Language-Based Security

Lecture Notes on Software Security

Erik Poll
Digital Security group
Radboud University Nijmegen

Updated August 2023

These lecture notes discuss *language-based security*, which is the term loosely used for the collection of features and mechanisms that a programming language can provide to help in building secure applications.

These features include: *memory safety* and *type safety*, as offered by so-called *'safe' programming languages*; language mechanisms to enforce various forms of access control (such as *sandboxing*), similar to access control provided by operating systems; and mechanisms that go beyond what conventional access control can do, namely *information flow control*.

Contents

Chapter 1

Introduction

Most security problems in IT systems are caused by software: ‘hacking’ a system comes down to exploiting software bugs or sometimes exploiting software features, as some features turn out to have unintended consequences. Sadly, as a rule of thumb, if a system contains software, it is very likely that it can be hacked.

The only other cause of security problems that can rival software in importance is *the human factor*. Instead of attacking the system by exploiting weaknesses in the software, attackers can also resort to *social engineering* to exploit weaknesses of users, sys-admins, or indeed programmers. Phishing has long been and remains the most popular attack vector. However, in many cases where security problems are blamed on users it is really the software that is to blame, by providing bad user interfaces. Pointing the finger to ‘stupid users’ is often a case of victim blaming: it is usually more justified to point the finger to the stupid system architects or incompetent software designers for overly complex and counterintuitive user interfaces, which make it too hard for normal users – or even experienced system administrators – to make the right security decisions.

Insecure software can be written in *any* programming language. Still, the programming language can make a big difference here, by the language features and APIs it provides and by the dangerous features it includes or omits. The programming languages can also offer support to programmers in the form of type checkers, IDEs, compilers and run-time execution engines (aka interpreters or virtual machines). All this can enforce some basic safety properties and make it easier to ensure more specific security properties. Another way in which the programming language makes a difference is that some programming languages will be more amenable to forms of static or dynamic analysis than others. All this is what programming language-based security is about.

The prime example of how the programming language can have a negative impact on security is **memory corruption**. Memory corruption bugs can happen if the programming language does not check array bounds, allows pointer arithmetic, or makes programmers responsible for memory management. Unfortunately some of the most popular programming languages around, notably C and C++, suffer from this. In so-called ‘safe’ programming languages this category of security flaws can be entirely ruled out. Besides **memory safety** there are other forms of safety that a programming language can provide of safety, notably **type safety**. Here different type systems offer different levels of expressivity and different levels of assurance. Memory safety, type safety and other forms of programming language safety are discussed in Chapter 3.

Programming languages differ not only in the safety guarantees they provide, but also in features and building blocks for implementing security functionality. One such feature is **sand-**

boxing, provided by Java and C#, which allows **access control** to be enforced on untrusted – or less trusted – code. Sandboxing is discussed in Chapter 4.

The notion of **information flow**, explored in Chapter 5, provides another way to specify and enforce security policies of programs, which goes beyond more traditional forms of access control or type systems can achieve.

There are many more forms of program analysis than discussed in these lecture notes that can help to combat certain categories of security flaws. *Static analysis tools* aka *source code analysers* do such analyses at or before compile time, i.e. statically. Such analysis tools can exists for any programming language, but some programming languages will be easier to analyse than others. The analyses can be integrated into IDEs (Integrated Development Environments) or compilers, though compilers typically perform static analysis in order to produce more efficient code, so efficiency rather than security is the primary motivation here. The term **SAST** (Static Application Security Testing) is sometimes used for static analysis tools specifically aimed at looking for security flaws. The term **DAST** (Dynamic Application Security Testing) is then used for dynamic analysis tools for security, i.e. for *security testing* tools[1].

Powerful forms of program analysis that go beyond type checking include **symbolic execution**, where a program is executed with symbolic values for inputs instead of concrete values [9], and **program verification**, where mathematical proofs are used to guarantee properties of a program, which can then include very expressive properties. Such mathematical proofs can even be embedded into the program code, resulting in *Proof-Carrying Code (PCC)* [26].

Overview

Chapter 2 briefly reviews the main security mechanisms provided by the operating system, namely (i) *compartmentalisation* and *access control* built around the notion of process, and, more fundamentally, (ii) *abstractions* (including the notion of process) on which programming languages have to build. Chapter 3 then looks at how *'safe' programming languages* can support writing secure programs. Chapter 4 discusses *language-based access control*, where the programming language provides a sandboxing mechanism for more fine-grained access control where different parts of a program are subject to different security policies. Chapter 5 explores the notion of *information flow*, which allows even more fine-grained access control, taking into account the source where data comes from and the destinations (or sinks) where it may flow to.

[1] The term 'testing' is normally used to refer to dynamic techniques, so purists are right to complain that the 'T' in SAST is incorrect and the 'D' in DAST is superfluous.

Implementation bugs vs design flaws

When discussing security vulnerabilities in software, it is common to distinguish **implementation bugs** and **design flaws**. Implementation bugs, also called **coding defects**, are low-level implementation defects in the code – effectively, mistakes introduced by the programmer who wrote the code. Design flaws are more fundamental problems, that exist prior to any line of code being produced and irrespective of the precise way it has been implemented.

In practice, it is hard to draw a precise border between these categories. For example, broken access control in an implementation may be due to a simple programming mistake in the coding phase that resulted in an incorrect or missing access control check. But the way in which access control is organised in an application – and possibly supported by the programming language – can make such bugs more or less likely. Having access control decisions scattered throughout the code in an ad-hoc matter makes problems much more likely. This is more a design flaw. Access control problems can also be introduced at a very high design level, by completely overlooking the need for any access control.

Secure functionality vs security functionality

Beware of the difference between *secure* functionality and *security* functionality. Security functionality provides security-related features such as access control, authentication, logging, encryption, etc. Some applications, e.g. VPN solutions, firewalls, or anti-virus solutions, have it as their sole aim to provide security functionality. But *any* functionality provides by *any* piece of software may contain security vulnerabilities that an attacker can exploit: so not only security functionality needs to be correct, but *all* functionality needs to be secure.

In the end, security should be understood as an emergent property of an application as a whole, and not as some collection of features. Beware of anyone who claims code is secure 'because' of some specific feature or technology, e.g. 'because it is written in programming language X' 'because it uses sandboxing', or – the most common mistake – 'because it uses encryption'. When it comes to software security – or indeed, security in general – there are no silver bullets.

Common security vulnerabilities

There are various lists of common security vulnerabilities in software, for example

- the OWASP TOP 10 (`https://owasp.org/Top10`);
- the book 'The 24 Deadly Sins of Software Security' [19], which includes a nice systematic overview of which programs/programming languages may be affected and how each sin can be spotted in a code review or by testing;
- the CWE/SANS TOP 25 Most Dangerous Software Errors (`http://cwe.mitre.org/top25`).

The 2021 edition of the OWASP Top Ten has become a somewhat strange mixture of extremely generic issues, such as 'Insecure Design', and very specific ones, such as 'Server-Side Request Forgery (SSRF)', There are actually several OWASP Top Tens: apart from the well-known Top Ten of Web Application Security Risks, there is also a Mobile Top Ten[a] and a Kubernetes Top Ten[b].

The Top Ten Software Security Design Flaws [5], not by OWASP, lists common higher level design flaws, as opposed to more concrete coding mistakes. OWASP's Top Ten of Proactive Controls[c] does something similar.

Checklists like the ones above are very useful, but don't ever be tempted into thinking that such lists are complete and cover everything that can go wrong. The categorisation of security flaws attempted in MITRE's CWE (the Common Weakness Enumeration, a companion to the better-known CVE list of concrete instances of security flaws) has grown to around a thousand bug categories. This is clearly way more than anyone can remember or use in a practical way as a checklist. The full CWE list is available as one giant PDF format[d] and the CWE website also provides various visualisations and taxonomies for parts of it[e]. Browsing through the CWE list is useful to do at least once, not to memorise parts of it, but simply to get a sense of its size and complexity.

[a] `https://owasp.org/www-project-mobile-top-10`

[b] `https://owasp.org/www-project-kubernetes-top-ten`

[c] `https://owasp.org/www-project-proactive-controls`

[d] `https://cwe.mitre.org/data/published/cwe_latest.pdf`

[e] For example, `https://cwe.mitre.org/data/pdf/1000_with_1344_colors.pdf`

Chapter 2

Operating system based security

Before we consider what role the programming language can play in providing security, it is useful to take a step back and consider the role an operating system (OS) traditionally plays in providing security, and look at

- the security mechanisms that operating systems traditionally provide, through separation and various forms of access control;
- what the basis for this access control is;
- the role that abstractions play here.

Note that as the execution engine for a programming language, such as Microsoft .NET for C#, F# and Visual Basic, provides more (security) functionality, it begins to look more and more like an operating system.

2.1 Operating system abstractions and access control

The operating system provides an **interface** between the hardware and programs executing on this hardware, and also an interface to the human user (see figure 2.1). Here the hardware

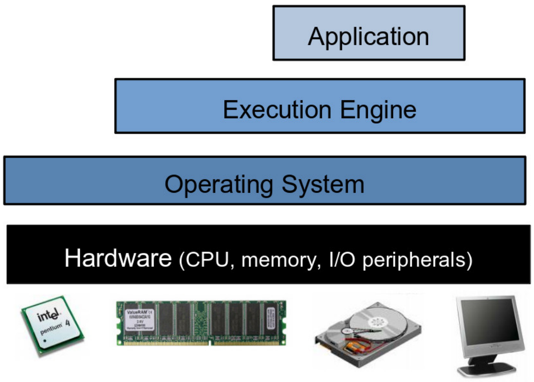


Figure 2.1: The infrastructure on which programs are built.

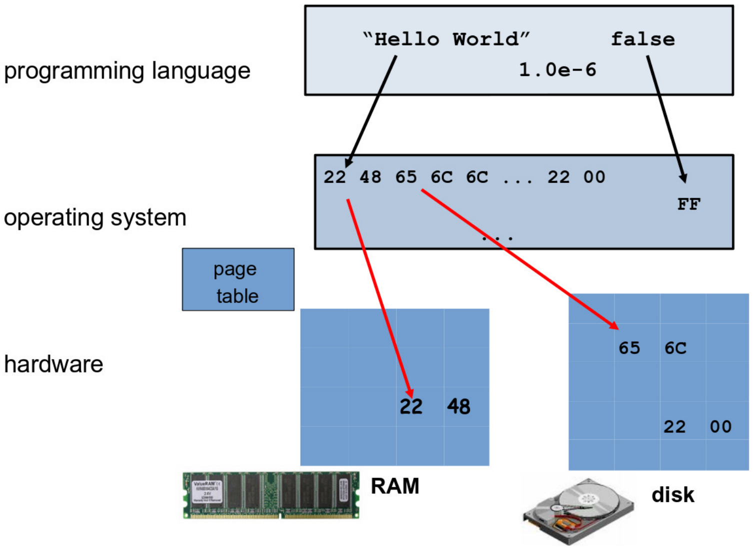


Figure 2.2: The different abstraction layers provided by programming language and operating system.

includes the CPU and the memory, plus a range of other I/O devices, typically including a hard disk, a keyboard, a display, and a network connection.

The operating system provides useful **abstractions** to hide a lot of the underlying complexity. For example, the operating system uses segmentation and page tables to provide virtual memory, and programs can then use a virtual address space for storing data without having to worry about where in memory or where on the hard disk the data actually resides (see figure 2.2). Another example is that the operating system provides a file system to programs and users, which hides the details of how the files are stored over various sectors of the hard disk.

The central abstraction that the operating system provides is that of a **process**. Each program in execution gives rise to a process, and the operating system tries to create the illusion that this process has exclusive access to the CPU (through time slicing) and to the computer's memory, and typically to far more memory than is physically available as RAM (through segmentation and paging).

As part of this abstraction, the operating system tries to ensure **separation** between the different processes, so that programs cannot interfere with each other. The prime example here is that processes have their own **address space** that others cannot touch, which is obviously crucial for security. A process that tries to access memory outside its address space will crash with a so-called *segmentation fault*. This way the operating system provides some **encapsulation** of processes. Processes can still interfere with each other when competing for the shared resources of the computer: e.g., a program that claims a lot of memory or a lot of CPU time could interfere with the progress that another process can make.

The operating system also provides security by enforcing **access control** for many of the abstractions it introduces[1]. The prime example here is the file system, where different users – and the programs they start – have different access rights for reading or writing certain files. Traditionally, access control provided by the operating system is organised by user and by process, where processes typically inherit the access rights of the user that started them, although

[1] Of course, the memory separation between processes can also be regarded as a form of access control.

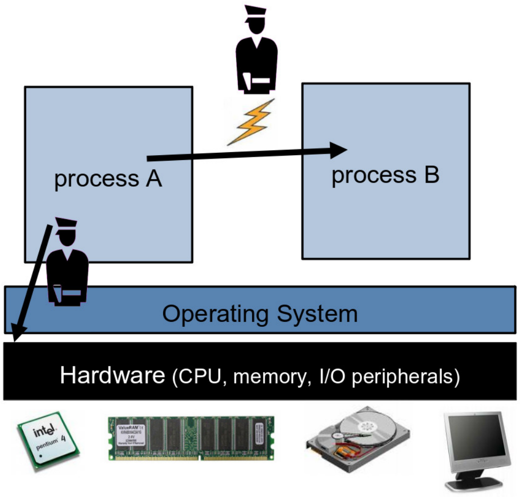


Figure 2.3: Classical operating system security controls: separation between processes and access control of system resources.

modern operating systems will offer some ways to tweak this.

Figure 2.3 illustrates the security mechanisms discussed above.

The abstractions provided by the operating system play a crucial role in any access control for them: if someone gets access to the raw bits on the hard drive, any access control provided by the file system is gone. Access to the data that is used to provide these abstractions needs to be tightly controlled: if someone can modify page tables, all guarantees about memory separation between processes by the operating system can be broken. This means that the code responsible for creating and maintaining these abstractions has a special responsibility when it comes to security. Such code needs special privileges, to get low-level access that other code does not have. Any bugs in this code can lead to serious security vulnerabilities; this code has to be *trusted*, and hopefully it is trustworthy. This has led to the distinction between **user mode** and **kernel mode**, which most operating systems provide. Part of the code that belongs to the operating system is executed in kernel mode, giving it complete and unrestricted access to the underlying hardware. So the access rights of a process do not only depend on the user that started that process, but also on whether the process is executing kernel or user code, which may vary in time.

2.2 Imperfections in abstractions

The abstractions that the operating system provides – and the illusion of each process having exclusively access to its own CPU and memory that it tries to provide – are not perfect. Such imperfections may lead to security vulnerabilities.

