CyBOK

SOFTWARE SECURITY

KNOWLEDGE AREA

Issue 1.0

AUTHOR: Frank Piessens – KU Leuven

EDITOR: Awais Rashid – University of Bristol

REVIEWERS:

Eric Bodden – Paderborn University

Rod Chapman – Altran UK

Michael Hicks – University of Maryland

Jacques Klein – University of Luxembourg

Andrei Sabelfeld – Chalmers University of Technology

The CyBOK project would like to understand how the CyBOK is being used and its uptake. The project would like organisations using, or intending to use, CyBOK for the purposes of education, training, course development, professional development etc. to contact it at contact@cybok.org to let the project know how they are using CyBOK.

Issue 1.0 is a stable public release of the Software Security Knowledge Area. However, it should be noted that a fully-collated CyBOK document which includes all of the Knowledge Areas is anticipated to be released by the end of July 2019. This will likely include updated page layout and formatting of the individual Knowledge Areas.

Software Security

Frank Piessens

June 2018

INTRODUCTION

The purpose of this Software Security chapter is to provide a structured overview of known categories of software implementation vulnerabilities, and of techniques that can be used to prevent or detect such vulnerabilities, or to mitigate their exploitation. This overview is intended to be useful to academic staff for course and curricula design in the area of software security, as well as to industry professionals for the verification of skills and the design of job descriptions in this area.

Let us start by defining some terms and concepts, and by defining the scope of this chapter. A first key issue is what it means for software to be *secure*? One possible definition is that a software system is secure if it satisfies a specified or implied security objective. This security objective specifies *confidentiality*, *integrity* and *availability* requirements[1] for the system's data and functionality. Consider, for instance, a social networking service. The security objective of such a system could include the following requirements:

- Pictures posted by a user can only be seen by that user's friends (confidentiality)
- A user can like any given post at most once (integrity)
- The service is operational more than 99.9% of the time on average (availability)

Different security requirements can be at odds with each other, for instance, locking down a system on the appearance of an attack is good for confidentiality and integrity of the system, but bad for availability.

A *security failure* is a scenario where the software system does not achieve its security objective, and a *vulnerability* is the underlying cause of such a failure. The determination of an underlying cause is usually not absolute: there are no objective criteria to determine *what* vulnerability is responsible for a given security failure or *where* it is located in the code. One might say that the vulnerability is in the part of the code that has to be fixed to avoid this specific security failure, but fixes can be required in multiple places, and often multiple mitigation strategies are possible where each mitigation strategy requires a different fix or set of fixes.

The definitions of "security" and "vulnerability" above assume the existence of a security objective. In practice however, most software systems do not have precise explicit security objectives, and even if they do, these objectives are not absolute and have to be traded off against other objectives such as performance or usability of the software system. Hence, software security is often about avoiding known classes of bugs that enable specific attack techniques. There are well-understood classes of software implementation bugs that, when triggered by an attacker, can lead to a substantial disruption in the behaviour of the software, and are thus likely to break whatever security objective the software

[1] Other common information security requirements like non-repudiation or data authentication can be seen as instances or refinements of integrity from a software perspective. But from other perspectives, for instance from a legal perspective, the semantics of these requirements can be more involved.

might have. These bugs are called *implementation vulnerabilities* even if they are relatively independent from application- or domain-specific security objectives like the example objectives above.

This document, the Software Security KA, covers such implementation vulnerabilities, as well as countermeasures for them. Many other aspects are relevant for the security of software based systems, including human factors, physical security, secure deployment and procedural aspects, but they are not covered in this chapter. The impact of security on the various phases of the software lifecycle is discussed in the Secure Software Lifecycle KA. Security issues specific to software running on the web or mobile platforms are discussed in the Web and Mobile Security KA.

The remainder of this chapter is structured as follows. Topic 1 (Categories) discusses widely relevant categories of implementation vulnerabilities, but without the ambition of describing a complete taxonomy. Instead, the topic discusses how categories of vulnerabilities can often be defined as violations of a partial specification of the software system, and it is unlikely that a useful complete taxonomy of such partial specifications would exist. The discussion of countermeasures for implementation vulnerabilities is structured in terms of where in the lifecycle of the software system they are applicable. Topic 2 (Prevention) discusses how programming language and Application Programming Interface (API) design can prevent vulnerabilities from being introduced during development in software programmed in that language and using that API. In addition, defensive coding practices can contribute to the prevention of vulnerabilities. Topic 3 (Detection) covers techniques to detect vulnerabilities in existing source code, for instance, during development and testing. Topic 4 (Mitigation) discusses how the impact of remaining vulnerabilities can be mitigated at runtime. It is important to note, however, that some countermeasure techniques could in principle be applied in all three phases, so this is not an orthogonal classification. For instance, a specific dynamic check (say, an array bounds check) could be mandated by the language specification (Prevention, the countermeasure is built in by the language designer), could be used as a testing oracle (Detection, the countermeasure is used by the software tester) or could be inlined in the program to block attacks at run-time (Mitigation, the countermeasure is applied on deployment).

CONTENT

1 Categories of Vulnerabilities

[1][2, c4,c5,c6,c7,c10,c11][3, c6,c9] [4, c17][5, c5,c9,c11,c13,c17]

As discussed in the Introduction, we use the term *implementation vulnerability* (sometimes also called a *security bug*) both for bugs that make it possible for an attacker to violate a security objective, as well as for classes of bugs that enable specific attack techniques.

Implementation vulnerabilities play an important role in cybersecurity and come in many forms. The Common Vulnerabilities and Exposures (CVE) is a publicly available list of entries in a standardised form describing vulnerabilities in widely-used software components, and it lists close to a hundred thousand such vulnerabilities at the time of writing. Implementation vulnerabilities are often caused by insecure programming practices and influenced by the programming language or APIs used by the developer. This first topic covers important categories of implementation vulnerabilities that can be attributed to such insecure programming practices.

Existing classifications of vulnerabilities, such as the Common Weakness Enumeration (CWE), a community-developed list of vulnerability categories, are useful as a baseline for vulnerability identification, mitigation and prevention, but none of the existing classifications have succeeded in coming up with a complete taxonomy. Hence, the categories discussed in this first topic should be seen as examples of important classes of vulnerabilities, and not as an exhaustive list. They were selected with the intention to cover the most common implementation vulnerabilities, but this selection is at least to some extent subjective.

Specific categories of implementation vulnerabilities can often be described as violations of a (formal or informal) specification of some sub-component of the software system. Such a specification takes the form of a contract that makes explicit what the sub-component expects of, and provides to its clients. On violation of such a contract, the software system enters an error-state, and the further behaviour of the software system is typically behaviour that has not been considered by the system developers and is dependent on system implementation details. Attackers of the system can study the implementation details and exploit them to make the system behave in a way that is desirable for the attacker.

