



The six rules of secure software development

**Code Responsibly:
Developers' Blueprint for Secure Coding**

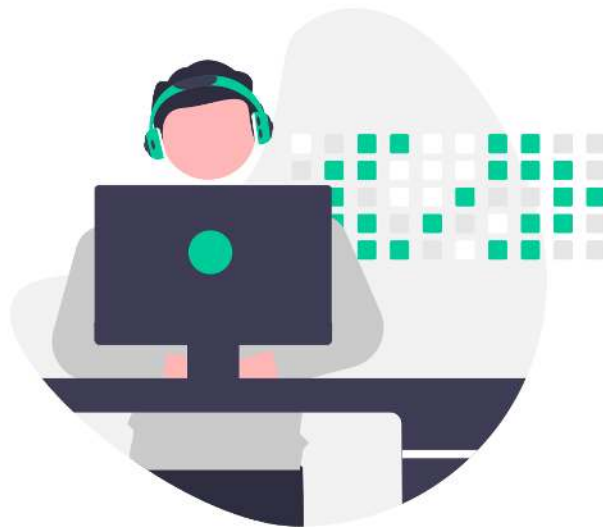




The six rules of secure software development

Software is more important than ever – our connected world’s beating heart is made of it. Unfortunately, as the importance of software increases, so does the activity of cybercriminals and other bad actors trying to make a profit at the developers’ expense. The Department of Homeland Security has long claimed that [90% of security incidents are a consequence of defects in the design or code of software](#). Many developers are unarmed against this onslaught – the number of new vulnerabilities discovered in software has been steadily going up each year since 2016 and this trend is [showing no signs of slowing down](#). If anything, the process is accelerating at a worrying rate. But this doesn’t mean the situation is hopeless – far from it! Many of these security problems have been known for a long time and we have a long list of industry best practices to help deal with them. In this eBook we introduce our six rules of secure software development that present the most important things you can do right now to stem the tide.

Cydrill 2024





The six rules of secure software development

Table Of Contents

Shift left	3
Adopt a secure development lifecycle approach	4
Cover your entire IT ecosystem	6
Move from reaction to prevention	9
Mindset matters more than tech	10
Invest in secure coding training	12
About Cydrill	15

1. Shift left

The rule of 'shift left' has turned into a bit of a buzzword in the last 7-8 years. Like the rest of these six rules, this is not a great revelation or a closely-held secret – in fact, the concept of shift-left testing was originally [coined in 2001 in a Dr. Dobb's article by Larry Smith](#). Back then, 'shift left' referred to testing early and often to find defects as early in the SDLC as possible – literally shifting activities to the left in the V-model of software development.

So, what does this have to do with security?

The idea is simple: move security considerations earlier in the software development lifecycle. Obviously, the earlier a security issue is discovered, the cheaper it is to fix it. Programmers shouldn't just rely on security experts to "do security stuff" a few weeks before shipping the code, but each team member should be actively involved with preventing, finding, and eliminating potential vulnerabilities during development. Of course, this only works if developers actually have the necessary security expertise! This makes understanding the potential threats and best practices (and thus, secure coding) absolutely critical for everyone: all architects, developers, testers and ops folks, not just a few chosen security champions.

This makes understanding the potential threats and best practices (and thus, secure coding) absolutely critical for everyone: all architects, developers, testers and ops folks, not just a few chosen security champions.



2. Adopt a secure development lifecycle approach

It is tempting to deal with software security as an ‘add-on’ to the process: a brief penetration test just before release, or maybe a two-week security review at the end of a project. But as discussed before in the context of shifting left, the later we deal with a security issue, the more expensive it gets. And, unfortunately, a lot of security issues stem from decisions made at an early stage of development such as design or even requirements specification!

We can solve this conundrum by building security in: instead of just ‘doing security’ at a certain point in the development lifecycle, we introduce security activities throughout the entire software development lifecycle (SDLC). This is an established best practice popularized within Microsoft via the [MS SDL](#) (Security Development Lifecycle) as well as security experts via the [BSIMM](#) (Build Security In Maturity Model) or the [OWASP SAMM](#) (Security Assurance Maturity Model):

