegal data conversions, correct parameters.
- The Security Manager—Controls runtime access to system resources. It uses the Access Controller to enforce the security rules defined for downloaded content.
- The Class Loader—Determines how and when applets can load code.
- The Security Package—Used for authentication of Java classes. It implements the JCA discussed in Chapter 6.

Components and application models

There is no precise definition of component. In some views, a component is a unit of independent deployment, a unit of third party composition, and has no persistent state [Szy98]. Other definitions add contractually specified interfaces. Others say it must have state. Other definitions consider a component to be any package bought from a vendor as a unit. In the next chapter we consider web services, components that can be accesses through the Internet, here we consider only application components.

Java components are called Enterprise Java Beans (EJBs) and are used to build business models in the application server [Fer03, Kov01]. EJBs have Deployment Descriptors, which contain Access Control Entries (ACEs). Each entry identifies a person, group, or role that can access the whole bean or some specific methods. These components execute in the server and are protected by the Java Authentication and Authorization Service (JAAS). The JAAS also enforces the ACEs in the beans.

One of Sun's design goals when they created the J2EE platform was to separate the security aspects of the development of components, the assembly of these components into applications, and the deployment of these applications. For this purpose, three roles have been defined: the *Component Provider* and *Application Assembler* specify which parts of an application require security, and the *Deployer* selects the specific security mechanisms to enforce that protection.

Enforcement of security is based on two approaches: declarative and programmatic security. *Declarative security* is based on authorization rules defined in a J2EE system by the deployment descriptor. This is a contract between the Application Component provider and the Deployer. Groups of components can be associated with one deployment descriptor. When declarative security is not sufficient to express the security constraints of the application, the *programmatic security* approach can be used instead. It consists of four methods, which allow the components to make decisions based on the security role of the caller.