1.1 Memory Management Vulnerabilities

Imperative programming languages support mutable state, i.e., these languages have constructs for allocating memory cells that can subsequently be assigned to, or read from by the program, and then deallocated again. The programming language definition specifies how to use these constructs correctly: for instance, allocation of n memory cells will return a reference to an array of cells that can then be accessed with indices 0 to $n - 1$ until the reference is deallocated (freed) again. This specification can be seen as a contract for the memory management sub-component. Some programming languages implement this contract defensively, and will throw an exception if a client program accesses memory incorrectly. Other programming languages (most notably, C and C++) leave the responsibility for correctly allocating, accessing and deallocating memory in the hands of the programmer, and say that the behaviour of programs that access or manage memory incorrectly is *undefined*. Such languages are sometimes called *memory unsafe* languages, and bugs related to memory management (*memory management vulnerabilities*) are a notorious source of security bugs in these languages.

- A *spatial* vulnerability is a bug where the program is indexing into a valid contiguous range of memory cells, but the index is out-of-bounds. The archetypical example is a *buffer overflow vulnerability* where the program accesses an array (a buffer) with an out-of-bounds index.
- A *temporal* vulnerability is a bug where the program accesses memory that was once allocated to the program, but has since been deallocated. A typical example is dereferencing a dangling pointer.

The C and C++ language specifications leave the behaviour of a program with a memory management vulnerability *undefined*. As such, the observed behaviour of a program with a vulnerability will depend on the actual implementation of the language. Memory management vulnerabilities are particularly dangerous from a security point of view, because in many implementations mutable memory cells allocated to the program are part of the same memory address space where also compiled program code, and runtime metadata such as the call stack are stored. In such implementations, a memory access by the program that violates the memory management contract can result in an access to compiled program code or runtime metadata, and hence can cause corruption of program code, program control flow and program data. There exists a wide range of powerful attack techniques to exploit memory management vulnerabilities [1].

An attack consists of providing input to the program to trigger the vulnerability, which makes the program violate the memory management contract. The attacker chooses the input such that the program accesses a memory cell of interest to the attacker:

- In a *code corruption attack*, the invalid memory access modifies compiled program code to attacker specified code.
- In a *control-flow hijack attack*, the invalid memory access modifies a code pointer (for instance, a return address on the stack, or a function pointer) to make the processor execute attacker-provided code (a *direct code injection attack*), or to make the processor reuse existing code of

the program in unexpected ways (a *code-reuse attack*, also known as an *indirect code injection attack*, such as a *return-to-libc attack*, or a *return-oriented-programming attack*).

- In a *data-only attack*, the invalid memory access modifies other data variables of the program, possibly resulting in increased privileges for the attacker.
- In an *information leak attack*, the invalid memory access is a read access, possibly resulting in the exfiltration of information, either application secrets such as cryptographic keys, or runtime metadata such as addresses which assist prediction of the exact layout of memory and hence may enable other attacks.

Because of the practical importance of these classes of attacks, mitigation techniques have been developed that counter specific attack techniques, and we discuss these in Topic 4.

1.2 Structured Output Generation Vulnerabilities

Programs often have to dynamically construct structured output that will then be consumed by another program. Examples include: the construction of SQL queries to be consumed by a database, or the construction of HTML pages to be consumed by a web browser. One can think of the code that generates the structured output as a sub-component. The intended structure of the output, and how input to the sub-component should be used within the output, can be thought of as a contract to which that sub-component should adhere. For instance, when provided with a name and password as input, the intended output is a SQL query that selects the user with the given name and password from the users database table.

A common insecure programming practice is to construct such structured output by means of string manipulation. The output is constructed as a concatenation of strings where some of these strings are derived (directly or indirectly) from input to the program. This practice is dangerous, because it leaves the intended structure of the output string implicit, and maliciously chosen values for input strings can cause the program to generate unintended output. For instance, a programmer can construct a SQL query as:

```
query = "select * from users where name='" + name +"'" and pw = '" + password + "'"
```

with the intention of constructing a SQL query that checks for name and password in the where clause. However, if the `name` string is provided by an attacker, the attacker can set name to `"John' --"`, and this would remove the password check from the query (note that `--` starts a comment in SQL).

A *structured output generation vulnerability* is a bug where the program constructs such unintended output. This is particularly dangerous in the case where the structured output represents *code* that is intended to include provided input as *data*. Maliciously chosen input data can then influence the generated output code in unintended ways. These vulnerabilities are also known as *injection vulnerabilities* (e.g., SQL injection, or script injection). The name 'injection' refers to the fact that exploitation of these vulnerabilities will often provide data inputs that cause the structured output to contain additional code statements, i.e. exploitation *injects* unintended new statements in the output. Structured output generation vulnerabilities are relevant for many different kinds of structured outputs:

- A *SQL injection vulnerability* is a structured output generation vulnerability where the structured output consists of SQL code. These vulnerabilities are particularly relevant for server-side web application software, where it is common for the application to interact with a back-end database by constructing queries partially based on input provided through web forms.
- A *command injection vulnerability* is a structured output generation vulnerability where the structured output is a shell command sent by the application to the operating system shell.

- A *script injection vulnerability*, sometimes also called a *cross-site scripting (XSS) vulnerability* is a structured output generation vulnerability where the structured output is JavaScript code sent to a web browser for client-side execution.

This list is by no means exhaustive. Other examples include: XPath injection, HTML injections, CSS injection, PostScript injection and many more.

Several factors can contribute to the difficulty of avoiding structured output generation vulnerabilities:

- The structured output can be in a language that supports sublanguages with a significantly different syntactic structure. An important example of such a problematic case is HTML, that supports sublanguages such as JavaScript, CSS and SVG.
- The computation of the structured output can happen in different phases with outputs of one phase being stored and later retrieved as input for a later phase. Structured output generation vulnerabilities that go through multiple phases are sometimes referred to as *stored injection vulnerabilities*, or more generally as *higher-order injection vulnerabilities*. Examples include stored cross-site scripting and higher-order SQL injection.

Attack techniques for exploiting structured output generation vulnerabilities generally depend on the nature of the structured output language, but a wide range of attack techniques for exploiting SQL injection or script injection are known and documented.

The KA on Web and Mobile Security provides a more detailed discussion of such attack techniques.

1.3 Race Condition Vulnerabilities

When a program accesses resources (such as memory, files or databases) that it shares with other concurrent actors (other threads in the same process, or other processes), the program often makes assumptions about what these concurrent actors will do (or not do) to these shared resources.

Such assumptions can again be thought of as part of a specification of the program. This specification is no longer a contract between two sub-components of the program (a caller and a callee), but it is a contract between the actor executing the program and its environment (all concurrent actors), where the contract specifies the assumptions made on how the environment will interact with the program's resources. For instance, the specification can say that the program relies on exclusive access to a set of resources for a specific interval of its execution: only the actor executing the program will have access to the set of resources for the specified interval.

Violations of such a specification are *concurrency bugs*, also commonly referred to as *race conditions*, because a consequence of these bugs is that the behaviour of the program may depend on which concurrent actor accesses a resource first ('wins a race'). Concurrency, and the corresponding issues of getting programs correct in the presence of concurrency, is an important sub-area of computer science with importance well beyond the area of cybersecurity [6].

But concurrency bugs can be security bugs, too. Concurrency bugs often introduce non-determinism: the behaviour of a program will depend on the exact timing or interleaving of the actions of all concurrent actors. In adversarial settings, where an attacker controls some of the concurrent actors, the attacker may have sufficient control on the timing of actions to influence the behaviour of the program such that a security objective is violated. A *race condition vulnerability* is a concurrency bug with such security consequences. A very common instance is the case where the program checks a condition on a resource, and then relies on that condition when using the resource. If an attacker can interleave his/her own actions to invalidate the condition between the check and the time of use, this is called a time-of-check to time-of-use (TOCTOU) vulnerability.

