Two Decades of Secure Software Development: Shifting Left, Right and Down

Erik Poll

erikpoll@cs.ru.nl
Digital Security group, Radboud University

Abstract. In the early 2000s security began to receive serious attention in the IT community. This led to the birth of several methodologies for secure software development (notably Microsoft SDL) and many other forms of security advice: Top N lists of common vulnerabilities, secure coding guidelines, and many security frameworks and standards. Now that twenty years later the cry for more secure software has reached policy documents such as the US National Cybersecurity Strategy and the EU Cyber Resilience Act, this paper reflects on developments in the field of software security over these past two decades.

1 Introduction

In the early 2000s there was a growing recognition in the IT community that security was becoming a big problem and that the insecurity of software played a key role here. This led to increased attention to software security, also called application security (or AppSec for short), as field of study. It was also around this time that I got to know Sjouke Mauw, as we were both taking our first steps in the field of security coming from the field of formal methods.

In January 2002 Bill Gates wrote his by now famous email to all Microsoft employees announcing security as key priority for Microsoft in the years to come [16]. A year earlier, in 2001, OWASP had started as community initiative to improve security of web applications. A few years later SAFEcode [49] followed as a collaboration between several large corporations to improve software security.

Fast forward 20 years and the security of software has become an important focus of government policies. The US National Cybersecurity Strategy [39] from 2023 explicitly mentions secure software development; it even announced plans to introduce legislation for software liability but these never materialised. Also in 2023, the EU introduced the Cyber Resilience Act (CRA) [14] that sets cybersecurity rules products containing software: it requires manufacturers to ensure that software in products is free from 'known exploitable security vulnerabilities'. The EU Radio Equipment Directive (RED) [13], which is narrower in scope than the CRA but came into effect sooner, also requires this.

With all this legislation more organisations will have to pay attention – or *more* attention – to the security of software they use or produce in the years to come, so this is good occasion to look back on developments in the field of

software security over the past two decades. There is a *lot* of information around about how to make software more secure – so much so, that it can be confusing for newcomers to the field, even experienced software engineers and computer scientists who never had to deal with security before. This paper aims to provide an overview for such newcomers to the field of software security.

1.1 Process vs Product

Many guidelines, standards, frameworks and methodologies for software security have been published over the past decades. Some security guidelines focus on the *process* of developing software: they propose activities that can be done in the development process to produce more secure software. Some of these guidelines aim to provide comprehensive *methodology* for all security activities in the entire software development lifecycle. The best known of these methodologies is Microsoft's SDL (Secure Development Lifecycle) but there are many others, as we discuss in Section 2.

Other forms of security advice focus more on the *product*. There is a huge variety here, as we discuss in Section 4, incl. lists of common security vulnerabilities such as the OWASP Top 10, secure coding guidelines, and standards prescribing security requirements or security controls.

The distinction between *process* and *product* is useful to keep an overview of different kinds of security guidance, but the two perspectives are related. Security activities in the development process often require knowledge about the software product. For example, all methodologies propose the use of tools to detect common security vulnerabilities as part of the development process; such tools obviously need information about common problems in software products. The two perspectives can even be mutually dependent. For example, doing vulnerability management as part of the software engineering process (with activities for handling reported security flaws, triaging these, fixing important ones and rolling out security updates) may require features in the software product (such as having the possibility to update, automated checks for updates, or crash reporting to help with detecting flaws).

Many security standards mix advice for the process and the product. For example, the ISA/IEC 62443 standard [24] for operational technology (e.g. industrial control systems) defines a secure development process and also lists detailed security requirements to implement in systems. NIST's cybersecurity standard for smart grids [40] even includes a discussion of common classes of software vulnerabilities (so-called CWEs, discussed in Section 5).

Some security standards and frameworks, for instance ISA/IEC 62443 or NIST's Cybersecurity Framework [42], have a broader scope than just the (software) development process and also consider wider deployment and organisational issues. More generally, any secure software development process will have to interact with security practices at an organisational level, for which the ISO 27000 family of standards for information security is the most widely used.

2 Security Guidelines for the Software Engineering Process

The growing attention to software security in the early 2000s led to several proposals for software engineering methodologies that take security into account, also called secure SDLC (Software Development Life Cycle) frameworks.

Gary McGraw, one of the founding fathers of the field of software security, proposed the 'Building Security In' methodology [33], later known as Cigital Touchpoints. Microsoft came with its Secure Development Lifecycle (SDL) [22]. OWASP initially came with CLASP (Comprehensive Lightweight Application Security Process), which no longer exists, and then with SAMM (Software Assurance Maturity Model) [5], which still does.

All these methodologies propose activities (or practices) to be carried at various stages in the software development lifecycle to improve security. A basic tenet in all methodologies is that security should be considered *throughout* the development lifecycle. The activities are usually grouped by the stages of the development lifecycle, such as design, coding, testing and incidence response, alongside overarching activities for education and training and for governance of the entire process. Some of these activities are very specific to software and software engineering, for instance the use of static or dynamic analysis to check code for flaws, commonly referred to as *SAST (Static Application Security Testing)* and *DAST (Dynamic Application Security Testing)*. Other activities are generic security activities that are not specific to software or software engineering, for example threat modelling as initial step (using techniques such as attack trees [32]) or having a process to handle security incidents.

As several large IT organisations began to roll out software security initiatives, using one of these approaches or a home-grown variant, BSIMM (Building Security In Maturity Model) was introduced as a maturity model to measure and compare such initiatives. The first edition of BSIMM from 2009 compared software security initiatives at nine large organisations [35]. BSIMM still exists and is periodically updated[1]. OWASP SAMM is another maturity model. Unlike BSIMM, SAMM is not just meant for *measuring* maturity: SAMM can also be used as framework to introduce or improve a secure software development process; the checklist used by BSIMM is far too long and detailed to be used for that.