- **MS SDL** is the most prescriptive of the three – which makes sense, considering it was a process that Microsoft originally developed for internal use in the early 2000s. Its 12 main practices cover security training of all stakeholders, the creation and maintenance of security requirements, threat modeling via data flow diagrams (DFD), secure use of cryptography, managing the risk of third-party components, heavy use of automated tools (SAST, DAST, SCA) and incident response.
- **BSIMM**, on the other hand, is a descriptive model. It is released every year, containing data about what companies are doing these days to improve their security and provides a scorecard to measure your company’s security posture. Then you can figure out which of those activities are most reasonable to implement in your specific context. The activities are grouped into 4 domains: Governance (managing a software security initiative with training as one of its three pillars) Intelligence (threat modeling and proactive security guidance), SSDL touchpoints (building security into development via design and code reviews as well as security testing), and Deployment (secure configuration and maintenance).
- **OWASP SAMM** is also a prescriptive model, giving concrete guidance in various categories, depending on what maturity level (1 to 3) the company is aiming for in the area of Governance (improving security at the organizational level- via education and guidance among others), Design (security requirements, secure design and threat modeling), Implementation (secure build and deployment including vulnerability management), Verification (manual and automated security testing and reviews), and Operations (incident response, hardening and patch management).

As for validating the real-world use of these models: the longitudinal analysis in BSIMM 14 (2023) shows that companies are steadily improving their security posture. In particular, after adopting BSIMM, companies tend to implement a secure SDLC, scale it with the development of security champions, create (and enforce) a security policy, and manage the risk of third-party components. The two priorities after these are threat modeling and security training for engineering teams. As a matter of fact, training engineers on security is emphasized in all of the above models: it is the very first practice in SDL and is part of Governance in both BSIMM and SAMM.

As a final note, penetration testing is often brought up as a one-size-fits-all solution. It is true that a quick and focused test to identify vulnerabilities in the system is useful as an ‘acid test’ before release. But over-reliance on penetration testing is [quite dangerous](#), and it is not a real substitute for secure software development! On the other hand, training developers in security is included in each of these secure SDLC models, with good reason.



3. Cover your entire IT ecosystem

When we're talking about securing code, we don't just mean the code specifically written by you – but also all third-party code that's included in the application. [What are weak links in the npm supply chain?](#) Zahan et al (2022) points out that 80% of all code in modern software comes from third-party packages! That is a massive attack surface, and ultimately the hackers don't care where the weak point in the system is and how it got there. If a third-party component is vulnerable, they'll exploit it just the same – as it happened with the Log4Shell vulnerability at the end of 2021 that impacted almost every Java application – and thus, Java developer – in the world.

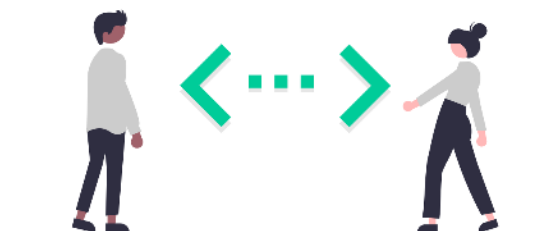
Not to mention that it is also lucrative for attackers to perform supply chain attacks: injecting malicious code into one of the open-source packages (or replacing them entirely). This can be difficult to notice if the package in question is, maybe, a [forgotten dependency-of-a-dependency-of-a-dependency](#) somewhere. The attack trends support this as well: according to the paper, supply chain attacks against applications (not just talking about npm here!) have increased 650% in 2021 alone. The [SolarWinds supply chain attack](#) against the United States government was so impactful it has [shaped the country's cybersecurity strategy](#) as a whole.

These issues are exacerbated in the container world – for example, the 'Red Kangaroo' study has found that at the end of 2020, 80% of all images on Docker Hub were found to contain at least one known vulnerability, with 51% of all images containing critical vulnerabilities!

We like to say that

“vulnerabilities in third-party code are not your fault, but they will definitely become your problem”.

You definitely need to have vulnerability management processes in place to identify, assess, and deal with vulnerabilities discovered in any of the program's dependencies – and a strategy on how to release security patches and even hotfixes if the situation calls for it.



4. Move from reaction to prevention

Discussing code security goes hand in hand with robustness and resilience. Resilience implies a system that is not significantly impacted by failures (limiting the amount of damage they can do, and making it possible to recover from them), while robustness implies a system that anticipates failures and prevents them from happening in the first place. Even though both of these are important, preventing an incident is always better than reacting to an incident after the fact!

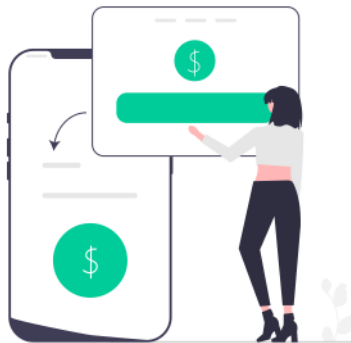
There are two philosophies to ensure robustness and resilience that are sometimes said to be opposites of each other: design by contract and defensive programming.