Race condition vulnerabilities are relevant for many different types of software. Two important areas where they occur are:

- Race conditions on the file system: privileged programs (i.e., programs that run with more privileges than their callers, for instance, operating system services) often need to check some condition on a file, before performing an action on that file on behalf of a less privileged user. Failing to perform check and action atomically (such that no concurrent actor can intervene) is a race condition vulnerability: an attacker can invalidate the condition between the check and the action.
- Races on the session state in web applications: web servers are often multi-threaded for performance purposes, and consecutive HTTP requests may be handled by different threads. Hence, two HTTP requests belonging to the same HTTP session may access the session state concurrently. Failing to account for this is a race condition vulnerability that may lead to corruption of the session state.

1.4 API Vulnerabilities

An Application Programming Interface, or API, is the interface through which one software component communicates with another component, such as a software library, operating system, web service, and so forth. Almost all software is programmed against one or more pre-existing APIs. An API comes with an (explicit or implicit) specification/contract of how it should be used and what services it offers, and just like the contracts we considered in previous subsections, violations of these contracts can often have significant consequences for security. If the client of the API violates the contract, the software system again enters an error-state, and the further behaviour of the software system will depend on implementation details of the API, and this may allow an attacker to break the security objective of the overall software system. This is essentially a generalisation of the idea of *implementation vulnerabilities as contract violations* from subsections 1.1, 1.2 and 1.3 to arbitrary API contracts.

Of course, some APIs are more security sensitive than others. A broad class of APIs that are security sensitive are APIs to libraries that implement security functionality like cryptography or access control logic. Generally speaking, a software system must use all the 'security components' that it relies on in a functionally correct way, or it is likely to violate a security objective. This is particularly challenging for cryptographic libraries: if a cryptographic library offers a flexible API, then *correct* use of that API (in the sense that a given security objective is achieved) is known to be hard. There is substantial empirical evidence [7] that developers frequently make mistakes in the use of cryptographic APIs, thus introducing vulnerabilities.

An orthogonal concern to secure *use* is the secure *implementation* of the cryptographic API. Secure implementations of cryptography are covered in the Cryptography KA.

1.5 Side-channel Vulnerabilities

The execution of a program is ultimately a physical process, typically involving digital electronic circuitry that consumes power, emits electro-magnetic radiation, and takes time to execute to completion. It is common, however, in computer science to model the execution of programs abstractly, in terms of the execution of code on an abstract machine whose semantics is defined mathematically (with varying levels of rigour). In fact, it is common to model execution of programs at many different levels of abstraction, including, for instance, execution of assembly code on a specified Instruction Set Architecture (ISA), execution of Java bytecode on the Java Virtual Machine, or execution of Java source code according to the Java language specification. Each subsequent layer of abstraction is implemented in terms of a lower layer, but abstracts from some of the effects or behaviours of that lower layer. For instance, an ISA makes abstraction from some physical effects such as electromagnetic radiation or power consumption, and the Java Virtual Machine abstracts from the details of memory management.

A *side-channel* is an information channel that communicates information about the execution of a software program by means of such effects from which the program's code abstracts. Some side-channels require physical access to the hardware executing the software program. Other side-channels, sometimes called *software-based side-channels* can be used from software running on the same hardware as the software program under attack.

Closely related to side-channels are *covert* channels. A covert channel is an information channel where the attacker also controls the program that is leaking information through the side-channel, i.e., the attacker uses a side-channel to purposefully exfiltrate information.

Side-channels play an important role in the field of cryptography, where the abstraction gap between (1) the mathematical (or source code level) description of a cryptographic algorithm and (2) the physical implementation of that algorithm, has been shown to be relevant for security [8]. It was demonstrated that, unless an implementation carefully guards against this, side-channels based on power consumption or execution time can easily leak the cryptographic key used during the execution of an encryption algorithm. This breaks the security objectives of encryption for an attacker model where the attacker can physically monitor the encryption process. Side-channel attacks against cryptographic implementations (and corresponding countermeasures) are discussed in the cryptography KA.

But side-channels are broadly relevant to software security in general. Side-channels can be studied for any scenario where software is implemented in terms of a lower-layer abstraction, even if that lower-layer abstraction is itself not yet a physical implementation. An important example is the implementation of a processor's instruction set architecture (ISA) in terms of a micro-architecture. The execution of assembly code written in the ISA will have effects on the micro-architectural state; for instance, an effect could be that some values are copied from main memory to a cache. The ISA makes abstraction of these effects, but under attacker models where the attacker can observe or influence these micro-architectural effects, they constitute a side-channel.

Side-channels, and in particular software-based side-channels, are most commonly a confidentiality threat: they leak information about the software's execution to an attacker monitoring effects at the lower abstraction layer. But side-channels can also constitute an integrity threat in case the attacker can modify the software's execution state by relying on lower layer effects. Such attacks are more commonly referred to as *fault injection* attacks. Physical fault-injection attacks can use voltage or clock glitching, extreme temperatures, or electromagnetic radiation to induce faults. Software-based fault-injection uses software to drive hardware components of the system outside their specification range with the objective of inducing faults in these components. A famous example is the Rowhammer attack that uses maliciously crafted memory access patterns to trigger an unintended interaction between high-density DRAM memory cells that causes memory bits to flip.

1.6 Discussion

Better connection with overall security objectives needs more complex specifications. We have categorised implementation vulnerabilities as violations of specific partial specifications of software components. However, the connection to the security objective of the overall software system is weak. It is perfectly possible that a software system has an implementation vulnerability, but that it is not exploitable to break a security objective of the system, for instance, because there are redundant countermeasures elsewhere in the system. Even more so, if a software system does not have any of the implementation vulnerabilities we discussed, it may still fail its security objective.

To have stronger assurance that the software system satisfies a security objective, one can formalise the security objective as a specification. During the design phase, on decomposition of the system in sub-components, one should specify the behaviour of the sub-components such that they jointly imply the specification of the overall system. With such a design, the connection between an implementation vulnerability as a violation of a specification on the one hand, and the overall security

objective of the system on the other, is much stronger.

It is important to note, however, that specifications would become more complex and more domain-specific in such a scenario. We discuss one illustration of additional complexity. For the vulnerability categories we discussed (memory management, structured output generation, race conditions and API vulnerabilities), the corresponding specifications express properties of single executions of the software: a given execution either satisfies or violates the specification, and the software has a vulnerability as soon as there exists an execution that violates the specification.

There are, however, software security objectives that cannot be expressed as properties of individual execution traces. A widely studied example of such a security objective is *information flow security*. A baseline specification of this security objective for deterministic sequential programs goes as follows: label the inputs and outputs of a program as either public or confidential, and then require that no *two* executions of the software with the same public inputs (but different confidential inputs) have different public outputs. The intuition for looking at pairs of executions is the following: it might be that the program does not leak confidential data directly but instead leaks some partial information about this data. If collected along multiple runs, the attacker can gather so much information that eventually relevant parts of the confidential original data are, in fact, leaked. The above specification effectively requires that confidential inputs can never influence public outputs in any way, and hence cannot leak even partial information. In a dual way, one can express integrity objectives by requiring that low-integrity inputs can not influence high-integrity outputs.