Many more secure software development methodologies have been proposed over the years. A survey from 2009 compared the three well-known methodologies at the time [10]: SDL, CLASP and Touchpoints. A more recent survey from 2023 found 28 secure software development methodologies to compare [28]. The key ingredients of all these methodologies are very similar but there are differences in emphasis, in level of detail, and in the way that activities are

[1] BSIMM is a commercial activity of Synopsys (formerly of Cigital, which was acquired by Synopsys). Detailed public information about the latest versions (e.g. BSIMM14 [56]) can be hard to find, but the BSIMM13 checklist is publicly available at https://github.com/rtxsecurity/bsimm13-parsable.

grouped. For example, OWASP SAMM (version 2) lists 15 security activities grouped in 5 'business functions', while BSIMM (version 14) has 12 practices across 4 'domains', with a further breakdown of these 12 practices into 126 activities. Microsoft SDL has been reorganised several times: the initial version had 12 stages, each corresponding to a phase in the development lifecycle, plus 'Education and Awareness' as a special initial stage [22]. The 2012 version had 5 phases – Requirements, Design, Implementation, Verification, Release – with 3 practices per phase, plus 'Training' and 'Executing an incident response plan' as practices in the pre- and post-SDL phase [37]. In the latest edition of SDL this has been reorganised and trimmed down to 10 practices [44].

One of the more recent methodologies is NIST's Secure Software Development Framework (SSDF [41]). It lists 20 practices in four groups, with a further breakdown of these 20 practices into 43 actions. These practices and actions are taken from 25(!) earlier documents (incl. Microsoft SDL, BSIMM and OWASP SAMM). This large number shows how messy the landscape of security standards unfortunately is.

At least SSDF provides clear cross-references to these other standards. An interesting initiative to cope with the growing set of security standards is OWASP OpenCRE (Open Common Requirement Enumeration [15]): it aims to provide mappings between security requirements in different standards.

2.1 SAST & DAST tools

All methodologies mention the use of SAST and DAST tools as practices to improve software security. Many such tools have appeared over the year.

A popular technique in DAST tools is *fuzzing* aka fuzz testing. The basic idea here is to send many (semi)automatically generated malformed inputs to an application and then see if it crashes due to bugs (notably memory corruption bugs). Fuzzing is a very old idea: it was used in the late 1980s to find memory corruption bugs in UNIX utilities [38]. Fuzzing techniques have improved significantly since then. Commercial fuzzing tools for specific protocols and file formats came on the market in the early 2000s (for instance Codenomicon [57]). These tools needed to be tailored to specific protocols or formats, but Microsoft's SAGE fuzzer avoided this by using symbolic execution to find inputs that trigger obscure code paths. SAGE successfully found many bugs in Microsoft Office and in Windows 7 and its successors [18]. The biggest breakthrough in fuzzing came with `afl` [63] in 2013 and its coverage-guided evolutionary approach (aka greybox fuzzing) to find interesting test cases. Fuzzing with afl could have prevented high-profile security problems such as the HeartBleed bug in OpenSSL; this observation led Google to launch the OSS-Fuzz initiative [53] to fuzz open source projects at scale: since 2016 this has discovered over 36,000 bugs in over 1,000 open source projects.

Most SAST tools use *data flow analysis* be to detect if user input can end up in dangerous places, as for instance happens in injection attacks (discussed in Section 5.5). This requires detailed information about the APIs involved. When there is a rapid evolution or turnover in platforms (as is the case with

JavaScript frameworks for web applications) keeping tools up to date requires constant effort. Most SAST tools can be configured with rules (or queries) for specific bug patterns, but some of the newer so-called query-based SAST tools [30], notably CodeQL and Semgrep, just provide generic analysis capabilities and always need to be used in combination with some rule-set; sharing such rule-sets may be a way around the need to update tools. With ever more SAST and DAST tools choosing the right one can be tricky. Unfortunately is not much research into comparing tools: commercial tool vendors are not keen to participate and it is hard to decide on a good set of benchmarks to base any comparison on. Already in 2005 NIST started the SAMATE initiative to evaluate and compare tools [3], but unfortunately it never produced the clear insights into the relative performance of tools that were hoped for.

2.2 Shifting Left and Shifting Down

Adopting a secure software development methodology is not something that can be done overnight. Introducing and then improving it will take time and be an ongoing process. The way that most organisations evolve to a more mature security process is by gradually moving security activities and knowledge to earlier development stages, moving from a reactive to a more proactive stance. This is known as *shifting left*. For example, a typical first step to improve software security is to have applications pen-tested, but ideally security problems would be caught earlier by shifting left: e.g. by deploying DAST or SAST tools during development, by improved training for developers, or doing risk analyses when the software architecture is designed, prior to any code development.

The ultimate way to shift left is to *shift down* by addressing security risks in underlying technology stacks, e.g. with the safer APIs or safer programming languages. This can eliminate entire classes of vulnerabilities, a goal that is highlighted in CISA's security-by-design guidance [7]. It can be regarded as the ultimate form of security-by-design, discussed below, as security is then already considered in the design of technology stacks before the development of individual applications that run on these stacks even begins. One could argue that shifting down should be called shifting up, as the outcome is that developers can focus on issues higher up in the software stack. We discuss such structural solutions to some common classes of security vulnerabilities in Section 5.

