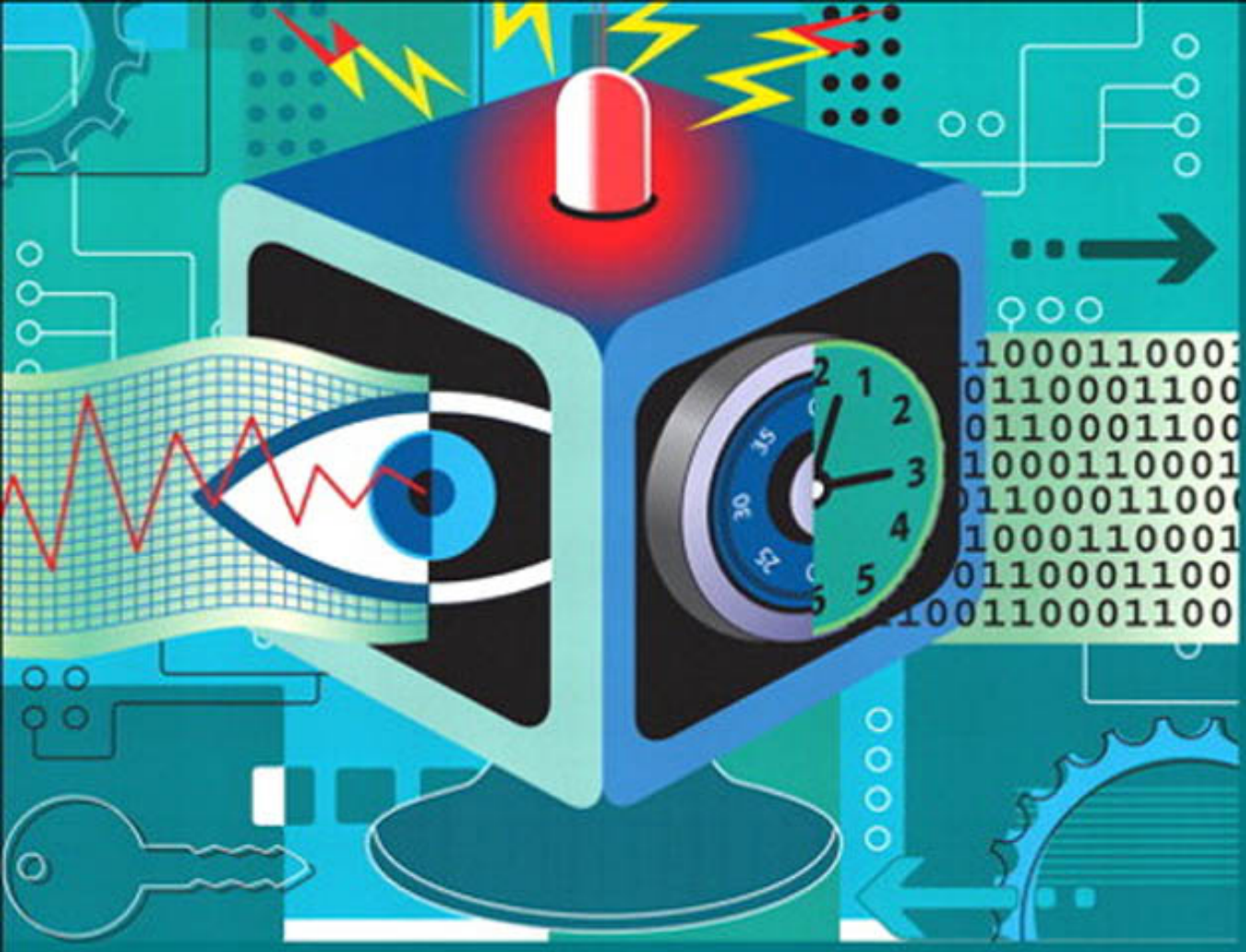

THE CRAFT OF SYSTEM SECURITY

Sean Smith
John Marchesini

The Craft of System Security

Perhaps. But try getting fire insurance if you ignore basic practices and current building codes! (And vilifying Morris only deflects attention from the main problem.)

6.5 Programming Language Security

Programming languages are the fundamental tool for building software. They give programmers a way to turn the concepts in their heads into a set of instructions that a machine can execute. Languages often succeed or fail on the basis of their power, expressiveness, and efficiency. But what about security? Could the safety and security of a system be determined by the language it's implemented in? What properties make a language secure? In this section, we look at these questions in some detail and examine some of the initial efforts to get answers.

A large number of security problems are the result of a programming error that, when exploited, puts the system in an unsafe state. So far in this chapter, we've been discussing specific types of errors that can lead to such security trouble as buffer and integer overflow, format strings, and so on. In this section, we look at the issue from a different perspective; we look beyond specific programming errors at the properties of the programming language itself.