But an information flow specification is more complex than the specifications we considered in previous sections because one needs *two* executions to show a violation of the specification. *Information leak vulnerabilities* are violations of a (confidentiality-oriented) information flow policy. They can also be understood as violations of a specification, but this is now a specification that talks about multiple executions of the software system. This has profound consequences for the development of countermeasures to address these vulnerabilities [9].

Side channel vulnerabilities are different. Side channel vulnerabilities are by definition not violations of a specification at the abstraction level of the software source code: they intrinsically use effects from which the source code abstracts. However, if one develops a model of the execution infrastructure of the software that is detailed enough to model side channel attacks, then side channel vulnerabilities can again be understood as violations of a partial specification. One can choose to locate the vulnerability in the execution infrastructure by providing a specification for the execution infrastructure that says that it should not introduce additional communication mechanisms. This is essentially what the theory of full abstraction [10] requires. Alternatively, one can refine the model of the source code language to expose the effects used in particular side channel attacks, thus making it possible to express side-channel vulnerabilities at the source code level. Dealing with general software side-channel vulnerabilities is not yet well understood, and no generally applicable realistic countermeasures are known. One can, of course, isolate the execution, i.e., prevent concurrent executions on the same hardware, but that then contradicts other goals such as optimised hardware utilisation.

Vulnerabilities as faults. The classification of vulnerabilities by means of the specification they violate is useful for understanding relevant classes of vulnerabilities, but is not intended as a complete taxonomy: there are a very large number of partial specifications of software systems that contribute to achieving some security objective. Vulnerabilities can, however, be seen as an instance of the concept of *faults*, studied in the field of dependable computing, and a good taxonomy of faults has been developed in that field [11].

2 Prevention of Vulnerabilities

[12, 13, 14] [15, c3]

Once a category of vulnerabilities is well understood, an important question is how the introduction of such vulnerabilities in software can be prevented or at least be made less likely. The most effective approaches eradicate categories of vulnerabilities by design of the programming language or API.

The general idea is the following. We have seen in Topic 1 that many categories of implementation vulnerabilities can be described as violations of a specification of some sub-component. Let us call an execution of the software system that violates this specification, an *erroneous execution*, or an execution with an error. From a security point of view, it is useful to distinguish between errors that cause the immediate termination of the execution (*trapped errors*), and errors that may go unnoticed for a while (*untrapped errors*) [13]. Untrapped errors are particularly dangerous, because the further behaviour of the software system after an untrapped error can be arbitrary, and an attacker might be able to steer the software system to behaviour that violates a security objective. Hence, designing a language or API to avoid errors, and in particular untrapped errors, is a powerful approach to prevent the presence of vulnerabilities. For instance, languages like Java effectively make it impossible to introduce memory management vulnerabilities: a combination of static and dynamic checks ensures that no untrapped memory management errors can occur. This effectively protects against the attack techniques discussed in 1.1. It is, however, important to note that this does not prevent the presence of memory-management *bugs*: a program can still access an array out of bounds. But the bug is no longer a *vulnerability*, as execution is terminated immediately when such an access occurs. One could argue that the bug is still a vulnerability if one of the security objectives of the software system is *availability*, including the absence of unexpected program termination.

In cases where choice or redesign of the programming language or API itself is not an option, specific categories of vulnerabilities can be made less likely by imposing safe coding practices.

This topic provides an overview of these techniques that can prevent the introduction of vulnerabilities.

2.1 Language Design and Type Systems

A programming language can prevent categories of implementation vulnerabilities that can be described as violations of a specification by:

1. making it possible to express the specification within the language, and
2. ensuring that there can be no untrapped execution errors with respect to the expressed specification.

Memory management vulnerabilities. A programming language specification inherently includes a specification of all the memory allocation, access and deallocation features provided by that language. Hence, the specification of the memory management sub-component is always available. A programming language is called *memory-safe* if the language definition implies that there can be no untrapped memory management errors. Languages like C or C++ are not memory-safe because the language definition allows for implementations of the language that can have untrapped memory management errors, but even for such languages one can build specific *implementations* that are memory-safe (usually at the cost of performance).

A language can be made memory-safe through a combination of:

1. the careful selection of the features it supports: for instance, languages can choose to avoid mutable state, or can choose to avoid dynamic memory allocation, or can choose to avoid manual deallocation by relying on garbage collection,
2. imposing dynamic checks: for instance, imposing that every array access must be bounds-checked, and

3. imposing static checks, typically in the form of a static type system: for instance, object-field access can be guaranteed safe by means of a type system.

Programming languages vary widely in how they combine features, dynamic and static checks. Pure functional languages like Haskell avoid mutable memory and rely heavily on static checks and garbage collection. Dynamic languages like Python rely heavily on dynamic checks and garbage collection. Statically typed object-oriented languages like Java and C# sit between these two extremes. Innovative languages like SPARK (a subset of Ada) [16] and Rust achieve memory safety without relying on garbage collection. Rust, for instance, uses a type system that allows the compiler to reason about pointers statically, thus enabling it to insert code to free memory at places where it is known to no longer be accessible. This comes at the expense of some decreased flexibility when it comes to structuring program code.

Structured output generation vulnerabilities. An important cause for structured output generation vulnerabilities is that the programmer leaves the intended structure of the output implicit, and computes the structured output by string manipulation. A programming language can help prevent such vulnerabilities by providing language features that allow the programmer to make the intended structure explicit, thus providing a specification. The language implementation can then ensure that no untrapped errors with respect to that specification are possible.

A first approach is to provide a type system that supports the description of structured data. This approach has been worked out rigorously for XML data: the programming language supports XML documents as first class values, and *regular expression types* [17] support the description of the structure of XML documents using the standard regular expression operators. A type-correct program that outputs an XML document of a given type is guaranteed to generate XML output of the structure described by the type.

A second approach is to provide primitive language features for some of the common use cases of structured output generation. Language Integrated Query (LINQ) is an extension of the C# language with syntax for writing query expressions. By writing the query as an expression (as opposed to building a SQL query by concatenating strings), the intended structure of the query is explicit, and the LINQ provider that compiles the query to SQL can provide strong guarantees that the generated query has the intended structure.

Race condition vulnerabilities. Race condition vulnerabilities on heap allocated memory are often enabled by *aliasing*, the existence of multiple pointers to the same memory cell. If two concurrent threads both hold an alias to the same cell, there is the potential of a race condition on that cell. The existence of aliasing also leads to temporal memory-management vulnerabilities, when memory is deallocated through one alias but then accessed through another alias. The notion of *ownership* helps mitigate the complications that arise because of aliasing. The essence of the idea is that, while multiple aliases to a resource can exist, only one of these aliases is the *owner* of the resource, and some operations can only be performed through the owner. An *ownership regime* puts constraints on how aliases can be created, and what operations are allowed through these aliases. By doing so, an ownership regime can prevent race condition vulnerabilities, or it can support automatic memory management without a garbage collector. For instance, a simple ownership regime for heap allocated memory cells might impose the constraints that: (1) aliases can only be created if they are guaranteed to go out of scope before the owner does, (2) aliases can only be used for reading, and (3) the owner can write to a cell only if no aliases currently exist. This simple regime avoids data races: there can never be a concurrent read and write on the same cell. It also supports automatic memory management without garbage collection: a heap cell can be deallocated as soon as the owner goes out of scope. Of course, this simple regime is still quite restrictive, and a significant body of research exists on designing less restrictive ownership regimes that can still provide useful guarantees.