2.3 Security by Design

All the methodologies mentioned in Section 2 stress the importance of taking security into account right from the start. The slogan *secure by design* has become popular to express this idea. Long before the slogan security-by-design became popular McGraw used the slogan 'building security in' as opposed to 'bolting security on' to express the idea that security should be considered in all stages of the development lifecycle [33].

Beware that people can interpret the term security-by-design differently, as pointed out by Del Real et al. [11]: some people use a narrow interpretation

where it *only* refers to the initial design phase of a system, whereas others take a broad interpretation where it applies to all the stages of the software development lifecycle. The 'Secure by Design' white paper by CISA and other national cybersecurity agencies [7] clearly takes a broad interpretation of the term. For example, it mentions having a vulnerability management program as a secure-by-design practice. In fact, it mentions documented adherence to a secure software development methodology as a secure-by-design practice, which implies that security-by-design spans the entire development life cycle.

2.4 Shifting Right and Resilience

All the talk of shifting left should not overshadow the fact that shifting right can also be important. No matter how much we try to shift left, some security flaws may be missed and some security threats may be overlooked altogether. Processes to deal with incidents will always be necessary: ideally to mitigate the impact when incidents occur, but at least to investigate incidents after the fact and discover and address root causes. Indeed, many good security solutions, not just in software in general, rely more on detection and response than on prevention.

Here we can still benefit from shifting left by building in possibilities for monitoring and response from the start (as argued for by Etalle [12]), which could be considered an instance of security-by-design. An example of this is the use of RASP (Runtime Application Self-Protection) in mobile apps [21], where the software is instrumented to detect suspicious behaviour at runtime. Building in such monitoring is an example of shifting left, using it can be seen as an example of shifting right.

The term *resilience* has become more popular in recent years to refer to the ability of systems or organisations to cope with security incidents. While the term may be useful to stress the importance of detection and recovery, we would argue that good security always includes resilience.

3 Changes in Software Engineering

Software development has changed a lot over the past two decades. Some of these changes complicate the adoption of a more secure software development methodologies, others introduce new risks which then require special attention, as discussed below.

3.1 Agile and DevOps

The methodologies discussed in Section 2 all use the classical stages of the waterfall model as frame of reference to position security activities. But software engineering has moved away from this model in the past decades, with Agile and DevOps as popular trends. This has led to proposals on how to integrate

security practices in Agile or DevOps approaches. For example, Microsoft published guidance on how to carry out SDL activities in an Agile setting [17] and the term DevSecOps was coined for ways to incorporate security practices into the DevOps process[2].

Adopting an Agile or DevOps way of working does not mean that the security activities proposed by the original secure development methodologies should no longer be used, or that different activities are needed. But incorporating these activities in the shorter and more frequent development and release cycle does pose an extra challenge. For example, when Agile or DevOps approaches lead to many incremental changes in a product it is hard to decide when to have it pen-tested: you cannot do a pen-test after every sprint or for every release. This means that shifting left becomes more important. It also means that automation of activities becomes more important, so that e.g. SAST or DAST can be integrated in CI/CD pipelines.

3.2 Supply-chain Risks

Another big change in software development in the past decades has been the dramatic rise in the use of open source components, enabled by code repositories such as github, Sourceforge, PyPi for Python, NPM for JavaScript, or Maven for Java. Most proprietary software products nowadays contain large amounts of open source code.

Using such third-party, often open-source, components comes with security risks: risks of accidental security flaws (such as the Log4J vulnerability discovered in 2021 [9]) and risks of deliberate security flaws or backdoors (such as the SolarWinds incident in 2020 [59]). Sonatype reported a 742% increase in supply chain attacks on open source software in 2022 [54].

All this has led a new popular type of static analysis tool, namely *SCA (Software Composition Analysis) tools*, to analyse the dependencies of software projects.

Another measure to manage software supply chain risks is the *SBOM (Software Bill of Materials)*. An SBOM is a machine-readable listing of the software components of a product. CISA's website [3] provides extensive documentation about SBOMs. For a discussion of benefits of SBOMs and challenges in adopting them see Zahan et al. [62]. A report by Idaho National Lab from 2023 gives insight into current adoption [55].

Software supply chain concerns have also led to new standards. For instance, OWASP's Software Component Verification Standard (SCVS) [46] proposes measures to reduce software supply chain risks; some of these are included in NIST's SSDF. More recent is the P-SSCRM framework for managing supply chain

[2] DevSecOps was started by a group of security practitioners, see https://www.devsecops.org). Microsoft publishes information about it (at https://www.microsoft.com/en-us/securityengineering/devsecops), as does OWASP (at https://devsecops.owasp.org); here there is even also a proposal for a DevSecOps maturity model.

[3] https://www.cisa.gov/sbom

risk [61]; it includes mappings to another 10 standards (incl. SCVS, SSDF and BSIMM). The latest version of Microsoft SDL also stresses the importance of securing the software supply chain by making this one of the ten SDL practices.

3.3 Risks of Leaking Credentials

Several trends in software development have increased the risks of leaking credentials (or secrets). There are ever more credentials around, in the software development process or in the software itself. For instance, Service-Oriented Architectures (SOAs), which have become more popular, involve credentials: to use external services an application will often need API keys to authenticate. Source code repositories, cloud storage solutions, automated build and deployment processes in CI/CD pipelines, cloud services to support this such as Azure DevOps all come with logins and credentials – and with ways of leaking them [36], for instance in logs. Secrets can also be leaked via Jira, Slack, Confluence, Microsoft Teams, etc. The trend of 'process as code' also contributes to more code that needs to use credentials.