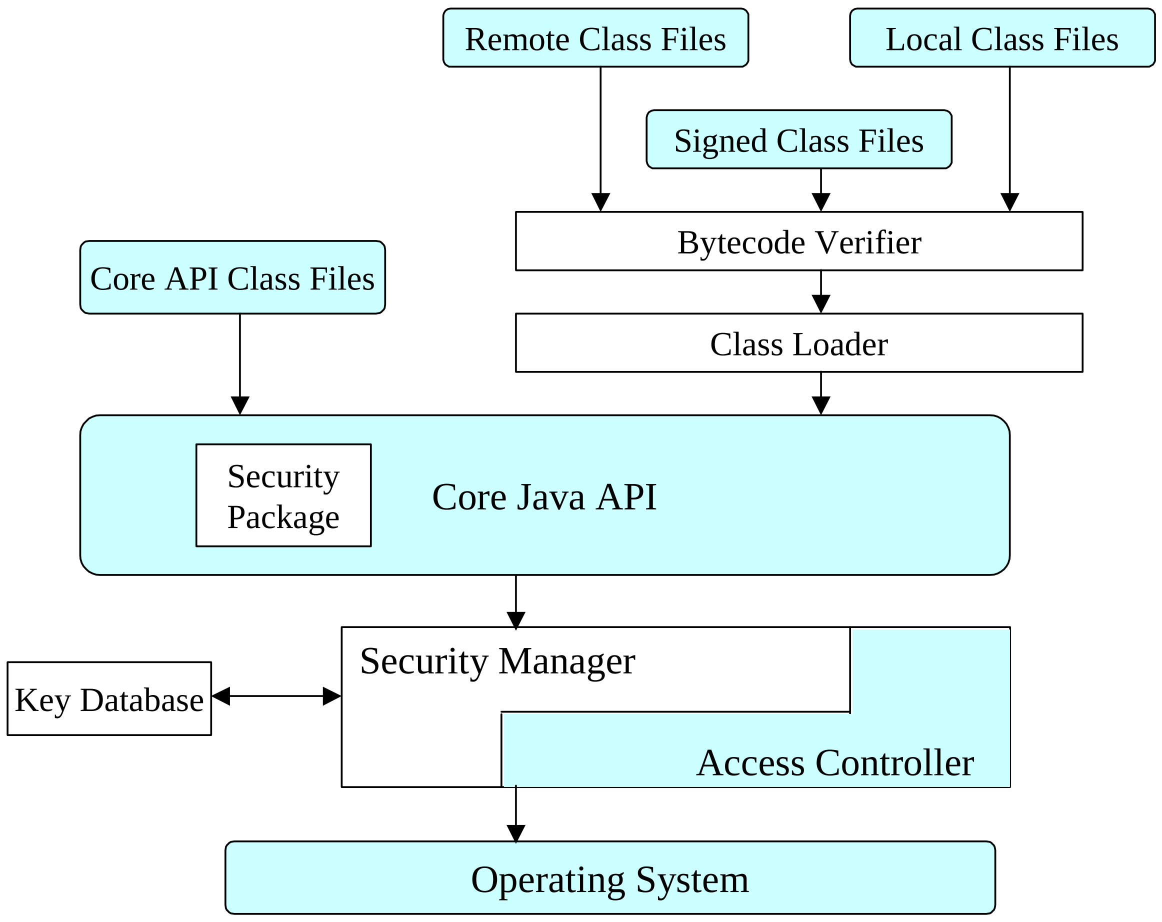

Figure 11.3 Java application security

The trade-offs between the declarative and programmatic security approaches are: The former is more flexible after the application has been written, and usually more comprehensible, and therefore results in fewer bugs. More important, it is done in a uniform way across the system. The latter could provide more functionality when the application is being written, but is buried in the application and is therefore difficult to change, and usually only fully understood by those who developed the application. Programmatic security does not allow checking for compliance with institution policies and alone is not acceptable for systems that require a high level of security [Sum97]. However, it could be useful to complement declarative security by adding special restrictions for application access, although it doesn't appear that it can be combined with declarative security in J2EE.

Microsoft has gone through three generations of components: COM, COM+, and .NET components. COM+ and .NET use an RBAC model with inheritance of rights [Edd99]. Roles can be defined at development or deployment time. At runtime the identity of the user on whose behalf the code is running is determined, and access is granted or denied based on those roles. These roles are typically mapped to credentials in Microsoft's Active Directory.

Role access in .NET components can be inherited from process to component to interface to method. This is an example of Implied Authorization (See Chapter 4), where access to a composite implies access to components; in particular, this can be interpreted as inheritance of authorizations along a generalization hierarchy as seen in Chapter 10. In .NET higher-level-defined accesses override lower-level rights [Low01], an opposite policy to the one defined in [Fer94]. For example, I may want to give access to the whole Student class to somebody but not allow that person to see the grades of specific students, something that cannot be done in .NET. A lower-level access is a more precise (finer) specification and it should override the higher and coarser access.

Figure 11.4 shows the ECF pattern [Kob00], that is an abstract representation of components and can describe by proper specialization EJB and COM components, including their security aspects.

Systems also use whole vendor packages as components, called Components off the shelf (COTS), and any of these could contain malicious code or be vulnerable to attacks. We cannot in general, control what is inside a component but if it has well-defined interfaces we can control what comes in and out. An interesting approach to add security for this type of components is used by Hewlett Packard [Zho98]. The idea is to partition a component into zones of trust and place specific parts of the component in different zones, related to each other by a multilevel model.

Secure systems design and application security

It is clear that to build secure systems one must select an appropriete language. Java (and possibly C#) appear as good languages for this purpose. Languages such as C++ could be used if restricted as it was done in [Sum75] for PL/I. If the application uses components, the choice is between J2EE and .NET. Neither of these is inherently more secure [Fer03], so the choice here is directed by other aspects: If one uses C#, one must also use .NET; if one uses Java then J2EE is necessary.

References

[Anc83] P. Ancilotti, M. Boari, and N. Lijtmaer, "Language features for access control", *IEEE Trans. on Software Eng.*, Vol. SE-9, No 1, January 1983, 16-24.