For example, a process may be able to detect that some of its memory is in fact swapped out to disk, based on the response time of certain operations. This property was used to break password security of the TENEX, an operating system developed in the early 1970s: depending on the response time of some incorrect password guess, it was possible to work out if the first n characters of the password guess were correct, by making sure only these first n characters were in main memory, and the rest were swapped to disk.

Another example is that by observing the contents of freshly allocated memory, a process can observe the contents of memory that used to belong to other processes. This is a classic way of snooping on other users in multi-user systems.

An example given in [17] shows the unexpected ways in which these imperfections can lead to security problems. In 1993 it was observed that every tarball[2] produced on UNIX Solaris 2.0 contained a fragment of the password file – `/etc/passwd` on UNIX systems – at the very end. The way that this happened was as follows:

1. To find out the owner and group permissions for the tar file to be created, the password file was consulted. (The password file tells which group a user belongs to.) This meant this file was read from disk, and stored in RAM. This RAM was then released.
2. Then the tar file was created. For this RAM memory was allocated. Because of the way memory allocation worked, this memory included the memory that had just before been used for the password file. And because memory is not wiped on allocation or de-allocation, this memory still contained contents of the password file.
3. The size of a tar file is always a multiple of a fixed block size. Unless the actual contents size is precisely a multiple of this block size, the block at the end will not be completely filled. The remainder of this block still contained data from the password file.

As tar files are typically used to share files over the internet the security implications are clear. Fortunately, in Solaris 2.0 the file `/etc/passwd` no longer contained hashed and salted passwords, these were in a separate file (the shadow password file).

A quick fix to the problem is replacing a call to `malloc` (which allocates memory) in the code of `tar` by a call to `calloc` (which allocates memory and zeroes it out). That only fixes this particular instance of the problem, and people always use `malloc` by default rather than `calloc` because it is faster. One can think about more fundamental solutions to this kind of problem, e.g., always zeroing out memory on allocation, or always zeroing out memory upon de-allocation if the memory contained sensitive data.

2.3 What goes wrong

The way that security vulnerabilities normally arise in operating systems is more mundane than the tarball example above. It typically happens due to

- **Software bugs**

  Especially memory corruption in high privilege library calls provided by the operating system are notorious source of problems, which then typically lead to **privilege escalation**.

  Modern operating systems come with large APIs for applications to use. Many of the system calls in this interface run with very high privileges. The standard example is the `login` function: `login` is invoked by a user who has not been authenticated yet, and who should therefore not be trusted and only be given minimal permission; however, the `login` procedure needs access to the password file to check if the given password guess is correct, so needs very high privileges. A buffer overflow weakness in `login` can possibly be exploited by an attacker to do other things with these privileges.

[2] A tarball is a file produced by the `tar` utility for archiving and compressing directories, similar to `zip`.

- **Complexity**

  Even if the operating system API was implemented without any bugs, the sheer complexity of operating systems means that users will get things wrong or run into unforeseen and unwanted *feature interaction*, where interplay of various options introduces security vulnerabilities.

  Textbooks on operating systems typically illustrate the idea of access control with read, write, and execute permissions, for users that can be organised in groups, and with one super-user or root who has permission to do anything. But real-life modern operating systems offer dozens of permissions and all sorts of user groups with different privilege levels. The advantage is that this enables very fine-grained access control. The disadvantage is that people configure it wrong and grant too many access rights. Govindavajhala and Appel provide an interesting account of how major software vendors got Windows XP access control wrong in the early 2000s [16], which then let any user configure an arbitrary executable to be used for some high-privilege system services[3]. Fifteen years later almost identical problems still crop up, for instance with CVE-2021-34481 that affected most versions of Windows[4].

  Note that there is a fundamental tension between the **Principle of Least Privilege**, and the more general **KISS principle**, i.e. *Keep it Simple, Stupid*. The Principle of Least Privilege says that only the minimal access rights needed to do some task should be given. The more fine-grained access control is, the better this principle can be followed. But the downside is that more fine-grained access control becomes more complex, making it more likely that people make mistakes. In line with the KISS principle, access control is easier to get right if it involves large-coarse chunks of all-or-nothing access, which is also known as **compartmentalisation**.

  One can try to combine these design principles by including several layers of access control, in line with the design principle of **Defence in Depth**, where one layer provides simple coarse compartments and another more fine-grained access control within such a compartment.

  Classic reading material on security design principles such as Least Privilege and Defence in Depth is the seminal article by Saltzer and Schroeder [29].

[3] See also `https://www.kb.cert.org/vuls/id/953860`.

[4] See `https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-34481`.

Chapter 3

Safe programming languages

A fundamental way in which a programming language can help in writing secure programs is by being *'safe'*. This chapter investigates what safety means here, and discusses the various flavours of safety, such as memory safety, type safety, and thread safety.

Precisely pinning down what safety means is tricky, and people have different opinions on precisely what constitutes a safe programming language. Usually, by a safe programming language people mean one that provides memory safety and type safety, but there are other forms of safety, as we will see. Safe programming languages provide some guarantees about the possible behaviour of programs, which can protect against security vulnerabilities due to unwanted behaviour. In essence, the central idea is:

> In a safe programming language, you can trust the **abstractions** provided by the language.

An alternative characterisation, which is a bit more concrete, is:

> In a safe programming language, programs always have a **precise and well-defined semantics**.

An important goal of safety is that safety makes it possible to reason about programs in a modular way: i.e., to make it possible to make guarantees about the behaviour of certain 'parts' of a program without knowing the rest of it. Here, depending on the programming languages, these 'parts' can be functions and procedures, objects, classes, packages, or some other notion of module.

> In a safe programming language, it is possible to understand the behaviour of a program in a **modular** way, and to make guarantees about part of a program by inspecting only that part and not the entire program.

The following sections will explain these general ideas in more detail, and give more specific examples. We begin by considering the issue of undefinedness, before turning to the more specific notions of memory safety and type safety.

Undefinedness & fuzzing

Leaving the meaning of some language constructs undefined under some circumstances is a potential cause of security problems. More generally, leaving things undefined in the specification of any system is a potential cause of trouble. Indeed, a standard technique to find security loopholes is to try out how a system behaves under the circumstances where the specification does not say how it should. The system might just crash, which might cause a Denial-of-Service (DoS) vulnerability (though under some circumstances crashing might be the safe thing to do). But you may discover all sorts of surprising – and possible insecure – behaviour.

The test technique of **fuzzing** is based on this idea. In fuzzing, software is tested by sending invalid, unexpected, or simply random inputs to an application in the hope of detecting security or robustness problems. Fuzzing with very large input strings can be used to hunt for buffer overflows: if an application crashes with a segmentation fault on some very long, but completely random, input string, you have found a potential security vulnerability.

Fuzzing is a very effective technique, especially if a program accepts some complex input format, for instance a complex file format (say, HTML, PDF, .docx, JPEG, MPEG, MP3, X.509, . . .) or a complex protocol anywhere in the protocol stack (say, TCP/IP, HTTP, SMB/CIFS aka Samba, Bluetooth, Wifi, GSM, 4G/LTE, 5G, . . .).

Note that just about any program supports some complex input language or format, and often more than one. The complexity of these input languages is an important underlying root cause – if not the ultimate root cause – of security vulnerabilities. To make matters worse, there are many of these complex input languages, protocols, and formats and their specifications are often unclear. All this results in bugs in the (typically hand-written) code for parsing and processing these languages, bugs that go on to cause security problems. These security problems are by their very nature easy to trigger for an attacker, as they arise in code responsible for input handling. This observation had led to *LangSec* (Language-theoretic Security) approach (`http://langsec.org`, [27, 8]) that tries improve security by addressing these root causes.

3.1 Safety & (un)definedness

Many program statements only make sense under certain circumstances. For example, the statement

```
a[i] = (float) x;
```

only makes sense if in the current execution state `a` is an array in which we can store floats, `i` has an integer value within the array bounds, and `x` can sensibly be converted into a float value. If one of these conditions is not met, then it is unclear what the semantics of this statement should be.

There are roughly two approaches that programming languages can take here:

1. One approach is to accept that a program statement may be executed when it does not really make sense. The semantics of the statement is *undefined* in these cases: essentially, anything may happen.

   It is then the obligation of the programmer to ensure that statements are only ever executed in situations where they make sense.

   What actually will happen in the undefined cases is down to the compiler and the platform responsible for the actual execution. This platform is the raw hardware (i.e., the CPU) in the case the compiler produces native code. It can also be the (software) execution engine used to execute programs in some intermediate language (e.g., a VM executing bytecode) and the hardware on top of which this engine runs.

2. The other approach is that the language ensures that a statement is only ever executed when it makes sense, or, when it does not, signals some error in a precisely defined manner, for example by throwing an exception.

   It is now the obligation of the language to somehow prevent or detect the execution of statements when this does not make sense. This can be through measures at compile-time (e.g., type checking), at run-time (by some execution engine that monitors for these conditions, or by the compiler including some checks in the code it produces), or a combination of the two.

The first approach leads to an **unsafe** programming language. The second approach leads to a **safe** programming language.

In safe languages, the semantics of a program – or any program fragment – is always precisely defined; even in cases where it does not make sense, it is precisely defined what error will be reported.

C and C++ are the prime examples of unsafe languages. Java and C# are meant to be safe, but still have some unsafe features. For instance, the Java Native Interface (JNI) allows Java programs to call native machine code, and pointer arithmetic is allowed in C# in code blocks that are marked as `unsafe`. So Java and C# programs are only safe if these features are not used. Most functional programming languages, such as Haskell, ML, or LISP, are safe, as are most logic programming languages, such as Prolog. There is a clear trend towards safer programming languages, with for instance in JavaScript, Scala, PHP, Python, Ruby, and Go. Even the newer languages aimed at low-level system programming, such as Rust and Swift, are designed with safety in mind, even though such languages typically sacrifice safety for speed.

The main advantage of the unsafe approach is *speed*. Execution of a safe language typically involves some run-time monitoring that slows execution down.

An important disadvantage is *security*. If the programmer is not very careful and just one program statement is executed in a state where it does not make sense, we have no way of knowing what may happen: it may re-format the hard disk, send all your private email correspondence to wikileaks, etc. This means we can no longer make any security guarantees about the program.

Conflicts between security and some practical consideration such as speed (or 'convenience') are common. Almost always people sacrifice security in favour of the short-term practical advantage, and then live to regret the consequences in the long term. Four decades after the introduction of C – and the decision not to do bounds checking in C – people are still fixing buffer overflows in C programs on a daily basis, even though ALGOL 60 introduced bounds checking as mechanism to prevent this problem over half a century ago.

3.1.1 Safety and security

It is possible to write insecure programs in any programming language. But without safety, writing secure programs gets a lot harder.

Consider a procedure `login(uname,passwd)` written in some unsafe programming language. Only if this procedure is very carefully written to check for any of the error conditions that would lead to undefined behaviour, we can make some guarantees about it (for instance to return true iff it is given a matching username and password).

If not, then we need to know the precise circumstances in which the procedure is called in order to make any guarantees. In the best case, this is a lot of work: we have to check all the places in the code where `login` is called, and check that these calls cannot trigger any undefined behaviour. When doing this, we should be careful not to make any assumptions on user input that the program may receive. For non-trivial programs this quickly becomes infeasible to check.

Note that this reveals a fundamental problem with unsafe languages, apart from any security issues: we cannot always understand code in a **modular** fashion. E.g., we may not be able to know what the procedure `login` will do without knowing the precise context in which it is called.

Things get worse if the procedure is part of some library that may be called from many places, or if `login` is an operating system call that may be called by any application program. In these cases we do not know the specific circumstances of calls to `login`, so we cannot rely on these to avoid running into undefined behaviour.

3.2 Memory safety

One important form of safety is memory safety. A programming language is **memory-safe** if programs are guaranteed to only access memory locations that they are permitted to access. Language features that break memory safety include:

- lack of array bounds checks – this allows the classic **buffer overflows**;
- pointer arithmetic;
- lack of checks for null pointers;
- programmer-controlled memory management, more specifically programmer-controlled de-allocation with e.g. `free()` in C(++), as this allows **dangling pointers** to memory that has been de-allocated, which gives rise to **use-after-free** and **double free** bugs;
- lack of mandatory or default initialisation of newly allocated memory (e.g. the use of `malloc()` rather than `calloc()` in C(++);
- unconstrained casting (e.g., casting a floating-point number to an array).

Bugs that arise because of these issues are commonly called **memory corruption bugs**. Strictly speaking, this term is not quite correct, as only bugs that involve *writing* to memory cause memory corruption; bugs in *reading* from memory do not corrupt memory, so the term **illegal memory access** is more accurate. Illegal memory access can happen because of **spatial errors**, for instance accessing outside array bounds , or **temporal errors**, for instance dereferencing a dangling pointer. The absence of spatial error is sometimes called bounds safety and the absence of temporal errors lifetime safety.

A good overview of how memory corruption attacks and defenses have evolved over the past decades is given in the SoK (Systematisation-of-Knowledge) paper by Szekeres et al. [35].

Obvious ways to prevent memory problems include doing array bounds checks, checking for null pointers, disallowing pointer arithmetic and unsafe casts. Checks on array bounds and checks for (non)nullness can be compiled into the code or performed by a runtime execution engine (aka a virtual machine). In theory such checks could even be performed by the hardware – i.e. a physical as opposed to a virtual machine – but in practice the repertoire of checks that hardware performs will be very limited.

3.2.1 Secure coding guidelines and safe(r) libraries

If you are stuck with using a memory-unsafe language like C and C++, there are things you can do make memory-corruption less likely. Some library functions are more error-prone than others, as are some datatypes. For instance, in C you have the choice between using `strcpy`, `strncpy` – which is consider safer –, and `strlcpy` – which is considered to be safer still. In C++ you have the choice between C-style strings and the safer C++=style strings. If you program in C or C++ you should know and stick to secure coding guidelines (for instance the SEI CERT Coding Standards [2, 3]). That is not to say that if you use a safe programming language you do not have to do this; for these there are also secure coding guidelines (e.g., see [1]).

3.2.2 Sanitisers

Another thing to do when coding in a memory-unsafe programming language is to use **sanitisers**. Sanitisers are compiler extensions or stand-alone tools that instrument code with safety checks to help with testing or debugging. An old tool for this still in use today is valgrind. More modern tools include ASan (AddressSanitizer) and MSan (MemorySanitizer). The SoK paper by Song et al. gives an overview of sanitisers for memory safety [32]. Note that use of sanitisers is not limited to finding and fixing on memory corruption issues; some tools (also) aim to help with concurrency bugs.