These risks have led to a new type of security tools, namely *secret scanning tools*; for a comparison of such tools see Basak et al. [2]. The popularity of SaaS, SOAs and micro-services has led to proposals for *SaaS Bill of Materials* or *SaaSBOMs* [23]. Just like an SBOM lists the software components used in an application, a SaaSBOM would list the SaaS services used by an application. After all, just like software components can pose security risks, so do software services.

An advantage of services over components is that it is not the responsibility of the application maker to do security updates of any services it uses. Downside is that it may involve credentials that can be leaked. Also, for all their problems, CVEs do provide insight in security issues in components, whereas for some services there may be less transparency about security issues.

4 Security Guidelines for the Software Product

The methodologies and tools discussed in Section 2 still need to be fed with more concrete information about common security problems and ways to avoid them. There is a broad range of such information, some of which is specific to a particular programming language, API, or type of application (e.g. web applications).

Top N list of standard security vulnerabilities, such as the OWASP Top 10 and CWE Top 25, are crucial for awareness and training and provide starting point for improved detection – or better still prevention.

Coding guidelines can help to reduce security problems or more generally improve software quality. Well-known examples are the SEI/CERT coding guidelines for C, C++, Java, Perl and Android [52].

Downside of Top N lists is that they tend to focus on flaws introduced in the coding phase so that design flaws may not get the attention they deserve.

There are also lists of common mistakes in the design phase [1] which provide a starting point for design principles to avoid these. The seminal article by Saltzer and Schroeder [50] already provided general design principles for building secure systems back in the 1970s.

There are also documents that propose standard security requirements – or security controls to implement to meet such requirements. One of the oldest and more mature examples is the OWASP ASVS (Application Security Verification Standard) [47], currently in its 4th edition. The ASVS aims to provide a comprehensive list of security requirements to ensure and security controls to implement to achieve this. Simply put, whereas the OWASP Top 10 provides a list of dont's, the ASVS provides a list of dos. The ASVS can be used in different ways: as guidance during design, implementation and testing; as metric when assessing security; or as standard in procurement. The awkward acronym has probably not helped the ASVS in getting the attention it deserves.

Alongside the ASVS for web applications OWASP also published a similar MASVS standard for mobile apps [48] and, as already mentioned in Section 3.2, the SCVS standard [46] aimed at tackling supply chain security. More standards with security requirements will be produced for the latest EU cybersecurity regulation, i.e. the CRA and RED.

5 Common Vulnerabilities and Ways to Avoid Them

Most people will start to learn about software security through examples of common security vulnerabilities such as buffer overflows or SQL injection. Knowledge about such common problems is crucial; without it you do not stand any chance to produce secure software. The OWASP Top 10 and the CWE Top 25 are the best known lists of common classes of security vulnerabilities (aka 'bug categories')[4]. Changes in these Top N lists over the years can also shed light on progress in improving software security – or the lack thereof.

Any Top N list should be taken with a serious pinch of salt: it is hard to get accurate statistics about security vulnerabilities, hard to classify them into categories, and hard to weigh impacts to pick the most important ones. Moreover, any Top N list is only an incomplete list of potential problems, as it will not include application-specific risks[5] Indeed, there are already several OWASP Top 10s by now: in addition to the original Top Ten of Web Application Security Risks, there is also an Mobile Top Ten, an API Top Ten and most recently a Top Ten for Large Language Model Applications.

The full CWE classification[6] has ballooned to around a thousand categories of security flaws. This may be useful to record very detailed statistics (though

[4] See https://owasp.org/Top10 and https://cwe.mitre.org/top25.

[5] These may be instances of broad and vague categories in the CWE classification (e.g. CWE-435, 'Improper Interaction Between Multiple Correctly-Behaving Entities') but such categories usually do not come with actionable advice on how to prevent them.

[6] https://cwe.mitre.org

the very fine-grained nature makes accurate classification hard: plenty of bug categories overlap) but it is clearly far too long for developers to use, say as a checklist.

5.1 The Big Three

It is easy to be overwhelmed by long lists of common security flaws, especially when looking through the entire CWE list. Fortunately, there are only three important families of problems that make up the large majority of problems. In no specific order, these are:

1. access control flaws, incl. authentication problems;
2. memory corruption flaws; and
3. input handling flaws, notably injection attacks.

If you look at any of the OWASP Top 10s or CWE Top 25s from the past decades you will find that nearly all entries belong to one of these three families.

There are overlaps between these three families. Exploitable memory corruption flaws often arise in input handling, so these could also be regarded as input handling flaws. Injection attacks, where some user input ends up in an API or back-end service that then can be abused, can be seen as access control problems because they involve by-passing access control by tricking a privileged victim application into performing actions (aka the Confused Deputy problem) and tighter access control can mitigate the impact. CSRF and SSRF (client-side and server-side request forgeries) are usually regarded as access control problems, as weak or implicit authentication is to blame, but they can also be viewed as injection attacks.

5.2 Access Control

Access control in the broad sense involves not just authorisation but also authentication, as well as logging and monitoring[7]. Security problems commonly arise in all these aspects: authentication mechanisms can be too weak, authorisation can be misconfigured or may be missing, and there can be insufficient logging and monitoring.

Access control flaws are fundamentally different from the other two families because we deliberately introduce access control to provide security; in fact, it is the most important security control that there is. Whereas we can hope to get rid of memory corruption or injection problems by improved platforms, we will always need access control, so flaws in it – esp. misconfiguring or forgetting it – will always remain a risk. This is not to say that better platforms or better design cannot make a difference. For instance, having a robust mechanism for session management built into platforms can help in reducing CSRF flaws.