6.5.1 Memory (Mis)management

Many of the scenarios we've been discussing in this chapter involve an attackers being able to get some sort of evil data into the target program's memory space. Protecting that memory space is vital to ensure the program's security. If we think of a program or a system as a collection of states and rules to transition through those states, then we see that the program's memory space holds the state of the program—the value of all the program variables, as well as the rules that govern how the program can transition through states.

A programming language that allows programs to access only intended memory locations is said to have *memory safety*. Without this feature, it is impossible or, perhaps, rather imaginative for a programming language to make any claims about being secure. A language without this feature can allow programs to arbitrarily read and write memory and thus gives attackers a means to alter the program state. A program written in a memory-safe language would never be able to overflow a buffer or write data to a memory location that it should not be able to.

Consider the C programming language for a moment; it is clearly not memory safe. As we discussed in Sections 6.1 and 6.2, programs written in C will gladly

copy input into portions of memory where they don't belong. This lack of memory safety is responsible for buffer overflow attacks, as well as the format string errors we discussed in Section 6.2 and the consequences of many integer overflow attacks.

Memory safety is a key language feature when it comes to considering the security of a particular programming language. Many newer languages, such as Java and C#, are memory safe. Trying to read or write to an inaccessible memory location in these languages causes the underlying runtime to throw an exception, possibly halting the program.

Although memory safety is necessary for security, it is not sufficient. Memory-safe languages can still have security problems, many of which are the result of bad system design, such as passing sensitive information in cleartext.

6.5.2 Type Safety

Another important consideration when evaluating the security of a programming language is *type safety*. Loosely speaking, a type-safe language will enforce that variables of a certain type can be assigned values only of that type. For instance, assume that we have a program variable y that is a pointer (that is, y has the type "pointer"). A type-safe language would allow us to assign only valid pointers to y, whereas a type-unsafe language would allow us to assign different types of values to y, possibly using a *type cast*, such as is often done in C programs.

So, what does type safety have to do with security? The answer is that type-safe languages are memory safe as well, and memory safety is a necessary property for security. To see why this relation is true, let's expand the previous example a little and assume that we are working in the C language, which is both type and memory unsafe. Since x is an integer, we could assign the value 123456 to x. Now, let's assume that we have some *pointer* variable y. Since C is type unsafe, we could assign x to y, essentially telling the system that y points to the address 123456. Listing 6.2 illustrates the scenario.

```
void foo ()
{
    int x;
    char *y;

    x = 123456;
    y = x;
}
```

Listing 6.2 A type-unsafe and memory-unsafe C function that generates a compiler warning

```
 void foo ()
{
    int x;
    char *y;

    x = 123456;
    y = (char*) x;
}
```

Listing 6.3 A type-unsafe and memory-unsafe C function that doesn't generate a compiler warning

When compiling the last line of code, the compiler will generate a warning that the assignment makes a pointer from an integer. The compiler's suggestion is to force the types to match via a *type cast*. Indeed, changing the program to Listing 6.3 makes the compiler happy.

Even in this case, the program is still memory unsafe. The moment that it tries to read or write data from y, the program will be reading or writing from the memory location 123456, which is just an arbitrary number, possibly input from the adversary. Without the ability to enforce type safety, such memory-safety violations are possible. (Furthermore, integer overflow vulnerabilities can essentially be blamed on a lack of type safety.)

It is also true that memory-unsafe languages cannot be type safe. To see why this is true, imagine that we have declared our variable x to be an integer again. Now, assume that we copy the appropriate number of bytes from memory into the value of x. How do we know that the bytes we've copied represent an integer? We don't, and since we can assign x's value from anywhere in memory, there's no way to enforce that x takes on only integer values. (Figure 6.6 shows the relationships between memory safety and type safety.)

If security is a driving force behind your choice of programming language, choose a type-safe language, if possible. Of course, using a type-safe language is no magic bullet; improper handling of sensitive data will make any system vulnerable, no matter what language it's written in. Using a type-safe language does free you from the buffer overflow issues of Section 6.1.

6.5.3 Information Flow

Another property that should be considered when evaluating a language's ability to construct secure systems is *information flow*. The idea here is that a language

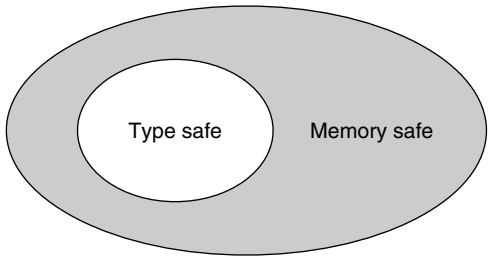

Figure 6.6 Type-safe languages are memory safe. Type-unsafe languages may or may not be memory safe. A memory-safe language could be type safe, although there is no guarantee, but a memory-unsafe language cannot.

should allow information and execution to flow only to appropriate portions of the system. This covers numerous aspects of the system: how information flows to external libraries and the underlying OS and its resources and how information flows between the objects and subsystems. (Recall from Chapter 2 the Old Testament focus on information flow.)