The overhead caused by the code instrumentation, both in execution speed and memory usage, can be substantial. So while useful in the testing and debugging phase, in the final production code the instrumentation typically has to be removed because the overhead is not acceptable. If the overhead is small enough then instrumentation could be left for increased safety. For instance, Apple is now leaving instrumentation for spatial safety in some production code [25].

3.2.3 Automated memory management

The typical way to prevent problems with memory management (more specifically programmer-controlled memory de-allocation with `free` statements) is to automate memory management using a **garbage collector (GC)**. The garbage collector then takes care of de-allocation of unused memory automatically. (NB this does not solve all memory management-related issues: it is still possible to have *memory leaks*.) Garbage collection was first used with LISP in the early 60s and then used by functional programming languages and by SmallTalk. It became ‘mainstream’ with its use in Java. Most modern programming languages nowadays rely on garbage collection, e.g. JavaScript, Ruby, Python, PHP, C#, and Go.

A downside of garbage collection is the additional overhead: the process of garbage collection requires memory space and time. Moreover, a program can temporarily become unresponsive when the garbage collector – periodically and at unpredictable moments – kicks in. Especially for real-time applications such behaviour is problematic. However, there are techniques to do real-time garbage collection, where the overhead of garbage collection becomes predictable [30].

There are techniques to automate memory management that do not rely on a garbage collector. These are for instance used in Rust[1], Objective-C and Swift[2]. These languages are intended for low-level system programming where one would like to avoid the overhead of automated garbage collection. Rust uses a notion of object ownership, which allows the compiler to

[1] `https://doc.rust-lang.org/book/`

[2] `https://swift.org`

insert calls to de-allocate object at compile time. Swift and Objective-C use reference counting, more specifically Automatic Reference Counting (ARC), to decide at runtime when object can be de-allocated. Unlike Rust's approach this produces runtime overhead, namely for the book-keeping of reference counts. But Rust's approach requires programmers to be aware of object ownership and the restrictions that imposes, which require learning new programming idioms and patterns.

3.2.4 Memory safety and undefinedness

Memory safety is a special case of the notion of safety as discussed in section 3.1: accessing a memory location that you should not be accessing will not have a well-defined semantics and can break any abstractions the language provides.

Memory unsafety makes it hard – if not impossible – to understand programs in a modular fashion. Consider a program consisting of two modules, P and Q, written in a language that is not memory-safe. Code belonging to module Q could then make changes anywhere in memory, including data belonging to module P. So code in module Q could corrupt data that module P relies on, for instance breaking data invariants. This means we cannot make *any* guarantees about P just by looking at the code of P: we also have to look at the code of Q to check that it not interfere with P, and as memory corruption bugs can be subtle this will be hard. Note that this also means we cannot make guarantees about extensible programs, i.e. programs which can be extended at runtime, in memory-unsafe language, as any future extension could corrupt existing behaviour.

If a language is memory-safe then programs can never crash with a segmentation fault. One might even consider it safe to switch off the checks the operating system performs on memory access for these programs. However, the execution of the memory-safe program might rely on some interpreter or execution engine that is written in an unsafe language, which could still cause out-of-bounds memory access. Also, memory safety might rely on the correctness of compilers and type checkers, which may contain bugs. So, in line with the security principle of **Defence in Depth**, keeping the memory access control performed by the operating system switched on is always a wise thing to do.

3.2.5 A stronger notion of memory safety: guaranteed initialisation

A stronger notion of memory safety also guarantees that programs never read **uninitialised memory**. Earlier we defined memory safety to mean that programs are guaranteed to only access memory locations that they are permitted to access, and one can argue whether accessing uninitialised memory should also count as access that is not 'permitted'.

If programs cannot access uninitialised memory than it becomes impossible for them to spy on the content of other processes as we discussed in Chapter 2. Another consequence – in practice usually a more interesting one – is that programs behave more deterministically: a program that reads uninitialised memory can behave differently every time it is started. This can make programs hard to debug, as errors can be hard to reproduce.

3.3 Type safety

Another important form of safety is type safety. As discussed earlier, safety involves avoiding erroneous conditions under which programs become meaningless. Type systems impose some restrictions on the programmer that avoid meaningless programs. It is therefore not surprising

```
1.    unsigned int tun_chr_poll(struct file *file, poll_table *wait){
2.       struct tun_file *tfile = file->private_data;
3.       struct tun_struct *tun = __tun_get(tfile);
4.       struct sock *sk = tun->sk;
5.       if (!tun) return POLLERR;
         ...
      }
```

Figure 3.1: This piece of code from the Linux kernel is a classic example of how undefinedness, or rather the compiler optimalisations allowed by undefinedness, can unexpectedly cause security problems [38]. In line 4 `tun` is de-referenced: If `tun` is NULL, this leads to undefined behaviour. Line 5 returns with an error message if `tun` is NULL. However, because the behaviour of the code is undefined if `tun` is NULL, the compiled code may exhibit *any behaviour* in that case. So a compiler may choose to remove line 5: after all, any behaviour is acceptable if `tun` is NULL, so there is no need to return with the error specified in line 5 in that case. Modern compilers such as gcc and Clang will actually remove line 5 as an optimisation. Removal of line 5 from the Linux kernel led to a security vulnerability (CVE-2009-1897). With the compiler flag `-fno-strict-overflow` such optimisations can now be turned off in gcc.

that many safe languages are typed. However, not all memory-safe languages are typed. For example, LISP is a safe, but untyped language.

Type soundness, also called **type safety** or **strong typing**, is the guarantee that executing well-typed programs can never result in type errors. This means that in a type-safe language we can rely on the abstractions offered by the type system.

C and C++ are examples of languages that are typed, but not type-safe. Here programs can easily undermine any guarantees of the type system. Java, C#, Python, Rust, Go, Swift, Kotlin, ML, and Haskell are type-safe. TypeScript is a strongly typed version of JavaScript. JavaScript offer some dynamic type safety, where runtime checks can halt execution with TypeError warnings, but the really weird JavaScript behaviour shown in Gary Bernhardt's excellent presentation on JavaScript[3] raises serious questions about what this form of type safety effectively means, as the JavaScript execution engine will happily execute nonsensical code like `s = "some string" * 4;`.

Type checking is the process of verifying that a program is well typed. This can be done at compile-time, in which case we talk of static type checking, or at run-time, in which case we talk of dynamic type checking. Static type checking has the obvious advantage of catching problems earlier. It also offers advantage in speed: if we can rule out some type errors statically, we can omit run-time checks, making programs run faster. (Of course, only in a type-safe language it is safe to do this.)

Most type-safe languages rely on a combination of static and dynamic checks. Typically, array bounds and some cast operations will be checked at run-time.

It is sometimes claimed that "well-typed programs cannot go wrong", but this is of course wildly overstated. At best well-typed programs are *less likely* to go wrong, and only if the type system is sound we can guarantee that *certain types things* cannot happen.

[3] https://www.destroyallsoftware.com/talks/wat

3.3.1 Expressivity

Many programming languages use type systems that have roughly similar expressivity and have a similar division between checks done at compile-time and checks done at run-time. However, there are important choices can be made.

For example, most imperative programming languages check if references are null at run-time. Their types allow null values and problems with accessing null references are only detected at run-time. But some or even all problems with null references can be detected at compile time if we use a more expressive type systems. Indeed, most functional programming languages do this.

For this we have to distinguish between types that allow null references and those that don't. E.g., the Spec# dialect of C# [6] distinguishes between class types `A` and `A?` for all classes `A`, where only values of `A?` are allowed to be null. Values of type `A` can be implicitly promoted to type `A?`, but using a value of type `A?` where a non-value of type `A` is expected requires an explicit cast, which will result in a check for nullness at run-time. Of course, this does introduce some extra work for programmer: they have to specify which references are and which are not allowed to be null. But note that a good programmer has to think about this anyway, so we are only providing notation to document this in the code – notation which is backed up by tool support in the compiler and type checker. In casting from `A?` to `A` null-pointer exceptions can of course still arise, but these can then be limited to specific – and hopefully few – program points. Kotlin is an example of a mainstream imperative programming language that allows the distinction of nullable and non-null types.

3.3.2 Breaking type safety

Safe type systems can provide strong guarantees about data, but type safety is a very fragile property. A tiny loophole in a type system can be fatal – and usually is, undermining the whole type system and destroying all the guarantees that it meant to provide. A loophole can be a flaw in the type system, in which case the type system is not safe, or a bug in the implementation of the type checker.

To exploit a loophole in a type system, one usually tries to create **type confusion**, where one creates references to the same location in memory which the type system thinks have different, and incompatible, types. Here it is important to realise that in the end all values, irrespective of their types, are just blobs of memory, sequences of bytes or words in RAM or on disk. And any blob of memory of the right length can be interpreted as a value of any type.

For an example, in some object-oriented language, suppose we have types `A` and `B`,

```
class A { int i };
class B { Object o };
```

and suppose we have variables `a` and `b` of type `A` and `B`, respectively. If through some flaw in the type system we can get `a` and `b` to point to the same location, then by changing the value of `a.i` we can change the value of `b.o`; effectively this means we can do pointer arithmetic.

If we can do the same with a reference of type `C`, where

```
class C { final int x };
```

then by changing the value of `a.i` we can change `c.x`, even though the type `C` indicates that the field `x` is final and should therefore be unchangeable. So guarantees that a language provides (e.g., those discussed later in Section 3.5) can also be broken by exploiting loopholes in the type system.

An classic example of a loophole in the implementation of the type checker of Navigator 3.0 was discovered by Drew Dean. Here type confusion could be created by defining a class with the name `A` and a class with the name `A[]`. The latter should not be allowed, as `[` and `]` are illegal characters to use in class names! The type checker then did not distinguish between arrays of `A` and objects of type `A[]`, providing an easy loophole to exploit[4].

In some implementations of Java Card, a dialect of Java for programming smartcards, bugs have also been found where exploiting some strange interactions between the normal object allocation mechanism of Java(Card) and smartcard-specific mechanisms in the Java Card API could lead to type confusion [22].

3.3.3 Ensuring type safety

To prevent fundamental flaws in a type system one can try to mathematically prove the correctness of type systems. This can be done as a pencil-and-paper exercise, typically for an interesting subset of the language which captures the essential features. It can also be done using a theorem prover, a software program that assists in logical reasoning in the same way as Maple or Mathematica assist in mathematical analysis. This provides a higher degree of confidence, especially as larger subsets of a real programming language are considered. For example, Java has 205 bytecode instructions; in a pencil-and-paper proof with 205 cases it is easy to overlook a mistake. Safety of various subsets of the Java language has been formally verified using such theorem provers (e.g., see [20]).

It is not so easy to define what it means for a type system to be safe, be it informally or formally. A formal definition is needed for any mathematical correctness proof.

One way is to define an **operational semantics** of the language in two ways: a 'typed (defensive) semantics' where all values are explicitly tagged with type information and where run-time checks are performed for every execution step, and an 'untyped semantics' which only performs those type checks which would normally happen at run-time (e.g., for array indices, nullness, or type casts), and then prove that for any program that passes the type checker the two semantics are equivalent.

Another way is to prove **representation independence**. The idea behind this is that a type system is safe if the abstractions it provides cannot be broken. For example, in a type-safe language it should be impossible to find out if the boolean value `true` is represented as 1 (or maybe a 32- or 64-bits word ending with a 1) and `false` as 0, or the other way around. If we define two semantics of a type-safe language, one where `true` is represented as 1 and `false` as 0 and one where `true` is represented as 0 and `false` as 1, we should be able to prove that for any well-typed program the semantics are equivalent regardless of which of these two representations is used.

3.4 Other notions of safety

When people talk about safe programming languages, they usually mean memory safety and type safety as in modern object-oriented languages such as Java. Still, there are other notions of safety than the ones discussed so far. There is no black-and-white distinction between safe and unsafe. Languages can be more or less safe to different degrees, and in different respects.

[4] This attack is discussed in Section 5.7 in [21].

3.4.1 Safety concerns in low-level languages

For lower-level language there are the additional safety notions about issues we take for granted in higher-level languages, such as

- **Control-flow safety**, the guarantee that programs only transfer control to correct program points. Programs should not jump to some random location.
- **Stack safety**, the guarantee that the run-time stack is preserved across procedure calls, i.e., that procedures only make changes at the top frame of the stack, and leave lower frames on the stack, of the callers, untouched.

For high-level languages one typically takes such guarantees for granted. E.g., in a high-level language it is usually only possible to jump to entry points of functions and procedures, and when these are completed execution will return to the point where they were called, providing control-flow safety. Of course, such a guarantee is fundamental to being able to understand how programs can behave. Only if the compiler is buggy could such safety properties be broken. However, an attacker may be able to break such guarantees: for instance *ROP (Return-Oriented Programming)* attacks [28] exploit the possibility to jump to arbitrary points in the code.

For lower-level languages one might also consider whether procedures (or subroutines) are always called with the right number of arguments. A language like C is not quite safe in this respect: a C compiler typically will complain if a procedure is called with the wrong number of arguments, but *format string attacks* demonstrate that the compiler does not catch all problems here and that it is possible for compiled code to call procedures with too few arguments, resulting in weird behaviour that can be exploitable for an attacker.

3.4.2 Safe arithmetic

Even for something as basic as arithmetic questions about safety arise. For example, consider the statement

```
i = i+1;
```

This statement makes sense if `i` is an integer, but what if computing `i+1` results in an integer overflow? Does this make 'sense'?

In C, a signed integer overflow is undefined behaviour, so C is not safe in this respect[5]. In Java the semantics is precisely defined: overflows behave the way one would expect in two's complement arithmetic (so adding 1 to $2^{32} - 1$ becomes -2^{32}), and such overflows are silent, meaning that execution continues and no exception is thrown. C# goes one step further: a code block or an expression can be marked as `checked`, in which case integer overflow will result in a run-time exception.

In some situations – say, in software for a bank – integer overflow can be a serious concern. One could consider a language where integer overflow results in an exception to be safer than a language with silent integer overflow, even if both are safe in the sense that the semantics is precisely defined.

By the way, integer overflows are a well-known cause of security vulnerabilities, usually in combination with buffer overflows: here the integer overflow leads to the incorrect buffer size.