[7] A useful mnemonic for this is AAAA: Authentication, Authorisation, Auditing and Action. The AAAA quartet often provides more actionable guidance than the more widely known CIA triad (Confidentiality, Integrity and Availability).

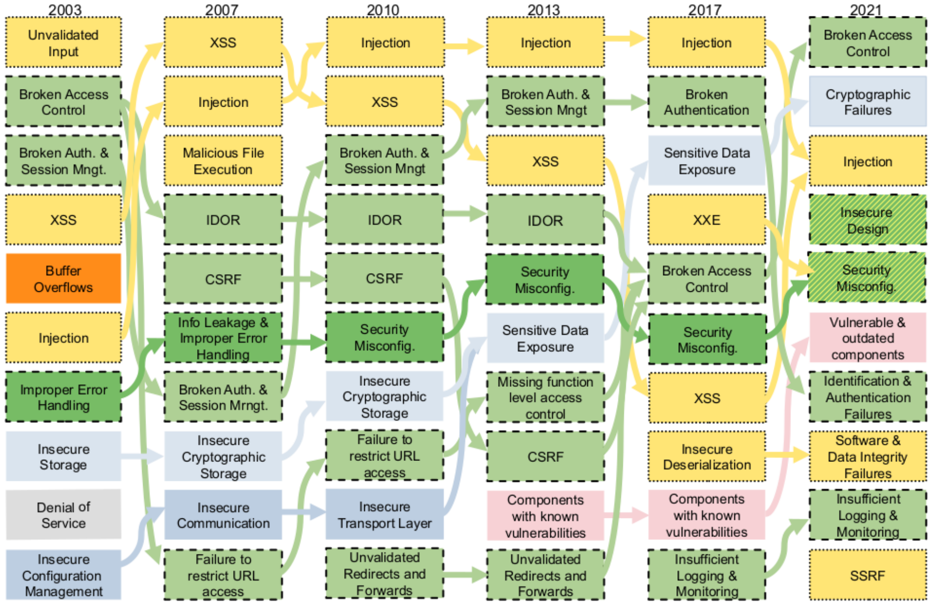

Fig. 1. Evolution of the OWASP Top 10 from 2003 to 2021 with access control problems in green (and dashed line) and input handling problems in yellow (and dotted line). These colours and the arrows indicating inclusions between categories are only approximate. E.g., 'Insecure Design' and 'Security Misconfiguration' mainly include access control issues, but also some that can be regarded as injection problems.

Some changes over time simply reflect the popularity of certain technologies: as XML became more popular XXE attacks became a bigger risk and entered the Top 10; SSRF made its appearance with the rising popularity of service-oriented architectures. Other changes are due to improved knowledge of attackers. For example, possibilities for insecure deserialisation had been around for decades but only entered the Top 10 in 2017 when attackers became aware of this.

There are some positive signs of improvements over the years. Buffer overflows disappeared after the first edition because web applications are mostly written in memory-safe languages. CSRF has dropped over the years because modern platforms for web applications provide good built-in session mechanisms. The authors of the 2021 edition conjecture that injection attacks dropped from the top spot due to improvements in platforms such as safer APIs. So these improvements are due to *shifting down* as discussed in Section 2.2.

Most changes in the OWASP Top 10 over the years are due to changes in the way that bug categories are organised. This highlights the difficulty in making a taxonomy of security vulnerabilities. As bug categories have been broadened, the OWASP Top 10 has become less specific to web applications: apart from the absence of memory corruption bugs, the 2021 edition of OWASP Top 10 is probably a good guide for just about any piece of software.

The 2021 edition saw a big shake-up in the categories: it is now a mixture of very broad categories, such as 'Insecure Design' (included to highlight the importance of shifting left) alongside very narrow ones, such as SSRF. Inclusion of 'Insecure Design' means it is encroaching on the territory of the methodologies discussed in Section 2. Indeed, the authors of the OWASP Top 10 suggest that organisations just beginning with efforts to improve software security could use the Top 10 as a starting point [45].

5.3 Memory Corruption

For software written in memory-unsafe programming languages it is hard to overstate the importance of memory corruption bugs. Both Microsoft and Google's Chrome team report that around 70% of all their security flaws are memory corruption bugs [58,20]. Of the 130 critical security flaws in Chrome up to 2019 only 5 were *not* clearly due to memory corruption [19].

Many countermeasures against memory corruption attacks have been introduced over the years. Most platforms now come with countermeasures to detect flaws (e.g. stack canaries, shadow stacks or more advanced forms of control flow integrity) or make exploitation harder (e.g.. ASLR, non-executable stacks and pointer encryption). There are coding guidelines for C(++) [52], improved libraries that are less error-prone[8], and many SAST and DAST tools that look for memory corruption flaws.

While the measures above will catch some memory corruption bugs or make them hard to exploit, it is clear that they only offer limited protection. In fact, what is amazing about the statistics from Microsoft is that the percentage of memory corruption bugs has remained around 70% and not changed over the years (at least from 2006 to 2018 [58]) despite huge investments in training, coding guidelines, improved libraries, and SAST and DAST tools.

Structural Solutions: Memory-Safety & LangSec The obvious structural solution against memory corruption is to switch to memory-safe programming languages. In 2023 CISA in the US, in collaboration with other national cybersecurity agencies, launched an initiative to push for a transition to memory safe programming languages[9]. With large industry players joining the Rust Foundation there is some real momentum building behind Rust as memory-safe programming language for low-level system programming.