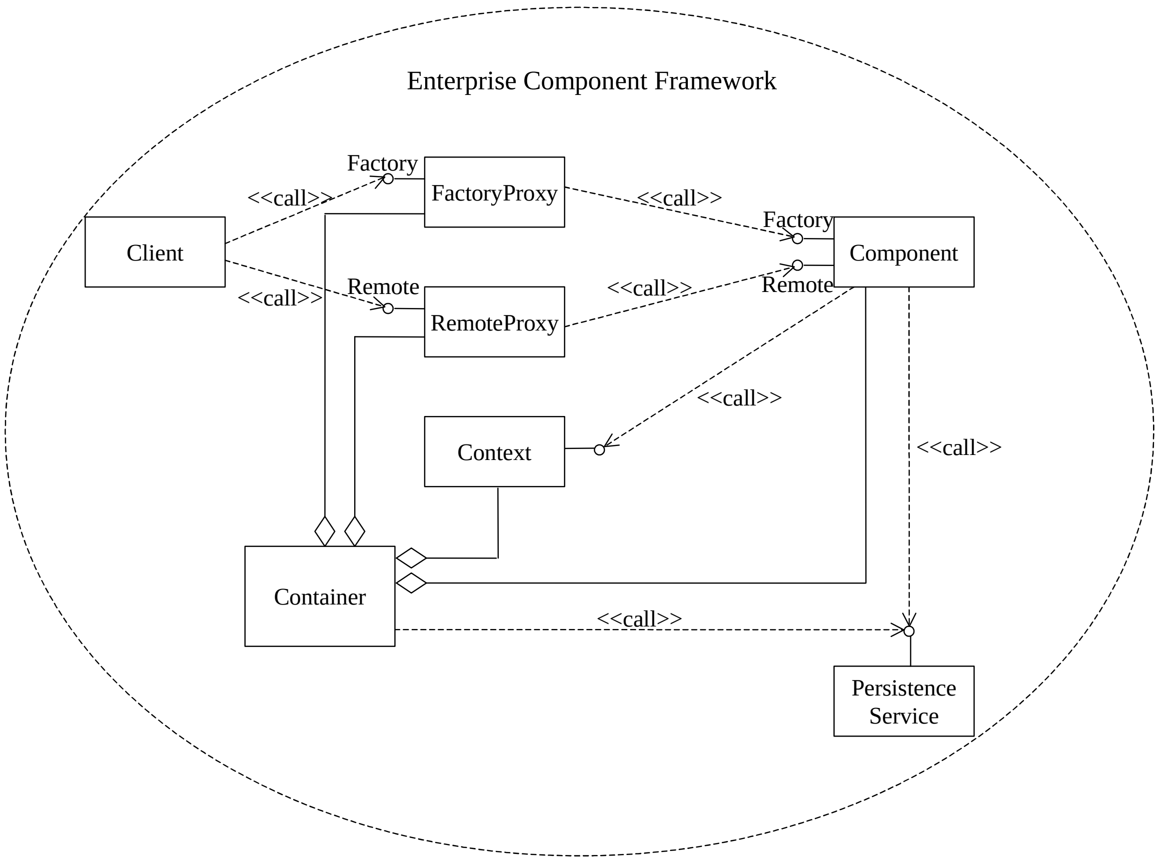

Figure 11.4   Enterprise Component Framework

[Che99] E. Chen, "Poison Java", *IEEE Spectrum*, August 1999, 38-43.

[Cow98] C. Cowan and C. Pu, "Death, taxes, and imperfect software: Surviving the inevitable", *Procs. of the New Security Paradigms Workshop*, 1998.

[Cow00] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade", http://www.cse.ogi.edu/DISC/projects/immunix

[Edd99]   G. Eddon, "The COM+ security model gets you out of the security programming business", *Microsoft Systems Journal*, vol. 14, No 11, November 1999, 35-50.

[Fer94] E. B. Fernandez, E. Gudes, and H. Song, ``A model for evaluation and administration of security in object-oriented databases", *IEEE Trans. on Knowledge and Database Eng.*, vol. 6, no. 2, April 1994, 275--292.

[Fer03] E.B. Fernandez, M. Thomsen, and M.H. Fernandez, "Comparing the security architectures of Sun ONE and Microsoft .NET", to appear as Chapter IX in *Information security policies and actions in modern integrated systems,* C. Bellettini and M.G. Fugini (Eds.), Idea Group Publishing, 2003.

[Fra99] G. Frascadore, "Java application server security using capabilities", *Java Report,* March 1999, 31-42.

[Har97a] B. Hartman, "Java security: Is it safe?", *Dist. Object Computing*, Nov./Dec. 1997, 57-61.

[Har99] B. Hartman, "Enterprise Java Beans security", *Dist. Object Computing,* Jan./Feb. 1999, 30-34.

[ibm] IBM Corp., IBM WebSphere Business Components, http://www-3.ibm.com/software/webservers/components

[Kob00] C. Kobryn, "Modeling components and frameworks with UML", *Comm of the ACM*, October 2000, 31-38.

[Kov98] L. Koved, A.J.Nadalin, D. Deal. And T. Lawson, "The evolution of Java security", *IBM Systems Journal*, vol. 37, No 3, 1998, 349-364.

[Kov01] L. Koved, A. Nadalin, N. Nagarathan, M. Pistoia, and T. Shrader, "Security challenges for Enterprise Java in an e-business environment", *IBM Systems Journal*, vol. 40, No 1, 2001, 130-152.

[Low01] J. Lowy, *COM+ (and .NET) make security a joy*, talk at Software Development West, April 2001.

[McE83] G.H. McEwen, "The design of a secure system based on program analysis", *IEEE Trans. on Software Eng.*, vol. SE-9, No 3, May 1983, 289-299.

[McG97] G. McGraw and E. Felten, "Java security: Dealing with hostile applets", *Java Report,* February 1997, 39-46.

[Oak01] S. Oaks, *Java security*, 2nd Ed., O'Reilly, 2001.

[Sum75] R. C. Summers, C. D. Coleman, and E. B. Fernandez, ``A programming Language Extension for Access to a Shared Data Base," *Proceedings of the 1975 ACM Pacific Conference*, pp. 114-118, 1975.

[Szy98] C. Szyperski, *Component software—Beyond object-oriented programming*, Addison-Wesley 1998.

[Yod97] J. Yoder and J. Barcalow, "Architectural patterns for enabling application security". *Procs. PLOP'97*, http://jerry.cs.uiuc.edu/~plop/plop97 Also, Chapter 15 in *Pattern Languages of Program Design*, vol. 4 (N. Harrison, B. Foote, and H. Rohnert, Eds.), Addison-Wesley, 2000.

[Zho98] Q. Zhong and N. Edwards, "Security control for COTS components", *Computer*, IEEE, June 1998, 67-73.