One of the more expensive software failures ever was due to a problem with arithmetic. The Ariane V rocket was lost due to a software error, where a large numeric value caused an

[5] One might think that 'undefined behaviour' here just means that `i` ends up with an unknown value, but the official C standard, C99, literally allows any behaviour when integer overflow happens, including say reformatting the hard drive.

```
1.    char *buf = ...;       // pointer to start of a buffer
2.    char *buf_end = ...;   // pointer to the end of a buffer
3.    unsigned int len = ...;
4.    if (buf + len >= buf_end)
5.              return; /* len too large */
6.    if (buf + len < buf)
7               return; /* overflow, buf+len wrapped around */
8.    /* write to buf[0..len-1] */
```

Figure 3.2: A classic example of how undefined behaviour caused by integer overflow allows compilers to introduce security vulnerabilities. The code above performs a standard check to see if is safe to write in range `buf[0..len-1]`. In line 6 the programmer assumes that if the integer expression `buf + len` overflows this will produce a negative number. However, integer overflow results in undefined behaviour according to the C standard []. This means that a compiler may assume that `x + y < x` can never be true for a positive number `y`; In case an overflow happens the expression might evaluate to true, but in that case the compiled code is allowed to exhibit any behaviour anyway so the compiler is free to do anything it wants. Modern compilers such as gcc and Clang do conclude that an overflow check such as `buf + len < buf` cannot be true for an unsigned and hence positive integer `len`, so they will remove line 6 and 7 as an optimisation.

overflow that triggered a (hardware) exception – an exception that was not caught and which first crashed the program, and next the rocket. In this case ignoring the overflow would have been better, especially since – ironically – the piece of software causing the problem was not doing anything meaningful after the launch of the rocket. (The report by the Flight 501 Enquiry Board makes interesting reading, especially the list of recommendations in Section 4 [14], which is a sensible list for security-critical software just as well as safety-critical software.)

3.4.3 Thread safety

Thread safety concerns the execution of multi-threaded or concurrent programs. With multi-threading cores and multi-core CPUs becoming more common, thread safety will become more of an issue. Indeed, concurrency is widely predicted to become a more important source of software problems (for more discussion, read [34]), and it will then also become a major source of security flaws. In fact, there already is a well-known category of security vulnerabilities when there is concurrent access to some resource, namely so-called **TOCTOU (Time-of-Check, Time-of-Use)** errors, also known as **'Non-Atomic Check and Use'**.

A procedure, function, method – or generally, some piece of code – is called thread-safe if it can be safely invoked by multiple threads at the same time. In an object-oriented language, a class is called thread-safe if multiple execution threads can simultaneously invoke methods on the same instance of that class.

In line with earlier characterisations of safety, one could define a programming language to be thread-safe if its semantics is well-defined in the presence of multi-threading. Even a supposedly safe programming language such as Java is inherently unsafe when it comes to concurrency: if a Java program contains data races, it may exhibit very strange behaviour, and one cannot make any guarantees about the semantics whatsoever. (A program contains a **data race** if two threads may simultaneously access the same variable, where one of these accesses is a write.)

For example, suppose initially x and y have the value 0 and we execute the following two threads:

```
Thread 1              Thread 2
 r1 = x;               r2 = y;
 y = 1;                x = 2;
```

One would expect either r1 or r2 to have the value 0 at the end of the execution. However, it is possible that in the end r1 == 2 and r2 == 1. The reason for this is that a compiler is allowed to swap the order of two assignments in a thread if, within that single thread, the order of these assignments does not matter. (Such re-orderings can make a program more efficient, if say a value held in a register can be reused.) So a compiler could reorder the statements in thread 1.

In fact, things can get a lot weirder still. Some language definitions do not even exclude the possibility of so-called *out-of-thin-air values*, where a concurrent program which only swaps values around between various variables can result in a variable receiving some arbitrary value, say 42, even though initially all the variables were zero [4].

The natural mental model of how a multi-threading program works, namely that all threads simply read and write values in the same shared memory – a so-called *strong memory model* – is fundamentally incorrect. Instead, we can only assume a *relaxed* or *weak memory model*, which accounts for the fact that individual threads keep shadow copies of parts of the shared memory in local caches, and where compiler optimisations like the one discussed above are possible. This affects both the possible behaviour of programs, and their efficiency. (For more on this read [23].)

Of course, functional programming languages have an easier job in ensuring thread safety, as they avoid side-effects – and hence data races – altogether. The same goes for other declarative programming languages, for instance logic programming languages such as Prolog.

One of the most interesting programming languages when it comes to thread-safety is Rust [6]. Rust was designed as a language for low-level system programming, i.e. as a direct competitor of C and C++, but with safety, including memory- and thread-safety, in mind. To ensure thread-safety, data structures in Rust are *immutable by default*. For mutable data structures, Rust then using the concept of *ownership* to ensure that only one owner can access it at the same time. Apart from the focus on thread-safety, Rust is also very interesting in the way it provides memory-safety without relying on a garbage collector for memory management.

3.5 Other language-level guarantees

Apart from typing information, a programming language can allow programmers to express other kinds of properties and also enforce these.

3.5.1 Visibility

One such property is **visibility**. In object-oriented languages the programmer can specify if fields and methods are private or public, or something in between. For example C++ has three levels of visibility: public, protected, and private. Java has four levels, as it also has a (default) package visibility. This provides some form of access control or *information hiding*. (We will stick to the term visibility because access control can already mean lots of different things.)

[6] For an intro, see https://doc.rust-lang.org/book.

Beware that the meaning of protected in Java and C++ is different.[7] More importantly, the guarantees that Java and C++ provide for visibility are very different. The absence of memory and type safety in C++ means that any visibility restrictions in C++ can be broken. In contrast, in safe languages such as Java and C# the visibility declarations are rigorously enforced by the programming language: e.g., if you define a field as private, you are guaranteed that no-one else can touch it.

The language guarantees provided by visibility are of course also useful from a software engineering point of view, as it rigorously enforces well-defined interfaces. This is also useful for security. E.g., an interface can provide a small **choke point** for all data to pass through, which can then be the place to perform input validation. In a language that is not type-safe one can never guarantee that such an interface cannot be by-passed. This becomes especially important in a setting where we have some untrusted code, i.e., if we trust some parts of the code more than others, typically because part of the code base is mobile code that we downloaded say over the internet, as is for instance the case with a Java applet in a web page. As we will discuss later, in Chapter 4, visibility is not enough here, and mechanism for enforcing more expressive policies than possible with visibility are needed.

Note that in Java protected is less restrictive in default package visibility: protected fields are accessible in all subclasses and in the package. This means that, from a security point of view, protected fields are not really that protected, and neither are package-visible fields: any code in the same package can access them. In a setting where we have untrusted, possibly malicious code, an attacker can get access to protected fields simply by declaring his attack code to be in the same package.

To defend against this, **sealed** packages were introduced in Java 1.4. By declaring a package as sealed, you effectively close the package, disallowing further extensions in other files. The package `java.lang` has always been a sealed package, since the very first release of Java, because it contains security-critical functionality that attackers should not be able to mess with by getting access to protected fields or methods.

If a class B is in a sealed package, and an attacker wants to get access to protected fields of objects of class B, instead of declaring a malicious class X in the same package as B, the attacker can also try to get creating a malicious subclass X of B in a different package: in this subclass the `protected` fields of B are visible. (Here it is important to realise that a subclass can be in a different package than its parent class, which may be surprising. Note that the way the visibility rules of Java work may not be so intuitive in these cases.) To prevent this, a class has to be declared as `final`.

All this means that `protected` is maybe a bit of a misnomer, as for fields it effectively only means

`protected-but-only-in-a-final-class-in-a-sealed-package`.

3.5.2 Constant values and immutable data structures

Another property commonly supported at programming language level is that of something being a **constant** or **immutable**, by declaring it `const` in C or C++, `final` in Java, or `readonly` in C#.

Although Java is a safe language, final fields are not quite constants: final fields are given their values during object initialisation (or during class initialisation, in case of static fields). So there is a (hopefully brief) period of time prior to this, when they still have their default initial

[7] In Java, protected methods are accessible in subclasses; in C++, protected methods are also accessible in so-called friend classes and functions.

value. Final instance fields cannot be read before they are initialised, so they really do behave as constants. However, final static fields can be read before they are initialised, namely in case there are circularities in the class initialisation.

For example, consider the two Java classes

```
class A {
  final static int x = B.y+1;
}

class B {
  final static int y = A.x+1;
}
```

Here one of the static fields, `A.x` or `B.y` – will have to be read before it is initialised (when it still has its default initial value zero), namely in order to initialise the other one. Depending on the order in which these classes are loaded `A.x` will be one and `B.y` two, or vice versa[8].

The example above shows that final static fields are not quite compile-time constants. In general, apparently simple notions such as being constant can be surprisingly tricky if you look closely at the semantics of a programming language. For instance, the precise semantics of `const` in C and C++ can be quite confusing.

Apart from having constant values for primitives types such as integers, one may also want more complex data structures to be constant or immutable. For example, in an object-oriented language one could want certain objects to be immutable, meaning that their state cannot be modified after they have been created.

The classic example of immutable objects are strings in Java. Once an object of type `String` has been created, it cannot be modified, so we can think of a Java string as a constant value. The fact that strings are immutable is in fact crucial for the security in Java, as we shall see when we discuss Java sandboxing. More generally, any object that you share with untrusted code should better be immutable, if you do not want that object to be modified by that code. One of the classic security loopholes in an early release of Java was due to mutability.

In fact, making objects immutable is a recommended programming practice. As Goetz puts it: ‘immutable objects can greatly simplify your life’ [15][9]. You do not have to worry about aliasing or about data races, and things become conceptually simpler.

Immutability of objects is a property that could be expressible and enforced by (the type system of) a programming language. The language Scala[10], which combines the object-oriented features of Java with aspects of functional programming languages, does so: Scala makes an explicit distinction between mutable and immutable data structures, both for primitives types and objects. As mentioned in Section 3.4.3, Rust also supports the notion of immutability of data structures, in order to achieve thread-safety.

[8] Clearly, such circularities in the initialisation are something you want to avoid! The code analysis tool FindBugs for Java includes a check to detect such circularities. FindBugs is free to download from `http://findbugs.sourceforge.net` and very easy to get working. If you have never used FindBugs or some other static analysis tool, download it and give it a go!

[9] `http://www-106.ibm.com/developerworks/java/library/j-jtp02183.html`

[10] `https://www.scala-lang.org`

Functionality versus security for visibility

Visibility provides another illustration of how security and functionality can be at odds with each other. Doug Lea's coding guidelines for Java[11] suggest that you should use `protected` rather than `private`:

> "Generally prefer protected to private.
>
> Rationale: Unless you have good reason for sealing-in a particular strategy for using a variable or method, you might as well plan for change via subclassing. On the other hand, this almost always entails more work. Basing other code in a base class around protected variables and methods is harder, since you have to either loosen or check assumptions about their properties. (Note that in Java, protected methods are also accessible from unrelated classes in the same package. There is hardly ever any reason to exploit this though.)"

But Gary McGraw and Edward Felten's security guidelines for Java[12] suggests the opposite, as these include

> "Make all variables private. If you want to allow outside code to access variables in an object, this should be done via get and set methods."

and warn against using package-level visibility (and hence also `protected`, as this is less restrictive):

> "Rule 4: Don't depend on package scope
>
> Classes, methods, and variables that aren't explicitly labelled as public, private, or protected are accessible within the same package. Don't rely on this for security. Java classes aren't closed [i.e., sealed], so an attacker could introduce a new class into your package and use this new class to access the things you thought you were hiding. . . .
>
> Package scope makes a lot of sense from a software engineering standpoint, since it prevents innocent, accidental access to things you want to hide. But don't depend on it for security."

The final sentence here recognises the possible conflict between a software engineering standpoint – extensibility is good – and security standpoint – extensibility is evil, or at least a potential source of problems.

Chapter 4

Language-based access control

When the operating system performs access control for a program in execution, as discussed in Chapter 2, it equally does so for all program code: the operating system has one access policy for the program as a whole, so all the code in it executes with the same permissions[1].

In contrast, **language-based access control** does not treat all parts of the program in the same way. Here the language provides a **sandboxing** mechanism to provide more fine-grained access control, where different parts of the program – which we will call components[2]– can be subject to different policies.

Sandboxing provide a defence against a more powerful attacker model or threat model than usual, Normally for a software application the attacker model is an **I/O attacker**, i.e. an attacker that can attack the application via its normal input and output channels. This attacker model can still be refined by specifying which input channels the attacker can access, or, in other words, what the **attack surface** is that the attacker can access. Sandboxing can provide protection against a more powerful **malicious code attacker**, i.e. an attacker who can inject code *inside* the application. This can happen in a direct way, say in a **supply chain attack** where the attacker compromises a library that is used, or even more directly, by a malicious insider who can insert malicious code in the application itself. It can also happen in an indirect way, namely if some security flaw in the codebase allows **remote code execution (RCE)**, and thereby gives the attacker the possibility to inject code.

The Log 4J vulnerability (CVE-2021-44228) is a good example of such a code injection vulnerability via a widely used Java library[3]. Th Log4J also highlights two negative facts: firstly, that the possibility to dynamically download code remains extremely dangerous, even if there are security measures such as sandboxing that can be used to mitigate these risks; secondly, the fact that so many Java applications were affected by the Log4J vulnerability demonstrates that applications do not make effective use of sandboxing as means to reduce the potential impact of supply chain attacks.

The term sandboxing is commonly associated with JavaScript, but as a programming language JavaScript does not have built-in features for sandboxing. When people talk about sand-

[1] An exception here are system calls: when a program invokes an operating system call, this system call is executed with higher privileges, for which the operating system will do a *context switch* for *user mode* into *kernel mode*.

[2] The notion of component was popularised by Szyperski [36], as a term for a reusable piece of software. We use the term rather loosely here, but components in the more precise sense of Szyperski could certainly be used as the components we consider.

[3] The report on Log4J US Cyber Safety Review Board [7] give a good summary of the incident and aftermath. It also makes recommendations to prevent similar incidents in the future. These recommendations are well-known and standard security advice, but it may still be interesting to read the report for a summary of such advice.

boxing in connection with JavaScript, the sandboxing is not done by the JavaScript execution engine but a surrounding environment such as a web browser. Here the HTML rendering engine (aka browser engine) *does* provide sandboxing, enforcing the access control for SOP (Same-Origin Policy), HTML5 iframe sandboxing and CSP (Content Security Policy). This engine works closely together with – and effectively includes – a JavaScript engine, so it is easy two confuse the two. Browser engines (and JavaScript engines) are not only used in web browsers but also in many stand-alone applications, both apps for smartphones and tablets and applications for desk- or laptops.

4.1 Language platforms

Programming language platforms (or 'frameworks'), such as Java or .NET, provide

1. an execution engine aka virtual machine that is responsible for executing code, and
2. an API that provides functionality (or services) of the platform itself and of the underlying operating system.

For the Java platform, officially called the Java Runtime Environment (JRE) and commonly referred to as 'the Java runtime', the execution engine is the Java Virtual Machine (JVM). The Java compiler compiles source code to *byte code* (aka class files) that the JVM can execute.