The LangSec methodology [4,29,51] provides an interesting perspective on memory corruption problems in input-handling. It highlights underlying root causes and suggests structural ways to tackle these. LangSec stands for language-theoretic security. The languages it refers to are not programming languages but *input languages*, i.e. the data formats, file formats or protocol message formats that applications have to process. Key observation behind the methodology is that important factors contributing to input handling problems are: (i) the complexity of input languages, (ii) the large number of input languages, (iii) the sloppy definitions of these input languages, and (iv) the expressivity of input languages. Hand-written parser code written in a memory-unsafe programming language for a complex and poorly-specified input format is almost guaranteed to contain exploitable security flaws. Having clearly defined and ideally simpler input formats and using parser generators to produce parsers instead of hand-writing them can avoid all this.

---

[8] The notoriously insecure function `gets()` was removed in the 2011 edition of the ISO C standard.

[9] https://www.cisa.gov/case-memory-safe-roadmaps

5.4 Injection Attacks

In injection attacks input supplied by an attacker ends up being parsed and processed by some back-end service or API which can be abused to trigger unwanted actions. There are many variants of injection attacks: the CWE classification includes over 30.

The category of injection attacks is larger than most people realise: XSS is also an injection attack, as are deserialisation attacks, XXE and SSRF. After all, in an SSRF attack malicious input to a server (often a URL) causes the server to make inappropriate requests to other systems; conceptually this is no different than a SQL injection or path traversal. Attacks with Word or Excel documents that contain malicious macros, a long-standing popular attack technique, are also examples of injection attacks.

Structural Solutions An early initiative to provide structural help in making web applications more secure was the OWASP ESAPI project[10] to provide APIs for various forms of validation and encoding needed in web applications. Important root cause of security problems in web applications, notably XSS, is that web applications handle a set of complex languages, namely HTTP, URLs and HTML, with JavaScript and CSS as sub-languages, which can be nested and then require various encodings to prevent misinterpretation. Note that this involves some of the root causes signalled by the LangSec approach, namely the large number and complexity of input languages.

Safer APIs can reduce the risk of injection attacks. The classic example is the use of parameterised queries or prepared statements to prevent SQL injection. A generalisation of this idea is the use of 'safe builders' to ensure proper output encoding [26], where the type checker of the programming language will catch insecure use of APIs. The Trusted Types API, an improved version of the DOM API in web browsers, uses this idea to combat XSS; this approach has shown to be very effective at Google to root out XSS, incl. DOM-based XSS as the most complex variant of XSS [60].

5.5 Improper Use of 'Improper Input Validation'

The notion of input validation has proved to be a persistent source of misunderstanding both in classifying and tackling input handling problems. It is telling that the first edition of the OWASP Top 10 had 'Unvalidated Input' as top entry but this category then disappeared in all subsequent editions (see Fig. 1). This is not because the problem was solved, but because bug classification was improved.

The classic example of lack of input validation is an application accepting a negative number when a positive number is expected. The *only* way to fix this is for the application to reject such invalid inputs. Input validation *can* be used to prevent injection attacks, but it is not the best solution and usually totally

[10] https://owasp.org/www-project-enterprise-security-api

inappropriate. Using *output encoding* is better, ideally using the type system of the programming language to ensure proper encodings, as discussed above; using a safer API that is not injection-prone is best. Similarly, input validation *can* prevent memory corruption bugs from being exploited, by preventing malformed inputs from reaching vulnerable parser code, but again this is not the best solution: validating input before it is parsed means introducing yet another parser, for the validation process, which can again be a source of problems. Instead of validating data it is better to parse data into an appropriate data structure to avoid problems with possible malformed or misinterpreted data once and for all, as nicely expressed by the slogan *'parse, don't validate'* [27].

Sloppy use of terminology adds to the confusion about input validation. There is a fundamental difference between (i) *rejecting invalid data* (e.g., rejecting an email address that is not a valid email address) (ii) *normalising* data (e.g., removing trailing space characters in a username or changing all characters to lower case) and (iii) *encoding data* because special characters may cause problems in some back-end (e.g., HTML-encoding data to prevent XSS). Unfortunately the terms 'validation' or 'sanitisation' are commonly loosely used for any of these operations, or indeed any combination. The existence of many (near)synonyms – neutralising, quoting, escaping and filtering – adds to the confusion.

6 Conclusion

The good news is that we know a lot about software security, and a lot more than 20 years ago. There are many methodologies for secure software engineering that broadly agree in the steps to be taken. There are many SAST and DAST tools that can be used as part of these methodologies – with fuzzing as a big success story – and standards proposing lists of security requirements. We know what the common security vulnerabilities are, even though the business of classifying them remains very messy, and we have insights into root causes for many and some ways to structurally tackle these.

Secure-by-Design is a nice slogan for all this but, as discussed in Section 2.3, may mislead people into thinking that just sticking a 'security by design' phase in front of their usual development process will take care of security. That would be a mistake, as all secure software development methodologies stress that security requires attention *throughout* the development lifecycle.

The bad news is that the huge number of standards, frameworks and tools makes it hard to see the forest for the trees. In fact, there are several forests: there is a forest of secure development methodologies, a forest of tools, a forest of standards proposing security requirements and the forest of vulnerability categories provided by the CWE classification. A single methodology or standard by itself can already be a mini-forest with dozens of activities or requirements.

Many methodologies and standards make very similar, if not identical, recommendations, but differences in terminology or ways of grouping things can make that hard to spot. NIST even produced a report documenting their standard approach to map relations between security standards [43]. This is not a

new phenomenon: already in 2009 BSIMM was started to provide a common frame of reference for the first secure software development methodologies that had emerged at the time.