An ownership regime can be enforced by the programming language by means of a type system,

and several research languages have done this with the objective of preventing data races or memory management vulnerabilities. The Rust programming language, a recent systems programming language, is the first mainstream language to incorporate an ownership type system.

Other vulnerabilities. Many other categories of vulnerabilities can, in principle, be addressed by means of programming language design and static type checking. There is, for instance, a wide body of research on language-based approaches to enforce information flow security [18]. These approaches have until now mainly been integrated in research prototype languages. SPARK is an example of a real-world language that has implemented information flow analysis in the compiler. Language-based information flow security techniques have also had a profound influence on the static detection techniques for vulnerabilities (Topic 3).

2.2 API Design

The development of software not only relies on a programming language, it also relies on APIs, implemented by libraries or frameworks. Just like language design impacts the likelihood of introducing vulnerabilities, so does API design. The base principle is the same: the API should be designed to avoid execution errors (where now, execution errors are violations of the API specification), and in particular *untrapped* execution errors. It should be difficult for the programmer to violate an API contract, and if the contract is violated, that should be trapped, leading, for instance, to program termination or to well-defined error-handling behaviour.

Where the programming language itself does not prevent a certain category of vulnerabilities (e.g. C does not prevent memory-management vulnerabilities, Java does not prevent race conditions or structured output generation vulnerabilities), the likelihood of introducing these vulnerabilities can be reduced by offering a higher-level API:

- Several libraries providing less error-prone APIs for memory management in C or C++ have been proposed. These libraries offer fat pointers (where pointers maintain bounds information and check whether accesses are in bound), garbage collection (where manual deallocation is no longer required), or smart pointers (that support an ownership-regime to safely automate deallocation).
- Several libraries providing less error-prone APIs to do structured output generation for various types of structured output and for various programming languages have been proposed. Examples include Prepared Statement APIs that allow a programmer to separate the structure of a SQL statement from the user input that needs to be plugged into that structure, or library implementations of language integrated query, where query expressions are constructed using API calls instead of using language syntax.
- Several libraries providing less error-prone APIs to cryptography have been proposed. These libraries use simplification (at the cost of flexibility), secure defaults, better documentation and the implementation of more complete use-cases (for instance, include support for auxiliary tasks such as key storage) to make it less likely that a developer will make mistakes.

The use of assertions, contracts and defensive programming [15, c3] is a general approach to construct software with high reliability, and it is a highly useful approach to avoid API vulnerabilities. Design by contract makes the contract of an API explicit by providing pre-conditions and post-conditions, and in defensive programming these preconditions will be checked, thus avoiding the occurrence of untrapped errors.

A programming language API also determines the interface between programs in the language and the surrounding system. For instance, JavaScript in a browser does not expose an API to the local file system. As a consequence, JavaScript programs running in the browser can not possibly access

the file system. Such less privileged APIs can be used to contain or *sandbox* untrusted code (see Section 4.3), but they can also prevent vulnerabilities. Object capability systems [19] take this idea further by providing a language and API that supports structuring code such that each part of the code only has the privileges it really needs (thus supporting the *principle of least privilege*).

The design of cryptographic APIs that keep cryptographic key material in a separate protection domain, for instance in a Hardware Security Module (HSM) comes with its own challenges. Such APIs have a security objective themselves: the API to a HSM has the objective of keeping the encryption keys it uses confidential – it should not be possible to extract the key from the HSM. Research has shown [4, c18] that maintaining such a security objective is extremely challenging. The HSM API has an *API-level vulnerability* if there is a sequence of API calls that extracts confidential keys from the HSM. Note that this is an API *design* defect as opposed to the implementation defects considered in Topic 1.

2.3 Coding Practices

The likelihood of introducing the various categories of vulnerabilities discussed in Topic 1 can be substantially reduced by adopting secure coding practices. Coding guidelines can also help against vulnerabilities of a more generic nature that can not be addressed by language or API design, such as, for instance, the guideline to not hard-code passwords. Secure coding practices can be formalised as collections of rules and recommendations that describe and illustrate good and bad code patterns.

A first approach to design such coding guidelines is heuristic and pragmatic: the programming community is solicited to provide candidate secure coding rules and recommendations based on experience in how things have gone wrong in the past. These proposed rules are vetted and discussed by the community until a consensus is reached that the rule is sufficiently appropriate to be included in a coding standard. Influential standards for general purpose software development include the SEI CERT coding standards for C [14] and Java [20].

For critical systems development, more rigorous and stricter coding standards have been developed. The MISRA guidelines [21] have seen widespread recognition and adoption for development of critical systems in C. The SPARK subset of Ada [16] was designed to support coding to enable formal verification of the absence of classes of vulnerabilities.

Rules can take many forms, including:

- the avoidance of dangerous language provided API functions (e.g., do not use the system() function in C),
- attempting to avoid undefined behaviour or untrapped execution errors (e.g., do not access freed memory in C),
- mitigations against certain vulnerabilities caused by the language runtime (e.g., not storing secrets in Java Strings, as the Java runtime can keep those Strings stored on the heap indefinitely), or,
- proactive, defensive rules that make it less likely to run into undefined behaviour (e.g., exclude user input from format strings).

Also, specific side-channel vulnerabilities can be addressed by coding rules, for instance avoiding control flow or memory accesses that depend on secrets can prevent these secrets from leaking through cache-based or branch-predictor based side-channels.

When they are not enforced by a type system, ownership regimes for safely managing resources such as dynamically allocated memory can also be the basis for programming idioms and coding guidelines. For instance, the Resource Acquisition Is Initialisation (RAII) idiom, move semantics

and smart pointers essentially support an ownership regime for C++, but without compiler enforced guarantees.

An important challenge with secure coding guidelines is that their number tends to grow over time, and hence programmers are likely to deviate from the secure practices codified in the guidelines. Hence, it is important to provide tool support to check compliance of software with the coding rules. Topic 3.1 discusses how static analysis tools can automatically detect violations against secure coding rules.

3 Detection of Vulnerabilities

[3, 22] [15, c4]

For existing source code where full prevention of the introduction of a class of vulnerabilities was not possible, for instance, because the choice of programming language and/or APIs was determined by other factors, it is useful to apply techniques to *detect* the presence of vulnerabilities in the code during the development, testing and/or maintenance phase of the software.

Techniques to detect vulnerabilities must make trade-offs between the following two good properties that a detection technique can have:

- A detection technique is *sound* for a given category of vulnerabilities if it can correctly conclude that a given program has no vulnerabilities of that category. An unsound detection technique on the other hand may have *false negatives*, i.e., actual vulnerabilities that the detection technique fails to find.
- A detection technique is *complete* for a given category of vulnerabilities, if any vulnerability it finds is an actual vulnerability. An incomplete detection technique on the other hand may have *false positives*, i.e. it may detect issues that do not turn out to be actual vulnerabilities.