The Java API provides basic building blocks for programs (for example, `java.lang.Object`) and an interface to services of underlying operating system (for example `System.out.println`). It also encompasses some components responsible for the security functionality of the platform:

- the *class loader*, which is responsible for loading additional code at runtime;
- the *bytecode verifier (bcv)*, which is invoked by the class loader to make sure this code is well-typed; the type checking by the bytecode verifier together with the runtime checks and memory management by the JVM ensue memory safety.
- the (optional) *security manager*, which is responsible for enforcing the additional sandboxing discussed in this chapter.

With this sandboxing the platform performs access control where it treats different parts of the code – different components) differently, as illustrated in Figure 4.1. There are then two layers of access control, one by the language platform and one by the underlying operating system

The access control that can be done by at the level of the programming platform is more fine-grained than at the level of the operating system. So this allows a more rigorous adherence to the security principle of '*least privilege*' [29].

Having these two layers could be regarded as an instance of the security principle of '*defence in depth*' [29]. In principle, one could get rid of the whole operating system, 'hiding' it under – or inside – the language platform, so that the language platform is the only interface to the underlying hardware for application programs; this is for example what is done on JavaCard smartcards.

4.2 Why do language-based access control?

Why would you want to do fine-grained access control which enforces different policies for different parts of the code? The fundamental reason is that different parts of the code may come

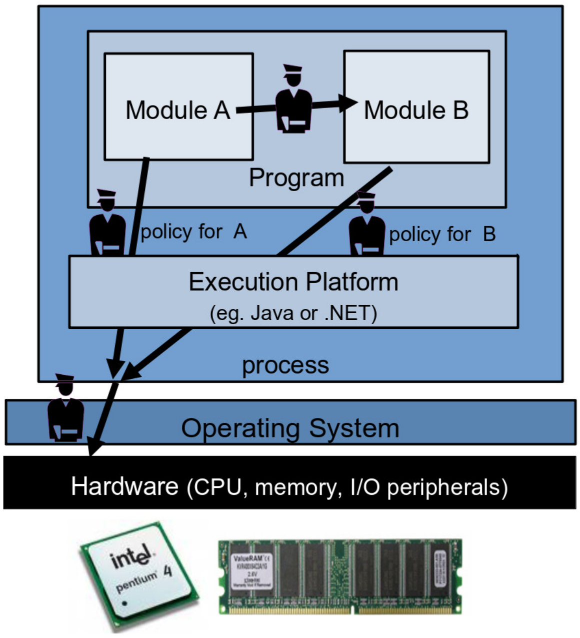


Figure 4.1: Security controls at programming platform level: separation between components and access control per component.

from different sources that are not all equally trustworthy. Especially **mobile code** – downloaded over the internet to extend an application 'on the fly' – should be treated with suspicion.

The security features of Java were designed to make it possible to securely run mobile code, downloaded over the internet at runtime. The original idea was that Java applications, so-called *applets*, could be launched from a web-browser. Clearly you might not want to execute this code with all the access rights of the user, or even the access rights of the web browser. For example, your web browser can access the hard disk, but you don't want to give some applet on any web page that right. The sandboxing mechanism in Java (aka *language-based access control*, *stack-based access control* or *stack walking*) discussed in this chapter was designed to make it possible to execute untrusted code – or, more generally, less trusted code – with precisely tuned permissions.

This sandboxing mechanism is not just useful in the scenario where mobile code downloaded and executed client-side. Another envisioned use case for Java was at the server side, where a web server could run multiple web application (so-called servlets) and keeping the sandboxing mechanism to separate them and control their access rights.

A more relevant use case for sandboxing nowadays might be to provide protection against supply chain attacks. We already mentioned the example of the Log4J vulnerability from 2021. Supply chain attacks have been steadily becoming more popular since the early 2010s, and sandboxing can be used to reduce the impact such attacks can have, by giving reduced access rights to such libraries.

But much more generally, it may be wise to also reduce the access rights of code that does not need certain high privileges, even if that code is not in untrusted – or less trusted – third party libraries. If we use sandboxing to give certain dangerous access rights only to a small part of the code base, then we can concentrate our quality assurance efforts on this part of the code: e.g. setting the best programmers on that part of code, testing it more thoroughly, subjecting it to code reviews, etc. Producing better quality and thereby more secure code takes effort and costs time and money. Concentrating these efforts on a small part of the codebase makes good economic sense.

Mobile code executed client-side in web browsers has become huge success: nearly every modern website involves mobile code. However, JavaScript has completely eclipsed Java (and other client-side programming languages such Adobe Flash and Microsoft Silverlight) as the programming language that is used for this: nearly all webpages nowadays do contain mobile code, but in mainly the form of JavaScript[4] (See [33] for some statistics on this.)

In the past decades the term 'mobile code' has quickly turned into a pleonasm: most code is downloaded or regularly updated over the internet, including the operating system itself. Many of these forms of mobile code are much more dangerous than applets, as there are no sandboxing mechanisms in place to constrain them.

Moreover, the distinction between data and code is often very blurred. Much of what we think of as data – e.g., Microsoft Office documents and spreadsheets, PDF or postscript files, web pages, and much of the 'content' that we view in apps on our mobile phones or laptop – can contain executable scripts. So much of the 'data' that we email around or download from the internet is in fact mobile code.

Trust, trustworthiness, and the TCB

The notion of trust is important in security. It is also a source of confusion, especially if people are sloppy in their terminology, and do not distinguish between **trust** and **trustworthiness**.

Depending on your point of view, trust can be something good and desirable, or something bad and undesirable. Trust between parties is good in that it enables easy interaction and good collaboration between them. However, trust is bad in that trust in another party means that party can do damage to you, if it turns out not to be trustworthy. For example, if you give someone your bankcard and tell them your PIN code, you trust them; this can be useful, for instance if you want them to do some shopping for you, but is clearly also potentially dangerous.

Note that if a party is not trustworthy, then it may be so unintentionally (because it is careless or, in the case of software, riddled with security vulnerabilities) or intentionally (because it is downright malicious).

When considering a system that is meant to meet some security objectives, it is important to consider which parts of that system are trusted in order to meet that objective. This is called the **Trusted Computing Base** or **TCB**.

Ideally, the TCB should be *as small as possible*. The smaller the TCB, the less likely that it contains security vulnerabilities. (Still, you should never underestimates people's stupidity – or an attacker's creativity – to introduce security vulnerabilities in even the smallest piece of software.) Also, the smaller the TCB, the less effort it takes to get some confidence that it is trustworthy, for example, in the case of software, by doing a code review or by performing some (penetration) testing.

[4] NB Despite the similarity in name, JavaScript has nothing to with Java! Java is an general purpose object-oriented programming language, which could be used for downloading and launching applications over the internet. JavaScript is a scripting language originally designed just for use in web pages, though it has over time been used for more other purposes. The only thing the two language have in common is that they use curly brackets as part of their syntax. It is somewhat ironic that JavaScript has eclipsed Java as the programming language for mobile code: the name JavaScript was originally chosen to suggest a relation with Java and benefit from the publicity and hype around Java.

4.3 Language-based access control and safety

Obviously, language-based access control is impossible in a programming language that is not memory safe. Without memory safety, there is no way to guarantee separation of memory used by different program components. For example, in Fig. 4.1, code in component A should be prevented from accessing data belonging to component B. Stronger still, it should be prevented from accessing data of the underlying language platform since any data used in enforcing the access control by the language platform could be corrupted.

It is also hard to envisage how one could achieve language-based access control without type safety. Access control between components inside a program (as suggested by the horizontal arrow in Figure 4.1) relies on the visibility declarations to define e.g., public versus private interfaces, as discussed in Section 3.5.1. Without type safety any restrictions on visibility can be accidentally or deliberately broken. Also, without type safety one has to be extremely careful with data passed between components as arguments or results of procedures and method calls, as this data can be malformed. This also applies to data shared between components via aliasing, as discussed in more detail in Section 4.5.2.

In the case of Java, the security of the sandboxing mechanism discussed in Section 4.4 below crucially relies only many of the notions of safety discussed in Chapter 3: memory safety, type safety, visibility restrictions, and also immutability of objects of some core Java classes, notably `String` objects, to avoid security problems due to aliasing.

In light of the discussion above, one could even define a language as being safe iff it is possible to securely provide sandboxing at the language level.

4.4 How to do language-based access control?

Any kind of access control level, involves **policies** to specify access rights, a **policy language** to express these in, and a **mechanism** to enforce them.

4.4.1 Policies for language-based access control in Java

Sandboxing at the language level involves policies that assign permissions to program components. The basis for assigning permissions is typically the **origin** of this code, which can be the physical origin of the code (did it come from the hard disk – and, if so, from which directory– instead of being downloaded over the internet) or digitally signatures on the code that prove its provenance.

The permissions that are given are rights to perform all kinds of actions, e.g., accessing the file system, the network, the screen, etc. Some permissions are provided by the language platform, and used to control access to functionality that the platform provides. In Java the standard permissions include

- `java.io.FilePermission` for the file system access
- `java.net.SocketPermission`, `java.net.NetPermission` for network access, and
- `java.awt.AWTPermission` for user interface access.

For specific platforms there may be other permissions. For example the MIDP platform for Java-enabled mobile phones distinguishes permissions for components[5] to dial phone numbers or

[5] The Java components here are called midlets rather than applets.

send SMS text messages[6].

An example policy in Java is

```
grant signedBy "Radboud"
{ permission
   java.io.FilePermission "/home/ds/erik","read";
};
grant codebase "file:/home/usr/*"
{ permission
   java.io.FilePermission "/home/ds/erik","read";
}
```

which grants all code signed by 'Radboud' and all code in `/home/usr` on the file system read-permission in the specified directory. Here 'Radboud' is an alias for which the corresponding public key has to be defined in the *keystore*.

4.4.2 Stack walking

Program components can invoke each other's functionality by method calls. This complicates the precise meaning of policies: if a component A uses functionality of another component B, should it run with its own permissions or that of the other component?

Assume that a method `good()` is in a component `Trusted` that has permission to reformat the hard disk, and a method `evil()` is in a less-trusted component `Evil` that does not have this permission.

If the method `good()` invokes the method `evil()`, should `evil()` have permission to reformat the hard disk? After all, it is acting 'on behalf of' `good()`. The obvious answer is that it should not: if so, it would be easy for some untrusted code to create havoc.

The opposite situation is more tricky: if `evil()` invokes `good()`, should `good()` still use its permission to reformat the hard disk? Clearly there are dangers. But it is unavoidable that trusted components have to carry out requests by untrusted components. In fact, this is an important role for trusted components.

For an analogy in the physical world, assume you walk into the branch office of your bank and ask a cashier for some money from your bank account. From the perspective of the bank, you are the untrusted party here and the cashier is trusted: the cashier has access to the money and you don't. There may even be some bullet proof glass between you and the cashier to help enforce this access control. The software equivalent would be an untrusted `Client` component invoking the `withdraw_cash` method of some `BankEmployee` component. When carrying out your request, the cashier will be using his privileges to get to the money. Carrying it out with only your permissions is clearly impossible. Of course, the cashier should be very careful, and e.g., verify your identity and check your bank balance, before handing over any cash. Unfortunately (or fortunately, depending on the point of view) human cashiers are not susceptible to buffer overflow attacks, where a client giving a carefully chosen and ridiculously long name will be given piles of cash without any checks.

A standard way to handle this is by **stack inspection** aka **stack walking**. Stack inspection was first implemented in Netscape 4.0, then adopted by Internet Explorer, Java, and .NET.

[6] Here the MIDP access control is very fine-grained. For example, a midlet can be given the right to send text messages or use internet access, but there may then still be a security pop-up where the user is asked for permission. Depending on how far the midlet is trusted, permission may then be granted to the midlet for its entire lifetime, only for the current session, or it may be a one-shot permission, e.g., for sending a single SMS.

The basic idea here is that

> whenever a thread T tries to access a resource, access is only allowed if *all* components on the call stack have the right to access the resource.

So the rights of a thread are the intersection of the rights of all outstanding method calls. The rationale for this is that if there is a method `evil` on the call stack that does not have some permission, there is the risk that this method is abusing the functionality. If the `evil` method is on top of the stack, it may try to do something unwanted itself; if the `evil` method is not on top of the stack, then it might be trying to trick some other code (namely the method higher on the stack that is invoked by `evil`) to do something unwanted.

As the example of the cashier at the bank suggests, this basic idea is too restrictive. A component can therefore override this and allow to use some of its privileges that a component lower on the call stack does not have. To do this, that component has to explicitly enable usage of this privilege, to reduce the chance that privileged functionality is exposed accidentally. In Java enabling a privilege is done using the method `enablePrivilege`. The decision to enable a privilege can also be revoked, by calling the method `disablePrivilege`.

Doing access control when some permission is used, now involves inspecting the frames on the call stack in a top-down manner – i.e., from the most recent method call down to the method call that started execution (e.g., a `main` method in some class). If a stack frame is encountered that does not have a privilege, access will be denied. If a stack frame is encountered that does have the permission *and* has it enabled, the walk down the stack is stopped and access is granted. If the stack inspection reaches the bottom of the call stack, which can only happen if all frames have the permission but none of them have explicitly enabled it, permission will be granted.

For a detailed description of stack inspection in Java, see Chapter 3 of [21]. The code in Figure 4.2 illustrates how code for the cashier example discussed above might look like.

Elevating privileges

The need for trusted code (or trusted processes) to make available some of its privileges to untrusted code (or untrusted processes) is quite common.

As mentioned in the footnote on page 28, one place in which this happens in traditional operating systems is in system calls: when a program invokes an operating system call, this system call is executed with higher privileges.

Another way this happens on UNIX systems is through `setuid` executables. These executables run with the access rights of the owner of the executable rather than the access rights of the user that invokes the program. Often `setuid` executables are owned by `root`, giving them maximal rights. Windows operating systems offer so-called Local System Services to do roughly the same thing.

The classic example to illustrate the inevitable need for such temporary Elevation of Privilege is a log-in operation: for an unauthenticated user to log in with a username and password, access to the password file is needed. So a high privilege (access to the password file) is needed by some action executed on behalf of a user who is at that point still completely untrusted.