A forest we have not even mentioned in this paper is the forest of known security vulnerabilities provided by the CVE catalogue. One initiative to cope with that forest is the KEV (Known Exploited Vulnerabilities) list, which was introduced by CISA in 2022 as a subset of the CVE list to help organisations prioritise certain patches. The EU CRA will force many software producers to improve their vulnerability management processes in years to come, as it requires produces to be free from known exploitable vulnerabilities. Here new scoring systems for the severity of vulnerabilities, for instance EPSS [25], have been proposed as alternative to CVSS [31] to help with prioritising patches.

Looking back over the past two decades it is actually surprising how little has changed: the basic ingredients of Microsoft's SDL or McGraw's Touchpoints are still the main ingredients of the newer methodologies. Looking at the kind of vulnerabilities that cause the bulk of security problems it is disappointing – not to say depressing – to see how little has changed, as vulnerabilities that were common two decades ago are still common today.

For memory corruption vulnerabilities it can be probably be excused that they still dominate the CWE Top 25. Getting rid of these bugs in memory-unsafe programming languages has proven to be very hard. The best hope to get rid of these bugs seems to be to move to memory-safe languages. Here it is good to see the momentum building behind Rust as an alternative.

For many of the common vulnerability types on the other hand, e.g. standard injection flaws such as SQL injection or path traversal, there is no excuse why these should still be so prevalent. These were already deemed to be 'unforgivable' back in 2007 [6]. It is embarrassing for the professional software engineering community that in 2024 government agencies still have to launch appeals to get rid of such vulnerabilities [8]. Their prevalence could be due to a lack of knowledge – i.e. security still not getting enough attention in training even though there is so much information about it – or a lack of incentives to put this into practice. It will be interesting to see if legislation such as the CRA will change that.

It is telling that in May 2024, 22 years after Bill Gates's original email to highlight security as a top priority, Microsoft was one of the companies to sign up to CISA's Secure-by-Design pledge[11]. Clearly the job to improve software security is never done. Looking ahead, one big unknown factor in the future is how AI will impact all this. AI can be a useful tool for both attackers and for defenders and it is not clear who has most to gain here. Moreover, the use of AI in software products will give rise to new risks [34].

Acknowledgement. Research funded by the Dutch Research Council NWO through the INTERSECT project (NWA.1160.18.301) and NCSRA III (ENCRY.2021.001).

[11] https://www.cisa.gov/securebydesign/pledge

References

1. Arce et al., I.: Avoiding the top 10 software security design flaws. Tech. rep., IEEE Computer Society Center for Secure Design (CSD) (2014)
2. Basak, S.K., Cox, J., Reaves, B., Williams, L.: A comparative study of software secrets reporting by secret detection tools. In: Empirical Software Engineering and Measurement (ESEM). IEEE (2023)
3. Black, P.E.: Software assurance with SAMATE reference dataset, tool standards, and studies. In: Digital Avionics Systems Conference. IEEE (2007)
4. Bratus, S., Locasto, M.E., Patterson, M.L., Sassaman, L., Shubina, A.: Exploit programming: From buffer overflows to weird machines and theory of computation. USENIX ;login: pp. 13–21 (2011)
5. Chandra, P.: OWASP Software Assurance Maturity Model (SAMM), version 1.0. OWASP (2009)
6. Christey, S.: Unforgiveable vulnerabilities. Tech. rep., MITRE (2007)
7. Shifting the balance of cybersecurity risk: Principles and approaches for security-by-design and -default. Tech. rep., CISA (June 2023)
8. Secure by design alert: Eliminating directory traversal vulnerabilities in software. Tech. rep., CISA (May 2024)
9. Cyber Safety Review Board: Review of the December 2021 Log4J event. Tech. rep., U.S. Department of Homeland Security (July 2022)
10. De Win, B., Scandariato, R., Buyens, K., Grégoire, J., Joosen, W.: On the secure software development process: CLASP, SDL, and Touchpoints compared. Information and Software Technology **51**(7), 1152 – 1171 (2009)
11. Del-Real, C., de Busser, E., van den Berg, B.: Shielding software systems: A comparison of security by design and privacy by design based on a systematic literature review. Computer Law & Security Review **52** (2024)
12. Etalle, S.: From intrusion detection to software design. In: European Symposium on Research in Computer Security (ESORICS'17). LNCS, vol. 10492, pp. 1–10. Springer (2017)
13. EU: Commission Delegated Regulation 2022/30 supplementing Directive 2014/53/EU (2022), aka Radio Equipment Directive (RED)
14. EU: Cyber Resilience Act: Regulation 2024/2847 on horizontal cybersecurity requirements for products with digital elements (2024)
15. Gasteratos, S., van der Veer, R.: Open Common Requirement Enumeration, see https://www.opencre.org.
16. Gates, B.: Trustworthy computing (January 15 2002), internal memo to all Microsoft employees
17. Gluck, D., Mazolli, R.: SDL-Agile requirements. Tech. rep., Microsoft (2012)
18. Godefroid, P.: Fuzzing: hack, art, and science. Communications of the ACM **63**(2), 70–76 (2020)
19. The rule of 2. Google (2019), https://chromium.googlesource.com/chromium/src/+/master/docs/security/rule-of-2.md, documentation of the Chromium project
20. Chromium security - memory safety. Google (May 2020), https://www.chromium.org/Home/chromium-security/memory-safety
21. Haupert, V., Maier, D., Schneider, N., Kirsch, J., Müller, T.: Honey, I shrunk your app security: The state of Android app hardening. In: Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 69–91. Springer (2018)
22. Howard, M., Lipner, S.: The Security Development Lifecycle. Microsoft (2006)