Trade-offs are necessary, because it follows from Rice's theorem that (for non-trivial categories of vulnerabilities) no detection technique can be both sound and complete.

Achieving soundness requires reasoning about *all* executions of a program (usually an infinite number). This is typically done by static checking of the program code while making suitable abstractions of the executions to make the analysis terminate.

Achieving completeness can be done by performing actual, concrete executions of a program that are witnesses to any vulnerability reported. This is typically done by dynamic detection where the analysis technique has to come up with concrete inputs for the program that trigger a vulnerability. A very common dynamic approach is *software testing* where the tester writes test cases with concrete inputs, and specific checks for the corresponding outputs.

In practice, detection tools can use a hybrid combination of static and dynamic analysis techniques to achieve a good trade-off between soundness and completeness.

It is important to note, however, that some detection techniques are heuristic in nature, and hence the notions of soundness and completeness are not precisely defined for them. For instance, heuristic techniques that detect violations of secure coding practices as described in 2.3 are checking compliance with informally defined rules and recommendations, and it is not always possible to unambiguously define the false positives or false negatives. Moreover, these approaches might highlight 'vulnerabilities' that are maybe not exploitable at this point in time, but should be fixed nonetheless because they are 'near misses', i.e., might become easily exploitable by future maintenance mistakes.

Static and dynamic program analysis techniques are widely studied in other areas of computer science. This Topic highlights the analysis techniques most relevant to software security.

Another important approach to detection of vulnerabilities is to perform manual code review and auditing. These techniques are covered in the Secure Software Lifecycle KA. When using tool-supported static detection, it makes sense to adjust such subsequent code review and other verification activities. For instance, if static detection is sound for a given category of vulnerabilities, then one might consider not to review or test for that category of vulnerabilities in later phases.

3.1 Static Detection

Static detection techniques analyse program code (either source code or binary code) to find vulnerabilities. Opposed to dynamic techniques, the static ones have the advantage that they can operate on incomplete code that is not (yet) executable, and that in a single analysis run they attempt to cover all possible program executions. Roughly speaking, one can distinguish two important classes of techniques, that differ in their main objective.

Heuristic static detection. First, there are static analysis techniques that detect violations of rules that are formal encodings of secure programming-practice heuristics. The static analysis technique builds a semantic model of the program, including, for instance, an abstract syntax tree, and abstractions of the data flow and control flow in the program. Based on this model, the technique can flag violations of simple syntactic rules such as, do not use this dangerous API function, or only use this API function with a constant string as first parameter.

An important indicator for the presence of vulnerabilities is the fact that (possibly malicious) program input can influence a value used in a risky operation (for instance, indexing into an array, or concatenating strings to create a SQL query). *Taint analysis* (sometimes also called *flow analysis*) is an analysis technique that determines whether values coming from program inputs (or more generally from designated *taint sources*) can influence values used in such a risky operation (or more generally, values flowing into a *restricted sink*). The same analysis can also be used to detect cases where confidential or sensitive information in the program flows to public output channels.

Many variants of static taint analysis exist. Important variations include (1) how much abstraction is made of the code, for instance, path-sensitive versus path-insensitive, or context-sensitive versus context-insensitive analysis, and (2) whether influences caused by the program control flow instead of program data flow are taken into account (often distinguished by using the terms *taint analysis* versus *information flow analysis*).

To reduce the number of false positives, a taint analysis can take into account *sanitisation* performed by the program. Tainted values that were processed by designated sanitisation functions (that are assumed to validate that the values are harmless for further processing) have their taint removed.

An important challenge is that taint analyses must be configured with the right sets of sources, sinks and sanitisers. In practice, such configurations currently often occur manually although some recent works have added tool assistance in which, for instance, machine learning is used to support security analysts in this task.

Sound static verification. Second, there are static analysis techniques that aim to be sound for well-defined categories of vulnerabilities (but usually in practice still make compromises and give up soundness to some extent). For categories of vulnerabilities that can be understood as specification or contract violations, the main challenge is to express this underlying specification formally. Once this is done, the large body of knowledge on static analysis and program verification developed in other areas of computer science can be used to check compliance with the specification. The three main relevant techniques are program verification, abstract interpretation and model checking.

Program verification uses a program logic to express program specifications, and relies on the programmer/verifier to provide an adequate abstraction of the program in the form of inductive loop invariants or function pre- and post-conditions to make it possible to construct a proof that covers all

program executions. For imperative languages with dynamic memory allocation, separation logic [23] is a program logic that can express absence of memory-management and race-condition vulnerabilities (for data races on memory cells), as well as compliance with programmer provided contracts on program APIs. Checking of compliance with a separation logic specification is typically not automatic: it is done by interactive program verification where program annotations are used to provide invariants, pre-conditions and post-conditions. However, if one is interested only in absence of memory management vulnerabilities, these annotations can sometimes be inferred, making the technique automatic. Also avoiding the use of certain language features (e.g., pointers), and adhering to a coding style amenable to verification can help making verification automatic.

Abstract interpretation is an automatic technique where abstraction is made from the concrete program by mapping the run-time values that the program manipulates to adequate finite abstract domains. For imperative programs that do not use dynamic allocation or recursion, abstract interpretation is a successful technique for proving the absence of memory management vulnerabilities automatically and efficiently.

Model checking is an automatic technique that exhaustively explores all reachable states of the program to check whether none of the states violates a given specification. Because of the state explosion problem, model checking can only exhaustively explore very small programs, and in practice techniques to bound the exploration need to be used, for instance, by bounding the number of times a program loop can be executed. Bounded model checking is no longer sound, but can still find many vulnerabilities.

Most practical implementations of these analysis techniques give up on soundness to some extent. In order to be both sound and terminating, a static analysis must *over-approximate* the possible behaviours of the program it analyses. Over-approximation leads to false positives. Real programming languages have features that are hard to over-approximate without leading to an unacceptable number of false positives. Hence, practical implementations have to make engineering trade-offs, and will under-approximate some language features. This makes the implementation unsound, but more useful in the sense that it reduces the number of false positives. These engineering trade-offs are nicely summarised in the 'Soundiness Manifesto' [24].

3.2 Dynamic Detection

Dynamic detection techniques execute a program and monitor the execution to detect vulnerabilities. Thus, if sufficiently efficient, they can also be used for just-in-time vulnerability mitigation (See Topic 4). There are two important and relatively independent aspects to dynamic detection: (1) how should one monitor an execution such that vulnerabilities are detected, and (2) how many and what program executions (i.e., for what input values) should one monitor?

Monitoring. For categories of vulnerabilities that can be understood as violations of a specified property of a single execution (See Topic 1.6), complete detection can be performed by monitoring for violations of that specification. For other categories of vulnerabilities, or when monitoring for violations of a specification is too expensive, approximative monitors can be defined.

Monitoring for memory-management vulnerabilities has been studied intensively. It is, in principle, possible to build complete monitors, but typically at a substantial cost in time and memory. Hence, existing tools explore various trade-offs in execution speed, memory use, and completeness. Modern C compilers include options to generate code to monitor for memory management vulnerabilities. In cases where a dynamic analysis is approximative, like a static analysis, it can also generate false positives or false negatives, despite the fact that it operates on a concrete execution trace.