Mechanisms that offer temporary Elevation of Privilege are notorious sources of security vulnerabilities: if they contain security vulnerabilities then the higher privileges allow an attacker to do greater damage.

```
  // Untrusted BankCustomer, without CashWithdrawalPermission
  class BankCustomer {
    ...
    public Cash getCash(Bank bank, integer amount){
      walkIntoBank(b);
      Cashier cashier = b.getCashier();
      return  cashier.cashWithdrawal(this, amount);
    }
    ...
  }

  // Trusted Cashier, with CashWithdrawalPermission
  class Cashier {
    ...
    public Cash cashWithdrawal (Customer c, integer amount){
      // perform security checks,
      // eg, authenticate customer & check his balance
      enablePrivilege(CashWithdrawalPermission);
      cash = teller.withdraw(amount);
      // update customer's bank balance
      ...
      return cash;
    }
    ...
}
```

Figure 4.2: Sample code for the banking example. A more secure implementation of `cashWithdrawal` would disable its `CashWithdrawalPermission` straight away after getting the cash, to avoid the risk of accidentally using this permission in the rest of the method.

4.5 Things that can go wrong

As discussed in Section 4.3, the TCB for language-based access control is quite large. Even if this TCB is correct (and hence trustworthy), which means that

- the type system is sound, and
- the implementation of the type checker is correct, and
- the language platform (including the virtual machine and the API classes responsible for the language-based access control) are implemented correctly & securely, and
- all the API classes that these implementations rely on are implemented correctly & securely,

there are still things that can go wrong, as discussed below.

4.5.1 The need for defensive programming

A particular Java class may still be vulnerable to attacks by untrusted code if it is not programmed *defensively*, even if there are no mistakes in the policy file and the keystore does not contain incorrect data.

For example, a component could be careless and expose public fields, allowing an attacker to do damage by changing these, or a component might allow a class to be sub-classed, allowing an attacker to create a sub-class whose instances have malicious behaviour.

To prevent such vulnerabilities, the programmer has to keep security in mind and program *defensively*. Defensive programming guidelines for Java are discussed in Chapter 7.1 of [21] or [37]. Secure coding guidelines for Java are also published by CERT[7] and Oracle[8].

Of course, when implementing any components that are in the TCB for the language-based access control platform itself, such defensive programming guidelines should also be followed. This code not only has to be correct, but it also has to be secure against possible attacks by malicious code running on the same VM.

4.5.2 Aliasing as cause of security trouble

One notorious loophole in enforcing separation between trusted and untrusted code is **aliasing**: if a trusted and untrusted component both have a reference to the same object this may be a way for the untrusted code to mess up the behaviour of the trusted code. As illustrated in Fig 4.3, aliasing can bypass any access control enforced by the language. A classic example of this is with `private` reference field: the language prevents access to private fields by 'outsiders', but the object that a private field points to *can* be accessed by anyone that has an alias to it.

Aliasing is harmless if the object in question is **immutable**. If a trusted and untrusted component share a reference to a `String` object in Java, then, because strings are immutable, there is no way the untrusted component can modify the shared reference. (Note that to ensure that `String` objects are immutable in the presence of untrusted code, it is crucial, amongst other things, that the class `java.lang.String` is final, to rule out malicious subclasses which allow mutable strings.)

[7] `https://www.securecoding.cert.org/confluence/display/java/The+CERT+Oracle+Secure+Coding+Standard+for+Java`

[8] `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`

The classic example of how aliasing of mutable data can be a security hole is the 'HotJava 1.0 Signature Bug'[9]. The problem was caused by the implementation of the getSigners method in java.lang.Class, a method which returns an array of the signers of a class. The signers associated with a class are an important basis of language-based access control, as policies can associate certain permissions with signers.

The code was something like

```
package java.lang;
public class Class {
   private String[] signers;

   /** Obtain list of signers of given class */
   public String[] getSigners() {
      return signers;
   }
  ...
```

The problem with the code above is that the public method getSigners returns an alias to the private array signers. Because arrays in Java are always mutable, this allows untrusted code to obtain a reference to this array and then, change the content by including a reputable signer such as sun.java.com in the hope of getting high privileges.

The solution to the problem is for getSigners to return a copy of the array signers rather than an alias.

Note that this example illustrates that private fields are not necessarily that private: private fields that are references may be widely accessible due to aliasing.

Alias control to prevent aliasing bugs

Unwanted aliasing is a not just a notorious source of security holes, but it is a notorious source of bugs in general. There has been a lot of research into type systems for programming languages which enforce some alias control to rule out unwanted aliasing. Especially in an object-oriented language it is required to make some hard guarantees about encapsulation of objects. E.g., one could consider adding a modifier private_and_unaliased to Java to declare reference fields as being 'really private', which are then not allowed to be aliased from outside the class.

How to restrict aliasing is still an active topic of research. An example of a language platform that enforces restrictions on aliasing is Java Card; on Java Card smartcards it is not possible for applications in different packages to share references across package boundaries.

Because sharing mutable data is dangerous, one countermeasure is to only share immutable data. If this is not possible, an alternative is to make copies of objects – by cloning them – when passing them to untrusted code, or after receiving them from untrusted code. For example, in the Class example above, the method getSigners should not return signers, but signers.clone().

One has to be careful here to make a sufficiently deep copy of the data. In Java cloning an array returns a *shallow* copy, so the elements of signers and signers.clone() are aliased. Because the array contains immutable strings, it would be secure in getSigners to return signers.clone(). If strings were mutable, this would still not be secure, and a deep clone of the signers array would be needed, where all the elements of the array are cloned as well.

[9] For this and more examples of security holes in early versions of Java, see http://sip.cs.princeton.edu/history/index.php.

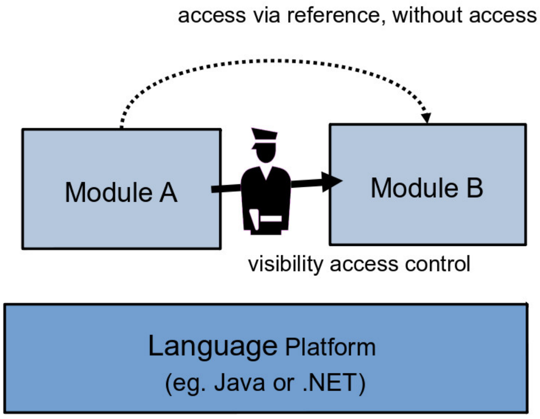

access via reference, without access control
Module A
Module B
visibility access control
Language Platform
(eg. Java or .NET)

Figure 4.3: Referencing as notorious loophole in language-level access control, bypassing language-level access control.

A way a programming language could help here is in offering some heuristics – or better still, guarantees – about how deep a clone method is. For example, the object-oriented programming language Eiffel distinguishes `clone` and `deep_clone` operations, where the former is a shallow clone. In other languages such as Java or C# it can be unclear how deep a copy the `clone` method returns, which may lead to mistakes.

4.5.3 The security promise and failure of Java

Security was a key consideration in the design of Java. When Java was introduced in the 1990s, it came with big promises and hopes to improve security.[10]

As discussed earlier, the security features of Java were designed to make it possible to securely run mobile code downloaded over the internet at runtime. The original idea was that Java applications, so-called *applets*, could be launched from a web-browser. Later on Sun also introduced Java Web Start as a way to download and run Java applications from the internet without the use of a browser.

The security features of Java include *memory-safety*, *type-safety*, *information hiding (or visibility)*, *immutable objects*, all discussed in the previous chapter, and *language-based access control*, also known as *stackwalking* or *stack based access control*, discussed in this chapter.

In the end, the Java platform ended up with quite a bad security reputation, especially in its use for mobile code executed via a browser plugin. In the early 2010's Java was widely reported to be the main source of security vulnerabilities, beating even Adobe Flash. For a historical impression of Java security woes, Brian Krebs' website is a good source[11]. There are also some good, more technical studies of Java's security problems [18, 12].

Several factors played a role in Java's security problems, some of them very specific to technical internals of Java, other much more generic:

[10] Java was introduced by the Sun Microsystems, which was acquired by Oracle in 2010. After Sun introduced Java, Microsoft introduced C# (pronounced C-sharp), an almost identical programming language with an associated execution platform, the .NET platform.

[11] `https://krebsonsecurity.com/tag/java`

- One important factor is the complexity of Java's sandboxing mechanism, which involves multiple components, including for instance the `ClassLoader` and the `SecurityManager`. This means a sizeable chunk of the Java API code is in the TCB for the sandboxing mechanism, and this code is quite complex. Obviously, the chances of bugs in the TCB increases with the size and the complexity of the code. The evolution of Java language and its APIs over time, with the original developers leaving and new programmers joining, meant that the code base grew and new bugs were introduced over time.

- A related issue is that part of the security mechanisms for the sandbox (for instance `ClassLoader` and the `SecurityManager`) are – at least partly – implemented in Java. This highly trusted code then runs alongside any downloaded Java code on the same VM. This presents a huge attack surface for a attacker who uses malicious code as attack vector: *all* the features of the Java languages and *any functionality* in the Java libraries can be (ab)used to attack the trusted code. Particular features that are very powerful and which have turned out to be especially dangerous here include the reflection API, the `MethodHandles` API, and deserialisation, as discussed in detail by Holzinger et al. [18].

  Reflection allows code to inspect itself: code in a class can use it to obtain references to its own class, to other classes, and then to fields and methods of these classes, and possibly by-pass visibility restrictions to accessing private fields.

  Deserialisation is not just a security worry in Java, but also in other programming languages. Deserialisation was included in the OWASP Top Ten in 2017. One issue is that deserialisation of malicious input may result in objects or data structures that are malformed or that violate basic assumptions that are made in the code; this may trigger weird behaviour that an attacker is interested in. Another more direct threat is that deserialisation triggers code execution: deserialising an object of some class will typically trigger the execution of constructors of that class to create the deserialised object. This provides a way for an attacker to trigger code execution. In large applications with rich APIs this may effectively allow the attacker to execute arbitrary code.

Other factors contributing to the security problems with Java are not so much about Java's internal workings, but more about the functionality provided and the way this functionality is integrated into other systems, notably web-browsers:

- The simple fact that downloading remote code is a built-in *feature* of Java, and it may even be a feature that is on by default in some browser plugins, obviously makes it an interesting place to attack and one that is easy to reach. It is no coincidence that Adobe Flash, another technology notorious a prime source of security vulnerabilities, also provides a powerful execution engine reachable via browser plugins.

- The widespread deployment of Java meant is was an interesting target for attackers, with the added advantage that it potentially works cross-platform on Windows, Apple and Linux machines.

- The fact that Java (like Flash) is third-party code on most systems does not help with getting updates rolled out.

  Additionally, Java's update mechanism has also come in for criticism because when installing new versions it might leave the older version behind. Obviously users having up with multiple version of Java on their computer, and possibly needing multiple versions to run old applications, does not help with keeping things up to date. Microsoft update mechanism for .NET worked in the same way.

Only in 2012, did Java 7 introduce the ability to disable any Java application from running in the browser by simply unchecking a tick-box in a control panel[12], which is obviously a sensible precaution if you do not intend to use this feature. Turning off Java in modern browsers is something you no longer have to worry about. In 2015 Chrome dropped support for the cross-platform plugin architecture NPAPI, which disabled Java (and also Microsoft's Silverlight). In 2018 the last version of Firefox still supporting NPAPI dropped it.

Code-based versus process-based access control

The nature of access control in language-based access control is different from access control in standard operating systems: the former is based on origin of the code and the associated visibility restrictions, the latter is based on the process identity. The examples below illustrate this.

Suppose on an operating system you start the same program twice. This results in two different processes. (Sometimes when a user tries to start a program twice, instead of starting a second process, the first process will simply spawn a new window. Indeed, when clicking an icon to start a program it not always clear if this starts a second process or if the first process simply creates a second window. You can find out by checking which processes are running on the operating system. Another way to find out is when one of the windows crashes: if the other window also crashes then it probably belonged to the same process. From a security point of view, it is preferable to have separate processes.) Although these two processes execute the same code, the operating system will enforce separation of the address spaces of these programs. That the processes execute the same code does not make any difference (except maybe that the processes might try to use the same files).

Now suppose that within a process that executes a Java program you start up two threads, for instance by opening two windows.

If these threads execute different code, say one is started by `new.SomeThread().start()` and the other by `new.AnotherThread().start()`, then the visibility restrictions of the programming language can prevent the first thread from accessing some data belonging to the second thread. For example, private fields in the `SomeThread` class will not be visible – and hence not accessible – in the `AnotherThread` class. If the classes are in different packages, they will not be able to access each other's package-visible fields.

But if both threads execute the same code, say they are both started by `new.SomeThread().start()`, they can access each others private, protected, and package-visible fields. (Note that in Java a private field is not only accessible in the object it belongs to, but by all objects of that class.)

Even if the threads execute different code, the visibility restrictions between them are rather fragile. We do not have hard and simple encapsulation boundaries between these threads like we have between different processes on the operating system. For example, suppose one threads executes `new.SomeThread().start()` and the other `new.AnotherThread().start()`, where `SomeThread` and `AnotherThread` are in different packages. There is then some separation between the threads since they only touch each other's public fields. However, the two threads might share references to some common object, due to aliasing, which might be a way for one execution to influence the other in unintended ways.

[12] https://www.oracle.com/technetwork/java/javase/7u10-relnotes-1880995.html

One prominent example where separation between different objects of the same class would be useful, are the different tabs of a window or a web browser.

For example, suppose you look at different web pages simultaneously with your web browser, either in different windows or in different tabs in the same window, and in one of these you are logged in to your Gmail account. If you visit a website with malicious content in another web page, you might be exposed to all sort of security vulnerabilities, such as cross-site scripting, session fixation, session hijacking, cross-site request forgery, or click jacking. The root cause of all these attacks is the same: the other window executes with the same permissions as your Gmail window.

If you would start up new *processes* instead of starting new windows or tabs within the same process to look at different web pages, many of these attack vectors would disappear. Of course, for this to work cookies should not be stored on disk and shared between these processes, but should be in main memory to benefit from the memory separation enforced by the operating system.

Chapter 5

Information flow

Information flow is about a more expressive category of security properties than traditionally used in access control. It is more expressive than access control at the level of the operating system discussed in Chapter 2 or at the level of the programming platform discussed in Chapter 4. This chapter discusses what information flow properties mean, both informally and formally, how they can be specified, and how they can be enforced.

Traditional access control restricts what data you can read (or write), but not what you can do with this data after you have read it. Information flow does take into account what you are allowed to do with data that you have read, and where this information is allowed to *flow* – hence the name. For write access, it not only controls which locations you can write to, but also controls where the value that you write might have come from.