Thinking about information flow in the context of programming can raise some tricky issues. Suppose that we're concerned about standard multilevel security. Let `L` be a low-security variable and `H` a high-security variable. Clearly, information leaks if we assign `H` to `L`:

```
L ⟵ H
```

However, Listing 6.4 shows how information can leak in more subtle ways as well. Sabelfeld and Myers provide a thorough survey of such issues [SM03].

One of the fundamental flows of information that deserves consideration occurs at the boundary between the application and the underlying OS and/or core APIs. In some cases, applications need a certain privilege level in order to perform specific operations. At the extreme, programs can be *sandboxed:* run in a virtual

```
H ⟵ {0,1}
if (H == 0)
   L ⟵ 0
else
   L ⟵ 1
```

Listing 6.4 This code permits bad information flow even though the high-security information in `H` is never written directly to `L`

compartment such that it has the least amount of privileges necessary to accomplish its task. At the extreme, such support can be built into the OS itself, as in SELinux—see Section 4.5.2. A step down from using such an OS is to use a tool such as *systrace* to generate and enforce access policies for the OS's system call interface [Pro06]. Additionally, the Java and .NET runtimes have a policy mechanism that governs how various code packages can interact. Although none of these approaches are features of the programming language proper, they all serve to govern how the application interacts with other portions of the system.

Inside the application itself, the flow of information can be governed by access protections on portions of code. This is standard practice in *object-oriented (OO)* languages. Such languages often rely on special keywords—typically, `public`, `private`, and `protected`—to specify the access policy on a particular object. `Public` data and methods are accessible by any program entity, such as another object in the system; `protected` data and methods can be modified and invoked by subclasses of a particular class; and `private` data and methods can be modified and invoked only by a particular object itself.

Some languages use other keywords or properties to specify a particular policy on data members. For example, in C#, the `readonly` keyword applied to a data member disallows any program entity other than the object's constructor to assign a value to a variable. C/C++ programs can use the `const` keyword to achieve similar effects and more, since `const` can be applied to methods as well.

6.5.4 The Once and Future Silver Bullet

With all these advanced programming language tools, one might wonder why we still have insecure implementations, especially when it seemed that all the common implementation security blunders arose from the C programming language.

The answer is complex. A widely held belief in the systems programming community is that a low-level language like C is necessary for the required efficiency. Management of hardware—with bits, bytes, addresses, and such—requires data manipulation that does not easily lend itself to type safety. Just because a programming language will constrain behavior according to some type of policy (or type policy) doesn't imply that the policy itself is correct or is in fact easy or quick to write and maintain. Finally, many of these tools have yet to reach the level of reliability necessary for production-grade code.

Programming language researchers offer counterarguments against all these issues. Nonetheless, the Internet runs on C and C++ and will continue to do so for the foreseeable future.

6.5.5 Tools

What should the security artisan do? One approach is to continue to use C but augment it with additional protection tools. One can run *static analysis* tools, such as *Splint* or *ITS4* on the source code. (Trent Jaeger and his colleagues even used type-based static analysis to verify that SE/Linux made calls to the Linux Security Module at the right times [ZEJ02]). One can run *dynamic analysis* tools, such as *Purify* or *Valgrind*, at runtime. One can build with safer libraries and compilers to help defend against stack smashing via such techniques as canaries and address space randomization, discussed earlier.

Another approach is to move to a type-safe variant of C, such as *CCured*, *Cyclone*, or *Clay*. (Unfortunately, when we surveyed the field, it appeared that these tools were not yet ready for production use.) Or, one could listen to the evangelists and move to a more advanced language, such as Java or OCaml. (See [DPS07] for a longer survey of these techniques.)

6.6 Security in the Development Lifecycle

It's easy to wonder why the software industry as a whole allows products with the types of flaws we've been discussing to make their way into customers' hands. It's much more difficult to find a good answer. If we compare the software industry to the automobile industry, for instance, it would be a fair conclusion that the software industry gets away with murder in comparison. For example, automobiles don't get sold with known defects. The price to "patch" units in the field is much higher for cars; it's called a recall. So, why is that we have come to terms with the fact that software will have security problems but are unwilling to accept the fact that our car may shut down at any moment? More important, why has the software industry come to terms with that? Put simply, the software industry hasn't had to make secure products and hasn't really wanted to—it's hard.

6.6.1 The Development Lifecycle

We won't discuss the issue of legal liability in this section but do discuss the difficulty of making a secure product from a process point of view. To review, the traditional software development lifecycle comprises five phases.

1. In the *requirements* phase, someone sits and talks to the customer in an effort to try and understand the customer's problem and what the requirements for a good solution are.