For structured output generation vulnerabilities, a challenge is that the intended structure of the generated output is often implicit, and hence there is no explicit specification that can be monitored. Hence, monitoring relies on sensible heuristics. For instance, a monitor can use a fine-grained dy-

namic taint analysis [22] to track the flow of untrusted input strings, and then flag a violation when untrusted input has an impact on the parse tree of generated output.

Assertions, pre-conditions and post-conditions as supported by the design-by-contract approach to software construction [15, c3] can be compiled into the code to provide a monitor for API vulnerabilities at testing time, even if the cost of these compiled-in run-time checks can be too high to use them in production code.

Monitoring for race conditions is hard, but some approaches for monitoring data races on shared memory cells exist, for instance, by monitoring whether all shared memory accesses follow a consistent locking discipline.

Generating relevant executions. An important challenge for dynamic detection techniques is to generate executions of the program along paths that will lead to the discovery of new vulnerabilities. This problem is an instance of the general problem in software testing of systematically selecting appropriate inputs for a program under test [15, c4]. These techniques are often described by the umbrella term *fuzz testing* or *fuzzing*, and can be classified as:

- *Black-box fuzzing*, where the generation of input values only depends on the input/output behaviour of the program being tested, and not on its internal structure. Many different variants of black-box fuzzing have been proposed, including (1) purely random testing, where input values are randomly sampled from the appropriate value domain, (2) model-based fuzzing, where a model of the expected format of input values (typically in the form of a grammar) is taken into account during generation of input values, and (3) mutation-based fuzzing, where the fuzzer is provided with one or more typical input values and it generates new input values by performing small mutations on the provided input values.
- *White-box fuzzing*, where the internal structure of the program is analysed to assist in the generation of appropriate input values. The main systematic white-box fuzzing technique is *dynamic symbolic execution*. Dynamic symbolic execution executes a program with concrete input values and builds at the same time a *path condition*, a logical expression that specifies the constraints on those input values that have to be fulfilled for the program to take this specific execution path. By solving for input values that do not satisfy the path condition of the current execution, the fuzzer can make sure that these input values will drive the program to a different execution path, thus improving coverage.

4 Mitigating Exploitation of Vulnerabilities

[1, 25]

Even with good techniques to prevent introduction of vulnerabilities in new code, or to detect vulnerabilities in existing code, there is bound to be a substantial amount of legacy code with vulnerabilities in active use for the foreseeable future. Hence, vulnerability prevention and detection techniques can be complemented with techniques that mitigate the exploitation of remaining vulnerabilities. Such mitigation techniques are typically implemented in the execution infrastructure, i.e., the hardware, operating system, loader or virtual machine, or else are inlined into the executable by the compiler (a so-called ‘inlined reference monitor’). An important objective for these techniques is to limit the impact on performance, and to maximise compatibility with legacy programs.

4.1 Runtime Detection of Attacks

Runtime monitoring of program execution is a powerful technique to detect attacks. In principle, program monitors to detect vulnerabilities during testing (discussed in 3.2 Dynamic Detection) could also be used at runtime to detect attacks. For instance, dynamic taint analysis combined with a

dynamic check whether tainted data influenced the parse tree of generated output has also been proposed as a runtime mitigation technique for SQL injection attacks.

But there is an important difference in the performance requirements for monitors used during testing (discussed in Topic 3) and monitors used at runtime to mitigate attacks. For runtime detection of attacks, the challenge is to identify efficiently detectable violations of properties that are expected to hold for the execution trace of the program. A wide variety of techniques are used:

- Stack canaries detect violations of the integrity of activation records on the call stack, and hence detect some attacks that exploit memory management vulnerabilities to modify a return address.
- Non-executable data memory (NX) detects attempts to direct the program counter to data memory instead of code memory and hence detects many direct code injection attacks.
- Control Flow Integrity (CFI) is a class of techniques that monitors whether the runtime control flow of the program complies with some specification of the expected control flow, and hence detects many code-reuse attacks.

On detection of an attack, the runtime monitor must react appropriately, usually by terminating the program under attack. Termination is a good reaction to ensure that an attack can do no further damage, but it has of course a negative impact on availability properties.

4.2 Automated Software Diversity

Exploitation of vulnerabilities often relies on implementation details of the software under attack. For instance, exploitation of a memory management vulnerability usually relies on details of the memory layout of the program at runtime. A SQL injection attack can rely on details of the database to which the SQL query is being sent.

Hence, a generic countermeasure to make it harder to exploit vulnerabilities is to *diversify* these implementation details. This raises the bar for attacks in two ways. First, it is harder for an attacker to prepare and test his/her attack on an identical system. An attack that works against a web server installed on the attacker machine might fail against the same web server on the victim machine because of diversification. Second, it is harder to build attacks that will work against many systems at once. Instead of building an exploit once, and then using it against many systems, attackers now have to build customised exploits for each system they want to attack.

The most important realisation of this idea is *Address Space Layout Randomisation (ASLR)*, where the layout of code, stack and/or heap memory is randomised either at load or at runtime. Such randomisation can be *coarse-grained*, for instance, by just randomly relocating the base address of code, stack and heap segments, or *fine-grained* where addresses of individual functions in code memory, activation records in the stack, or objects in the heap are chosen randomly.

The research community has investigated many other ways of automatically creating diversity at compilation time or installation time [25], but such automatic diversification can also bring important challenges to software maintenance as bug reports can be harder to interpret, and software updates may also have to be diversified.

4.3 Limiting Privileges

The exploitation of a software vulnerability influences the behaviour of the software under attack such that some security objective is violated. By imposing general bounds on what the software can do, the damage potential of attacks can be substantially reduced.

Sandboxing is a security mechanism where software is executed within a controlled environment (the ‘sandbox’) and where a policy can be enforced on the resources that software in the sandbox can

access. Sandboxing can be used to confine untrusted software, but it can also be used to mitigate the impact of exploitation on vulnerable software: after a successful exploit on the software, an attacker is still confined by the sandbox.

The generic idea of sandboxing can be instantiated using any of the isolation mechanisms that modern computer systems provide: the sandbox can be a virtual machine running under the supervision of a virtual-machine monitor, or it can be a process on which the operating system imposes an access control policy. In addition, several purpose-specific sandboxing mechanisms have been developed for specific classes of software, such as, for instance, *jails* that can sandbox network- and filesystem-access in virtual hosting environments. The Java Runtime Environment implements a sandboxing mechanism intended to contain untrusted Java code, or to isolate code from different stakeholders within the same Java Virtual Machine, but several significant vulnerabilities have been found in that sandboxing mechanism over the years [26].

Compartimentalisation is a related but finer-grained security mechanism, where the software itself is divided in a number of *compartments* and where some bounds are enforced on the privileges of each of these compartments. This again requires some underlying mechanism to enforce these bounds. For instance, a compartimentalised browser could rely on operating system process access control to bound the privileges of its rendering engine by denying it file system access. Exploitation of a software vulnerability in the rendering engine is now mitigated to the extent that even after a successful exploit, the attacker is still blocked from accessing the file system. Very fine-grained forms of compartimentalisation can be achieved by *object-capability systems* [19], where each application-level object can be a separate protection domain.