23. Hughes, C., Haydock, W.: The case for a SaaS bill of material. CSO online (2021), https://www.csoonline.com/article/571267/the-case-for-a-saas-bill-of-material.html
24. ISA/IEC 62443, Security for Industrial Automation and Control Systems, parts 1-4. International Society of Automation (ISA) (2007-2023)
25. Jacobs, J., Romanosky, S., Edwards, B., Adjerid, I., Roytman, M.: Exploit Prediction Scoring System (EPSS). Digital Threats **2**(3) (2021)
26. Kern, C.: Preventing security bugs through software design (2015), invited talk at USENIX Security'15
27. King, A.: Parse, don't validate (2019), https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate, personal blog
28. Kudriavtseva, A., Gadyatskaya, O.: Secure software development methodologies: A multivocal literature review (2023)
29. LangSec: Recognition, validation, and compositional correctness for real world security (2013), http://langsec.org/bof-handout.pdf
30. Li, Z., Liu, Z., Wong, W.K., Ma, P., Wang, S.: Evaluating C/C++ vulnerability detectability of query-based static application security testing tools. IEEE Transactions on Dependable and Secure Computing **21**(5), 4600–4618 (2024)
31. Liska, A.: CVSS scores are dead. Let's explore 4 alternatives (2021), presentation at RSA 2021 conference
32. Mauw, S., Oostdijk, M.: Foundations of attack trees. In: Information Security and Cryptology (ICISC 2005); revised selected papers. pp. 186–198. Springer (2006)
33. McGraw, G.: Software security: building security in. Addison-Wesley (2006)
34. McGraw, G., Bonett, R., Shepardson, V., Figueroa, H.: The top 10 risks of machine learning security. Computer **53**(6), 57–61 (2020)
35. McGraw, G., Chess, B.: The Building Security in Maturity Model (BSIMM). In: USENIX Security. USENIX (2009)
36. Meli, M., McNiece, M.R., Reaves, B.: How bad can it git? characterizing secret leakage in public github repositories. In: NDSS). The Internet Society (2019)
37. Microsoft Security Development Lifecycle (SDL) – version 5.2. Microsoft (2012), https://learn.microsoft.com/en-us/previous-versions/windows/desktop/cc307748(v=msdn.10)
38. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the ACM **33**(12), 32–44 (1990)
39. National Cybersecurity Strategy. The White House (March 2023)
40. NIST: IR 7628 (version 1), Guidelines for Smart Grid Cybersecurity (September 2014)
41. NIST: Secure Software Development Framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities (February 2022)
42. Cybersecurity Framework (CSF 2.0). NIST (February 2024)
43. NIST: IR 8477, Mapping relationships between documentary standards, regulations, frameworks, and guidelines: Developing cybersecurity and privacy concept mappings (February 2024)
44. Ornstein, D., Rice, T.: Building the next generation of the Microsoft Security Development Lifecycle (SDL). Tech. rep., Microsoft (2024)
45. How to start an AppSec program with the OWASP Top 10. OWASP, https://owasp.org/Top10/A00_2021-How_to_start_an_AppSec_program_with_the_OWASP_Top_10/, date visited 4-Sept-2023.
46. Software Component Verification Standard (SCVS), version 1.0. OWASP (2020)
47. Application Security Verification Standard (ASVS), version 4.0.3. OWASP (2021)

48. Mobile Application Security Verification Standard (MASVS), version 2.0.0. OWASP (2023)
49. SAFECode secure development practices. SAFECode, https://safecode.org/category/resource-secure-development-practices
50. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE **63**(9), 1278–1308 (1975)
51. Sassaman, L., Patterson, M.L., Bratus, S., Shubina, A.: The halting problems of network stack insecurity. USENIX ;login: **36**(6), 22–32 (2011)
52. SEI CERT coding standards. SEI (Software Engineering Institute), CMU, https://wiki.sei.cmu.edu
53. Serebryany, K.: OSS-Fuzz - Google's continuous fuzzing service for open source software (2017), invited talk at USENIX 2017
54. 8th annual state of the software supply chain. Tech. rep., Sonatype (2022)
55. Stoddard, J.T., Cutshaw, M.A., Williams, T., Friedman, A., Murphy, J.: Software Bill of Materials (SBOM) sharing lifecycle report. Tech. Rep. INL/RPT-23-71296-Rev000, Idaho National Laboratory (2023)
56. BSIMM 14 report. Tech. rep., Synopsys (2023)
57. Takanen, A.: Fuzzing: the past, the present and the future. In: Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC) (2009)
58. Thomas, G.: A proactive approach to more secure code (2019), Microsoft Security Response Center blog, https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code
59. United States Government Accountability Office: Federal response to SolarWinds and Microsoft Exchange incidents. Tech. Rep. GOA-22-104746, U.S. Department of Homeland Security (2022)
60. Wang, P., Bangert, J., Kern, C.: If it's not secure, it should not compile: Preventing DOM-based XSS in large-scale web development with API hardening. In: ICSE'21. pp. 1360–1372. IEEE (2021)
61. Williams, L., Migues, S., Boote, J., Hutchison, B.: Proactive software supply chain risk management framework (P-SSCRM) version 1. arXiv preprint (arXiv:2404.12300) (2024)
62. Zahan, N., Lin, E., Tamanna, M., Enck, W., Williams, L.: Software bills of materials are required. are we there yet? IEEE Security & Privacy **21**(2), 82–88 (2023)
63. Zalewski, M.: American fuzzy lop (afl) (2012), https://lcamtuf.coredump.cx/afl/