As an example, consider someone using their smart phone to first locate the nearest hotel by say using Google maps, and then book a room there via the hotel's website with their credit card. Maybe they even use some HotelBooker app on their phone that does all these actions for them. The sensitive information involved includes the location information and the credit card details. The location data will have to be given to Google, to let it find the nearest hotel. The credit card information will have to be given to the hotel to book a room. So an *access control policy* for a HotelBooker app would have to allow access to both location data and the credit card number. There is no need to pass the credit card information to Google, or to pass the current location data to the hotel; especially the former would be very undesirable. Hence an *information flow policy* for a HotelBooker app might say that location data may *only* be leaked to Google and the credit card may *only* be leaked to the hotel. Note that such an information flow policy specifies more precise restrictions than access control can, and involves the way information flows inside the application.

Integrity versus Confidentiality

Information flow properties can be about **confidentiality**, as in the example above, or about **integrity**. For confidentiality we are worried about where some data that has been *read* might end up. For integrity we are worried about where some data that is *written* might have come from. The former can violate the confidentiality of the data, the latter can violate the integrity of the data.

As an example, suppose an application receives untrusted, and potentially malicious, input over the internet. Such untrusted data is usually called **tainted** data. You might not want this input to be used in sensitive actions, say as the argument of `system()`, the standard C operation to pass a raw command to the underlying operating system.

Lack of input validation is a major source of security vulnerabilities; these all involve unwanted information flows where some untrusted (i.e., low integrity) data ends up in a sensitive place (where only high integrity data should be used). Such problems can be detected and prevented to some degree by information flow control using a policy for integrity. This is also called **taint checking**.

Integrity and confidentiality are *duals*. This means that for every property of confidentiality there is a similar but somehow opposite property for integrity. We will see some more examples of this duality later: for every way to specify or enforce information flow for confidentiality, there is a corresponding – dual – approach for integrity.

Integrity versus confidentiality

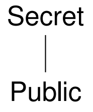
The duality between confidentiality and integrity shows up in many places. E.g., asymmetric cryptography can be used to encrypt data (ensuring confidentiality) or to sign data (ensuring integrity).

When you think about security, often your first instinct is to worry about confidentiality. Indeed, some people think that security is all about secrecy. But in many (most?) situations integrity is a far greater worry than confidentiality. After all, for data for which confidentiality is important, such as your medical records, your criminal record, or information about your finances, integrity is usually also important – and often much more. You may find it annoying if other people can find out how much money you have in your bank account, but if they can change this number it is a more serious security concern!

Moreover, many security problems in software are caused by malicious input and insufficient input validation – i.e., insufficient integrity checks. Malware tries to destroy the integrity of the systems we use, even if the goal is to then steal confidential information.

5.1 Principles for information flow

The work on information flow dates back to the seminal work of Denning & Denning [11]. Information flow properties, be it for integrity or confidentiality, involve a **lattice** to classify data. The simplest possible lattice for confidentiality just distinguishes public and secret data:



Much more complicated lattices are possible, for instance to classify data at different secrecy levels or on different topics, as illustrated in Figure 5.1.

The lattice on the right in Fig. 5.1 shows that secrecy levels do not have to be a total ordering: someone with clearance to read confidential information about the Netherlands is not necessarily cleared to read confidential information about Belgium, or vice versa. Still, given incomparable secrecy levels there is always a least upper bound – in this case ‘Conf. Benelux’.

A lattice for integrity nearly always just considers two categories, which distinguishes tainted

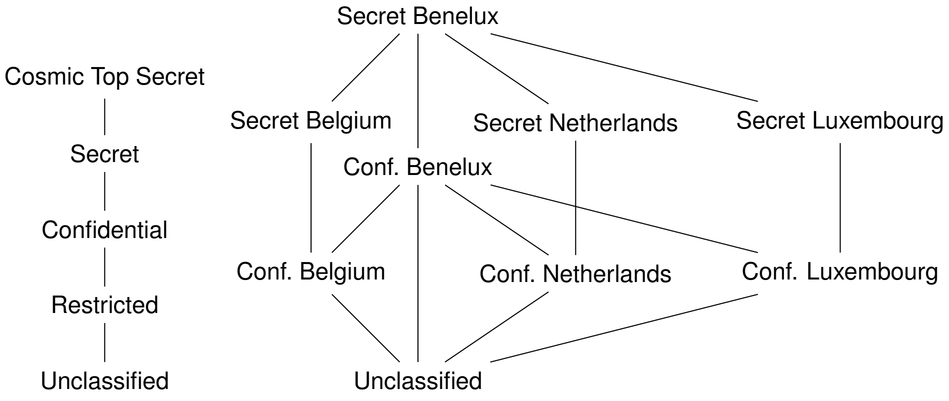
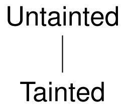


Figure 5.1: Lattices for confidentiality: on the left the official NATO classification (which does really include the level 'Cosmic'!), on the right some imaginary classification of sensitive information about the Benelux countries.

data – typically untrusted input – from the rest:



These lattices provide an ordering on the different categories: data has a higher degree of confidentiality or integrity if you go up in the lattice. There is also an implicit 'inclusion' in one direction. For example, it is safe to treat some public data as if it were secret, but not the other way around. Similarly, it is safe to treat some untainted data as if it were tainted.

Information flow properties involve **sources** where information come from and **sinks** where information end up in. In the discussion and examples below we will often use program variables as both sources and sinks. In addition to this, input mechanisms give rise to sources and output mechanisms to sinks.

5.1.1 Implicit and explicit flows

In information flow we distinguish explicit and implicit flows.

An **explicit** (or **direct**) information flow happens when we do an assignment of a value to a variable:

```
x = y;
```

is an explicit flow from `y` to `x`.

If a variable `lo` holds public data and a variable `hi` holds secret data, then the assignment `lo = hi;` breaks confidentiality, whereas `hi = lo;` does not.

Dually, if a variable `lo` holds untrusted, tainted data and a variable `hi` holds untainted data, then `hi = lo;` breaks integrity, whereas `lo = hi;` does not.

Implicit (or **indirect**) flows are trickier. Suppose that program variable `lo` holds public data and variable `hi` holds secret data. The statement

```
if (hi > 0) { lo = 99; }
```

does not leak the exact value of hi to lo, but it does leak *some information about* the value of hi to lo. Someone observing lo could tell whether hi is negative or not.

In the worst case, an implicit flow is just as bad as an explicit one. For example, for boolean variables b1 and b2, the (implicit) information flow in

```
if (b1) then b2 = true else b2 = false;
```

is equivalent to the (explicit) information flow in

```
b1 = b2;
```

Implicit flows can become quite tricky. Suppose we have two arrays, priv and pub, where priv contains secret data and pub contains public data. The assignments below are then not problematic

```
pub[3] = lo;
priv[4] = hi;
```

But what about the statements below?

```
pub[hi] = 23;
priv[hi] = 24;
priv[lo] = 25;
```

The assignment pub[hi] = 23; does leak confidential information, because someone observing the content of pub could learn something about the value of hi. More subtle is the problem with priv[hi] = 24; this may look harmless, but it may leak information: e.g., if hi is negative, the statement will throw an exception because of a negative array index; if attackers can observe this, they can tell that hi was negative.

Finally, priv[lo] = 25 will throw an exception if lo is negative or if lo larger than the length of the array priv; if the length of the array priv is considered confidential, then in the second case some confidential information is leaked.

Information can not only be leaked by throwing exceptions, but also by the execution time of a program, or by the fact that a program terminates or not. For example, consider the program fragment below

```
for (i = 0; i < hi; i++) { ... };
```

If after this for-loop there is some event that an attacker can observe, then the timing of this event leaks some information about the value of hi. This is also the case if there is some observable event inside the loop.

The program fragment

```
for (i = 0; i < 10; i=i+hi) { ... };
```

will not terminate if hi is zero, so this program fragment not only leaks information by the time it takes to complete, but also by the fact whether it terminates at all. This may be easier to observe by an attacker.

Hidden channels

In the examples above, the throwing of an exception, the timing, and the termination of a program is used as so-called **covert** or **hidden** channel.

Especially when implementing cryptographic algorithms timing is a notorious covert channel. For instance, a naive implementation of RSA will leak information about the key through time, as the time it takes will increase with the numbers of 1's in the secret key. There are all sorts of covert channels that might leak sensitive information: such as power usage, memory usage, electromagnetic radiation, etc. One of the most powerful attacks on smartcards to retrieve secret cryptographic keys nowadays is through power analysis, where the power consumption of a smartcard is closely monitored and statistically analysed to retrieve cryptographic keys.

The notion of covert channels is about observing data, rather than influencing it, so it is an issue for confidentiality, but less so (not at all?) for integrity. So here the duality between confidentiality and integrity appears to break down.

5.2 Information flow in practice

There are different ways to use the notion of information flow to improve security:

- One way is through **type systems** which take levels of confidentiality or integrity into account. Breaches of information flow policies can then be detected at compile-time by means of type checking. (Of course, breaches could also be detected at runtime.) Section 5.3 describes this possibility in more detail. An early realisation of such a type system for a real programming language is the Jif extension of Java, which is based on JFlow [24].

  With the extension of the Java annotation system to allow annotations on types (JSR 308), information about security levels can be added to Java code without the need to extend the language. The SPARTA project uses to make compile-time information flow guarantees for Android apps [13].

- Many **source code analysis tools** perform some form of information flow analysis. Source code analysis tools, also known as code scanning or static analysis tools, analyse code at compile time to look for possible security flaws. The capabilities of source code analysis tools can vary greatly: the simplest versions just do a simple syntactic check (i.e., a CTRL-F or `grep`) to look for dangerous expressions, more advanced versions do a deeper analysis of the code, possible including information flow analysis.

  Information flow analyses by code analysis tools focus on integrity rather than confidentiality, and is also called *taint checking*. Some sources of data are considered as tainted (for example, arguments of HTTP POST or GET requests in web applications) and the tool will try to trace how tainted data is passed through the application and flag a warning if tainted data ends up in dangerous places (for example, arguments to SQL commands) without passing through input validation routines.

  So the tool has to know (or be told) which routines should be treated as input validation routines, i.e., which routines take tainted data as input and produce untainted data as output. Also, the tool will have to know which API calls give rise to information flows. For example, if a Java string `s` is tainted, then

  ```
  s.toLowerCase()
  ```

probably has to be considered as tainted too, but

```
org.apache.commons.lang.StringEscapeUtils.escapeHtml(s)
```

can be considered as untainted. Or rather, the last expression can only be considered as untainted when it is used as HTML, because it may still be tainted when used in say an SQL query.

Taken to its extreme, such an information flow analysis would be equivalent to having a type system. However, most source code analysis tools take a more pragmatic and ad-hoc approach, and cut some corners to keep the analysis tractable without too much programmer guidance (in the form of type annotations), without too much computation (e.g., tools may refrain from doing a whole program analysis and only look for unwanted information flow within a single procedure), and without generating too many false positives. So in the end the analysis will often be both unsound and incomplete (i.e., it will have some false positives and some false negatives).

- Instead of the static approaches above, one can also do a **dynamic taint propagation**. Here tainted data is tagged at runtime, and during execution of the program these tags are propagated, and execution is halted if data that is tagged as tainted ends up in dangerous places, such as arguments to security-sensitive API calls. The idea of dynamic taint propagation was popularised by Perl's taint mode.

  Compared to a static analysis (either by a type system or a source code analysis tool), such a dynamic approach has the advantage of being simpler and being more precise. The obvious disadvantage is that it will only detect potential problems at the very last moment.

- Dynamic taint propagation can also be used to detect certain buffer overflow exploits, and combat the worms that plague operating systems. The basic idea is the same: untrusted input is tagged as tainted, these tags are propagated, and the system flags a warning if tainted data ends up in the program counter, is used as an instruction, or ends up in a critical argument of a security-sensitive system call [10]. Unlike a traditional signature-based approach to detect known exploits, this can even detect zero-day exploits. Such a dynamic taint propagation could even be pushed down to the hardware level, on a 65-bits machine where 1 bit is used to track tainting information, and the remaining bits are used for regular 64-bit data values. Unfortunately, there are limits to the type of exploits that can be stopped in this way [31].

Run-time monitoring for information flow

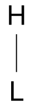
Information flow properties are harder to enforce by run-time monitoring than access control properties. Access control for some application can be enforced by run-time monitoring of its (input and output) behaviour and then intervening to stop any disallowed actions. For information flow this is not possible. For example, consider a run-time monitor for the HotelBooker app above, which tries to ensure that my credit card number is not leaked to anyone except the hotel. Let's assume my credit card number is 1234 5678 9012 3456. If the monitor sees this number appear, say as parameter in an http-request to `mafia.org`, the monitor could spot it and stop the request. But would a monitor also spot and stop such a request with the parameter ”6543 2109 8765 4321”? Worse, still, if the monitor spots the number 1 being communicated over the network, how can it possibly know if this is not leaking the first digit of my credit card number?

The only way a run-time monitor could know this is by tracking values passed around *inside* the application, and this is what the dynamic techniques for enforcing information flow policies mentioned above do.

5.3 Typing for information flow

This section discusses how information flow policies can be enforced by typing, for a toy language, in a sound way – more specifically, sound with respect to the formal characterisation of information flow using **non-interference.**

For simplicity, we just consider a lattice with two levels of secrecy: H(igh) and L(ow).



All program variables will have to be assigned a level (type). We will write the level of a program variable as a subscript, so for example we write $\mathtt{x}_t$ to mean that program variable $\mathtt{x}$ has level t. For a real programming language, the levels of program variables would have to be declared, for instance as part of the type declarations.

The type system involves **typing judgements** for expressions and for programs. A typing judgement for an expression is of the form $e : t$, that expression e has security level t. For expressions we have the following type derivation rules[1]

$$\text{(variable)} \quad \frac{}{x_t : t}$$

$$\text{(binary operations)} \quad \frac{e_1 : t \quad e_2 : t}{e_1 \oplus e_2 : t} \text{ where } \oplus \text{ is any binary operation.}$$

$$\text{(subtyping for expressions)} \quad \frac{e : t \quad t < t'}{e : t'}$$

In the subtyping rule, $<$ is the ordering on types induced by the lattice. For the simple lattice we consider, we just have $L < H$. This rule allows us to increase the secrecy level of any expression.

Instead of having a separate subtyping rule for increasing the secrecy level, as we have above, this could also be built into the other rules. For example, an alternative rule for binary operation would be

$$\frac{e_1 : t_1 \quad e_2 : t_2}{e_1 \oplus e_2 : t_1 \sqcup t_2}$$

where $t_1 \sqcup t_2$ is the least upper bound of t_1 and t_2 in the lattice. For example, for the lattice in Fig 5.1, 'Secret Belgium $\sqcup$ Secret Netherlands' would be 'Secret Benelux'. Note that this alternative rule is derivable from the rules for binary operations and subtyping given above. This alternative rule makes it clear that if we combine data from different secrecy levels, the combination gets the highest secrecy level of all the components. For example, a document

[1] These rules are given in the standard format for type derivation rules. In this format premises of a rule are listed above the horizontal line, and the conclusion below it. It can be read as an implication or an if-then statement: if the typing judgements above the line hold then we may conclude the typing judgement below it.

which contains secret information about Belgium and about the Netherlands would have to be classified ‘Secret Benelux’.