To mitigate side-channel vulnerabilities, one can isolate the vulnerable code, for instance, on a separate core or on separate hardware, such that the information leaking through the side channel is no longer observable for attackers.

4.4 Software Integrity Checking

Under the umbrella term *Trusted Computing*, a wide range of techniques have been developed to measure the state of a computer system, and to take appropriate actions if that state is deemed insecure. A representative technique is *Trusted Boot* where measurements are accumulated for each program that is executed. Any modification to the programs (for instance, because of a successful attack) will lead to a different measurement. One can then enforce that access to secret keys, for instance, is only possible from a state with a specified measurement.

Parno et al. [27] give an excellent overview of this class of techniques.

CONCLUSIONS

Software implementation vulnerabilities come in many forms, and can be mitigated by a wide range of countermeasures. Table 1 summarises the relationship between the categories of vulnerabilities discussed in this chapter, and the relevant prevention, detection and mitigation techniques commonly used to counter them.

Acknowledgments The insightful and constructive comments and feedback from the reviewers and editor on earlier drafts have been extremely valuable, and have significantly improved the structure and contents of this chapter, as have the comments received during public review.

CROSS-REFERENCE OF TOPICS VS REFERENCE MATERIAL

Vulnerability category	Prevention	Detection	Mitigation
Memory management vulnerabilities	memory-safe languages, fat/smart pointers, coding rules	many static and dynamic detection techniques	stack canaries, NX, CFI, ASLR, sandboxing
Structured output generation vulnerabilities	regular expression types, LINQ, Prepared Statements	taint analysis	runtime detection
Race condition vulnerabilities	ownership types, coding guidelines	static and dynamic detection	sandboxing
API vulnerabilities	contracts, usable APIs, defensive API implementations	runtime checking of pre- and post-conditions, static contract verification	compartimentalisation
Side channel vulnerabilities	coding guidelines	static detection	isolation

Table 1: Summary overview

	Du:computer-security [2]	Dowd:art [5]	Anderson:security-engineering [4]	Pierce:2002:TPL:509043 [12]	C-coding-standard [14]	swebokv3 [15]	Chess:static-analysis [3]
1 Categories of Vulnerabilities							
1.1 Memory Management Vulnerabilities	c4,c5	c5					c6
1.2 Structured Output Generation Vulnerabilities	c10,c11	c17					c9
1.3 Race Condition Vulnerabilities	c7	c9					
1.4 API Vulnerabilities	c6	c9,c11					
1.5 Side-channel Vulnerabilities			c17				
2 Prevention of Vulnerabilities							
2.1 Language Design and Type Systems				c1			
2.2 API Design			c18			c3	
2.3 Coding Practices					*		
3 Detection of Vulnerabilities							
3.1 Static Detection							*
3.2 Dynamic Detection						c4	
4 Mitigating Exploitation of Vulnerabilities							
4.1 Runtime Detection of Attacks	c4						
4.2 Automated Software Diversity	c4						
4.3 Limiting Privileges	c7						

FURTHER READING

Building Secure Software [28] and 24 Deadly Sins of Software Security [29]

Building Secure Software was the first book focusing specifically on software security, and even if some of the technical content is somewhat dated by now, the book is still a solid introduction to the field and the guiding principles in the book have withstood the test of time.

24 Deadly Sins of Software Security is a more recent and updated book by mostly the same authors.

The Art of Software Security Assessment [5]

Even if this is a book that is primarily targeted at software auditors, it is also a very useful resource for developers. It has clear and detailed descriptions of many classes of vulnerabilities, including platform-specific aspects.

Surreptitious Software [30]

Software security in this chapter is about preventing, detecting and removing software implementation vulnerabilities. However, another sensible, and different, interpretation of the term is that it is about protecting the software code itself, for instance, against reverse engineering of the code, against extraction of secrets from the code, or against undesired tampering with the code before or during execution. Obfuscation, watermarking and tamperproofing are examples of techniques to protect software against such attacks. *Surreptitious Software* is a rigorous textbook about this notion of software security.

OWASP Resources

The Open Web Application Security Project (OWASP) is a not-for-profit, volunteer-driven organisation that organises events and offers a rich set of resources related to application security and software security. They offer practice-oriented guides on secure development and on security testing, as well as a collection of tools and awareness raising instruments. All these resources are publicly available at `https://www.owasp.org`.

REFERENCES

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.13

[2] W. Du, *Computer Security: A hands-on Approach*, 2017.

[3] B. Chess and J. West, *Secure Programming with Static Analysis*, 1st ed. Addison-Wesley Professional, 2007.

[4] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed. Wiley Publishing, 2008.

[5] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.

[6] B. Goetz, J. Bloch, J. Bowbeer, D. Lea, D. Holmes, and T. Peierls, *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006.

[7] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. ACM, 2013, pp. 73–84.

[8] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[9] F. B. Schneider, “Enforceable security policies,” *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, Feb. 2000.

[10] M. Abadi, “Protection in programming-language translations,” in *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, ser. ICALP ’98. London, UK, UK: Springer-Verlag, 1998, pp. 868–883.

[11] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[12] B. C. Pierce, *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[13] L. Cardelli, “Type systems,” in *The Computer Science and Engineering Handbook*, 1997, pp. 2208–2236.

[14] Software Engineering Institute – Carnegie Mellon University, “SEI CERT C coding standard: Rules for developing safe, reliable, and secure systems,” 2016.

[15] IEEE Computer Society, P. Bourque, and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*, 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014.

[16] “SPARK 2014,” http://www.spark-2014.org/about, accessed: 2018-04-17.

[17] H. Hosoya, J. Vouillon, and B. C. Pierce, “Regular expression types for xml,” *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 1, pp. 46–90, Jan. 2005.

[18] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, pp. 5–19, Sep. 2006.

[19] M. S. Miller, “Robust composition: Towards a unified approach to access control and concurrency control,” Ph.D. dissertation, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[20] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda, *The CERT Oracle Secure Coding Standard for Java*, 1st ed. Addison-Wesley Professional, 2011.

[21] MISRA Ltd, *MISRA-C:2012 Guidelines for the use of the C language in Critical Systems*, Motor Industry Software Reliability Association Std., Oct. 2013. [Online]. Available: www.misra.org.uk

[22] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331.

[23] J. C. Reynolds, “Separation logic: A logic for shared mutable data structures,” in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74.

[24] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, “In defense of soundiness: A manifesto,” *Commun. ACM*, vol. 58, no. 2, pp. 44–46, Jan. 2015.

[25] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “Sok: Automated software diversity,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 276–291.

[26] P. Holzinger, S. Triller, A. Bartel, and E. Bodden, “An in-depth study of more than ten years of java exploitation,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16, 2016, pp. 779–790.

[27] B. Parno, J. M. McCune, and A. Perrig, “Bootstrapping trust in commodity computers,” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 414–429.

[28] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way (Paperback) (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, 2002.

[29] M. Howard, D. LeBlanc, and J. Viega, *24 Deadly Sins of Software Security: Programming Flaws*

and How to Fix Them, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2010.

[30] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Addison-Wesley Professional, 2009.