To rule out explicit information flows, we clearly want to disallow assignment of H-expressions to L-variables. For a typing judgement for programs of the form $p : \mathsf{ok}$, meaning that program p does not leak information, the type derivation rule for assignment would have to be

$$\frac{e : t}{x_t = e : \mathsf{ok}}$$

In combination with the subtyping rule for expressions given above, this rule also allows expressions e with a lower classification level than t to be assigned to a program variable x_t of level t.

If our programming language would include some output mechanism, say some `print`-statement, the type derivation rule might be

$$\frac{e : L}{\texttt{print}(e) : \mathsf{ok}}$$

so that only non-confidential information, i.e., information of level L, can be printed.

The type system should also rule out implicit flows, as in a program fragment

$$\texttt{if}\ e_1\ \texttt{then}\ x_l = e_2$$

where x_l is a low variable and $e_1 : H$; just requiring that e_2 has level L is not good enough for this.

In order to rule out implicit flows, the type system for programs should keep track of the levels of the variables that are assigned to in programs: typing judgements for programs, of the form $p : \mathsf{ok}\, t$, will mean that

> p does not leak information and does not assign to variables of levels lower than t.

The type derivation rules are then as follows

$$\begin{array}{rc}
\text{(assignment)} & \dfrac{e : t}{x_t = e : \mathsf{ok}\, t} \\[2ex]
\text{(if-then-else)} & \dfrac{e : t \quad p_1 : \mathsf{ok}\, t \quad p_2 : \mathsf{ok}\, t}{\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2 : \mathsf{ok}\, t} \\[2ex]
\text{(composition)} & \dfrac{p_1 : \mathsf{ok}\, t \quad p_2 : \mathsf{ok}\, t}{p_1; p_2 : \mathsf{ok}\, t} \\[2ex]
\text{(while)} & \dfrac{e : t \quad p : \mathsf{ok}\, t}{\texttt{while}\ (e)\{p\} : \mathsf{ok}\, t} \\[2ex]
\text{(subtyping for commands)} & \dfrac{p : \mathsf{ok}\, t \quad t' < t}{p : \mathsf{ok}\, t'}
\end{array}$$

The rule for assignment says that we can only assign values e of level t to variables x_t of level t; using the subtyping rule for expressions we can also store values of lower levels than t in a variable of level t.

Note that here the rule for `if-then-else` ensures that the level of variables assigned to in the then- or else-branch is equal to the level of the guard e. (In fact, it can also be higher, as we can use the subtyping rule for expressions to increase the level of the guard e; see Exercise 5.3.1 below.) This rules out unwanted implicit flows: assigning to a variable of a lower level than the guard would be an unwanted implicit information flow. Similarly, the rule for `while` forbids assignments to variables that have a lower level than the guard.

Also note that the subtyping rule for commands goes in the opposite direction as the one for expressions[2].

Important questions for the type system above, as for any type system, are:

- Is the type system **sound**? In other words: are well-typed programs guaranteed not to contain unwanted information flows?
- Is the type system **complete**? In other words: are all programs that do not contain unwanted information flows guaranteed to be typable?

The type system is not complete. Simple examples can show this: for example, suppose e_{high} is a high expression, and x_{low} is a low variable, then the following program lines are not well typed

$$x_{low} = e_{high} - e_{high};$$
$$\texttt{if}\ e_{high}\ \texttt{then}\ x_{low} = 7\ \texttt{else}\ x_{low} = 7;$$

even though they clearly do not leak information.

The type system above is sound. To prove this, we need some formal definition of what it means for a program not to contain unwanted information flows. The standard way to do this is by means of **non-interference**, which defines interference (or dependencies) between values of program values. For this we have to consider *program states*, which are vectors that assign a value to every program variable. For program states we can consider if they agree on the values for the low variables:

Definition 5.3.1 For program states μ and ν, we write $\mu \approx_{low} \nu$ iff μ and ν agree on low variables.

The idea behind this definition is that $\mu \approx_{low} \nu$ means that an observer who is only allowed to read low variables cannot tell the difference between states μ and ν.

Semantically, a program can be seen as a function that takes an initial program state as input and produces a final program state as output. Given the notion of $\approx_{low}$ we can now define the absence if information flow as follow:

Definition 5.3.2 (Non-interference) A program p does not leak information if, for all possible start states μ and ν such that $\mu \approx_{low} \nu$, whenever executing p in μ terminates and results in μ' and executing p in ν terminates and results in ν', then $\mu' \approx_{low} \nu'$.

The idea is that if in an initial state μ we change the values of one or more of the high (secret) variables then, after executing p, we cannot see any difference in the outcome as far as the low (public) variables are concerned. In other words, the values of high variables do not interfere with the execution as far as low variables are concerned. This means an observer who is only allowed to observe low variables cannot learn anything about the values of high values.

For this characterisation of (absence of) unwanted information flows we can now prove:

Theorem 5.3.1 (Soundness) If p : ok t then p is non-interferent, i.e., does not leak information from higher to lower levels.

[2] People familiar with the Bell-LaPadula system for mandatory access control with multiple levels of access can recognise the 'no read up' and the 'no write down' rules in the subtyping rules for expressions and commands, respectively.

The typing rules that rule out potential implicit flows are overly restrictive, in that they disallow many programs that do not leak. In other words, the type checker will produce a lot of false warnings. A practical way out here, used in the SPARTA approach [13], is to rely on manual analysis for these cases: the type system then only generates a warning for potential implicit flows, and these warnings have to be manually checked by a human auditor.

Exercises on typing for information flow and non-interference

Exercise 5.3.1 (Alternative rules) Fill in the ... below to give an alternative rule for if-then-else, which can be derived from the rule for if-then-else and the subtyping rule for expressions

$$\frac{e : t \quad p_1 : \mathsf{ok}\, t' \quad p_2 : \mathsf{ok}\, t' \quad \ldots}{\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2 : \ldots}$$

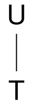
Now fill in the ... below to give an even more general rule, which can be derived if we also use the subtyping rule for commands

$$\frac{e : t \quad p_1 : \mathsf{ok}\, t_1 \quad p_2 : \mathsf{ok}\, t_2 \quad \ldots}{\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2 : \ldots}$$

Hint: here it is useful to use the notation $\sqcap$ for greatest lower bound.

Exercise 5.3.2 (Typing for integrity) As an exercise to see if you understand the idea of the type system above, define a type system for integrity instead of confidentiality.

For simplicity, we just consider a lattice with two levels of integrity: U(ntainted) and T(ainted).



Assume that all program variables x_t are labelled with an integrity level t.

First define rules for expressions e for judgements of the form $e : t$, which mean that expression e has integrity level t or higher.

Then define rules for programs p for judgements of the form $p : \mathsf{ok}t$, which mean that program p does not violate integrity (i.e., does not store tainted information in a variable of level U), and only stores information in variables of level t or lower.

Exercise 5.3.3 (Non-interference for integrity) As an exercise to see if you understand the idea of non-interference, define a notion of non-interference for integrity instead of confidentiality.

As part of this you will have to think about the appropriate notion of equivalence $\approx$ on program states that you need to express this.

Termination-sensitive information flow

The typing derivation rules above do not take non-termination into account as a hidden channel. For termination-sensitive information flow, the rules for `if-then-else` and `while` have to be more restrictive, namely:

$$\text{(if-then-else)} \quad \frac{e : L \quad p_1 : \mathsf{ok}\, t \quad p_2 : \mathsf{ok}\, t}{\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2 : \mathsf{ok}\, t}$$

$$\text{(while)} \quad \frac{e : L \quad p : \mathsf{ok}\, t}{\texttt{while}\ (e)\{p\} : \mathsf{ok}\, t}$$

The rule for `while` here excludes the possibility of a high (secret) condition e, as this may leak information. For example, the program `while(b){ skip }` will leak the value of `b` in the termination behaviour. Similarly, the rule for `if-then-else` excludes the possibility of a high (secret) guard, because this may leak information – namely in case p_1 terminates but p_2 does not, or vice versa. Note that these rules are very restrictive! It means it is impossible to branch on any conditions that involve high information.

With these rules the type system will again not be complete. To prove soundness, we now need a different characterisation of what it means for a program not to leak, which takes non-termination into account as a hidden channel:

Definition 5.3.3 (Termination-sensitive Non-interference) A program p is *termination-sensitive non-interferent* (i.e., does not leak information, not even through its termination behaviour) if, for all $\mu \approx_{low} \nu$, whenever executing p in μ terminates and results in μ', then executing p in ν also terminates and results in a state ν' for which $\mu' \approx_{low} \nu'$.

For the more restrictive rules for `if-then-else` and `while` we can now prove soundness.

Theorem 5.3.2 (Soundness) If $p : \mathrm{ok}\, t$ then p is termination-sensitive non-interferent, i.e., does not leak information from higher to lower levels, even through its termination behaviour.

More complicated notions of non-interference can be given to account for execution time as a hidden channel, or to define what unwanted information flow means for non-deterministic programs.

Growing suspicions

Looking back at the evolution of computing, we can see a steady increase in complexity in ever more fine-grained access control, in response to a steady decline in trust.

- Initially, software ran on the bare hardware, free to do anything it wanted.
- Then operating systems (and hardware) introduced a distinction between kernel and user mode, and began to enforce access control per user.
- At first all processes started by a user were trusted equally, and trusted as much as the user himself, so ran with all the user's privileges. Gradually options appeared to reduce privileges of individual processes, to face the fact that some applications should be trusted less than others, even when executed by one and the same user.
- Language-based access control is a next step in the evolution: different parts of a single program are trusted to a different degree, and hence executed with different permissions, as for instance enforced by the Java sandbox.
- The notion of information flow suggest a possible next step in the evolution: our trust in a process running on some computer might not only depend on (i) the user who started the process and (ii) the origin of the different parts of the code, but also on the origin of the input that we feed to it: the same piece of code, executed by the same user, should be trusted less when acting on untrusted input (say input obtained over the web) than when acting on trusted input (say input typed on the user's keyboard). It remains to be seen if such forms of access control will ever become mainstream.

  Note that Windows Office already does this: Word or Excel files that are downloaded over the internet or received as email attachments are opened *Protected View*, meaning that macros in these documents are disabled. This measure has been introduced because macros in such files are a notorious and simple way for attackers to gain access to systems.

Bibliography

[1] SEI CERT Coding Standards. Available from https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards.

[2] *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*. CERT, 2016. Available online at [1].

[3] *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*. Software Engineering Institute, Carnegie Mellon University, 2016. Available online at [1].

[4] Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010.

[5] Iván Arce, Kathleen Clark-Fisher, Neil Daswani, Jim DelGrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary McGraw, Brook Schoenfield, Margo Seltzer, Diomidis Spinellis, Izar Tarandach, and Jacob West. Avoiding the top 10 software security design flaws. Technical report, IEEE Computer Society Center for Secure Design (CSD), 2014.

[6] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.

[7] Cyber Safety Review Board. Review of the December 2021 Log4j event. Technical report, US Department of Homeland Security, 2022.

[8] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. *;login:*, pages 13–21, 2011.

[9] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[10] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 133–147. ACM, 2005.

[11] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[12] Ieu Eauvidoum and disk noise. Twenty years of escaping the Java sandbox. *Phrack magazine*, September 2018.

[13] Michael D Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, et al. Collaborative verification of information flow for a high-assurance app store. In *Computer and Communications Security (CCS'14)*, pages 1092–1104. ACM, 2014.

[14] J.-L. Lions et al. Ariane V Flight 501 failure - Enquiry Board report. Technical report, 1996. Available from http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[15] Brian Goetz. Java theory and practice: To mutate or not to mutate? immutable objects can greatly simplify your life. Technical report, 2003. Available from http://www-106.ibm.com/developerworks/java/library/j-jtp02183.html.

[16] Sudhakar Govindavajhala and Andrew W. Appel. Windows access control demystified. Technical report, Princeton University, 2006.

[17] M. Graff and K.R. Van Wyk. *Secure coding: principles and practices*. O'Reilly Media, 2003.

[18] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of Java exploitation. In *Computer and Communications Security (CCS'16)*, pages 779–790. ACM, 2016.

[19] Michael Howard, David LeBlanc, and John Viega. *The 24 deadly sins of software security*. McGraw-Hill, 2009.

[20] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30:235–269, 2003.

[21] Gary McGraw and Ed W. Felten. *Securing Java: getting down to business with mobile code*. Wiley Computer Pub., 1999. Available online at `http://www.securingjava.com`.

[22] Wojciech Mostowski and Erik Poll. Malicious code on Java Card smartcards: Attacks and countermeasures. In *CARDIS*, volume 5189 of *LNCS*, pages 1–16. Springer, 2008.

[23] Alan Mycroft. Programming language design and analysis motivated by hardware evolution. In *SAS'07*, number 3634 in LNCS, pages 18–33. Springer, 2007.

[24] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241. ACM, 1999. See also `http://www.cs.cornell.edu/jif/`.

[25] Yeoul Na. Keynote talk: '-fbounds-safety': Enforcing bounds safety for production C code, 2023. Presentation available at `https://www.youtube.com/watch?v=RK9bfrsMdAM`.

[26] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Symposium on OS Design and Implementation (OSDI'96)*, pages 229–244. USENIX, 1996.

[27] Meredith L. Patterson and Sergey Bratus. LangSec: Recognition, validation, and compositional correctness for real world security, 2013. USENIX Security BoF hand-out. Available from `http://langsec.org/bof-handout.pdf`.

[28] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[29] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[30] Fridtjof Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 94–103, 2007.

[31] Asia Slowinska and Herbert Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *SIGOPS EUROSYS'09*. ACM, 2009.

[32] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *Symposium on Security and Privacy (S&P)*, pages 1275–1295, 2019.

[33] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the Web tangled itself: Uncovering the history of client-side web (in) security. In *USENIX Security'17*, pages 971–987, 2017. Conference presentation available at `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/stock`.

[34] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[35] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Symposium on Security and Privacy (S&P)*, pages 48–62. IEEE, 2013.

[36] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. Addison-Wesley Professional, 2002.

[37] John Viega, Gary McGraw, Tom Mutdoseh, and Edward W. Felten. Statically scanning java code: Finding security vulnerabilities. *Software, IEEE*, 17(5):68–77, 2000. Almost identical content is available at `http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html`.

[38] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems (APSYS'12)*. ACM, 2012.