- Design by Contract (DbC) defines so-called contracts for functions to declare expected preconditions, postconditions and invariants – and works under the assumption that these contracts will not be broken. These contracts are frequently implemented via asserts (not present in production code) and in case there is a failure at runtime, they are typically handled via exceptions. In type-safe languages, DbC may be a built-in feature of the language itself that won't even allow compilation if the contracts can be violated. Rust is a good example for this.
- Defensive programming assumes that any interaction with the system may be incorrect, erroneous, or even malicious. To this end, the developer should explicitly implement input validation in functions that process user input of any kind. Input validation means the implementation of checks that verify that the received input corresponds to the developer's expectations. This should happen in the context of the specific function, “there and then”, right before the input is to be used. If the input fails these checks, it is rejected, so that no piece of code will be executed with unexpected inputs it is not prepared to handle.

Design by contract seems to be better for code efficiency and maintainability – after all, implementing defensive programming techniques requires writing additional code, which adds complexity and is itself a potential source of bugs. But when we look at code security, the goal is to reduce the attack surface and thus guard against intentional misuse, which is exactly what defensive programming provides. Furthermore, reacting to a bad input after it's already been processed is much more dangerous than proactive input validation that can catch it beforehand. This is recognized by many secure coding standards (see e.g. MISRA C:2023 Directive 4.14)

Just to reiterate: in security, preventing an error is always better than catching the error after it has already happened!

As an example, consider processing an XML document describing a money transfer. Following DbC, we can define a 'contract' (an XML schema) and make sure the input conforms to it. This prevents many different attacks (e.g. the attacker duplicating tags, or specifying a negative value for the money transfer). But not every kind of bad input can be covered by a schema. Just a few examples: the attacker can send us a document that references a nonexistent user, performs XXE, contains an invalid transaction date (e.g. 2 years in the future), or performs a cross-site scripting attack against the recipient by specifying a comment like `<script>alert('hacked!')</script>`.



This doesn't mean that design by contract is bad – in fact, those techniques are very useful, but they need to be combined with defensive programming techniques to effectively protect against vulnerabilities. Whenever code security is concerned, input validation is perhaps the single most critical thing you can do according to experts – it's the first category in the [Seven Pernicious Kingdoms](#) and its [improper use](#) comprise the root cause of many other vulnerability types; it is [#5 on the OWASP Proactive Controls \(OPC\)](#) list, and also has its own [cheat sheet on OWASP!](#)

Even redundancy isn't necessarily a dirty word here – in fact, validating the same input multiple times (in different parts of the code) is an example of **defense in depth**, which is an essential protection principle. For example, even if the XML schema ensures that the money transfer value isn't negative, the function doing the transfer should still have a sanity check on the value to be transferred. We should simply accept that everyone makes mistakes, and the code should be always prepared for that.



5. Mindset matters more than tech

If you ask anyone “what do you do to prevent cyberattacks?”, it is likely the answer will be “firewalls and IDS”. It’s true that web application firewalls and intrusion detection systems are important (see A9 in the OWASP Top Ten 2021!), but they won’t solve the problem of vulnerable code.

They may mitigate the effects of already existing vulnerabilities and make exploitation of these vulnerabilities more difficult, but even in that arena the attackers are constantly coming up with new ways to get around perimeter defenses (e.g. [Server-side Request Forgery aka SSRF](#)) and evade WAF filters to deliver their payload.

As a matter of fact,

no firewall could stop the exploitation of zero-days like Heartbleed or Log4Shell before it was already too late.

But how do we deal with vulnerable code, especially in codebases that have been around for decades?

The sheer amount of code that developers must deal with is increasing rapidly. Sourcegraph’s [The Emergence of Big Code](#) (2020) shows that developers have to work with remarkably more code than ever before: 51% of participants claimed the amount of code at a company has increased by a factor of 100 compared to the previous 10 years, and over 90% of them said coding velocity and the value of the code itself has also increased drastically. In order to find, fix, and prevent vulnerabilities, developers need to be responsible for them and take ownership of the code in question – that can be a challenge by itself in these massive codebases.

And then there is legacy code...

Some companies are looking at AI to solve this problem by automatically identifying vulnerabilities or just making sure all code is secure. Putting aside the [nascent and vulnerable nature of machine learning applications](#), this ultimately relies on these AIs being able to write secure code by default. But right now, that goal is far out of reach. Let’s face it: we’re still light-years away from achieving flawless AI-generated code. Consider that the models are mainly trained on the ‘wisdom of the masses’: open-source projects and popular third-party Q&A sites such as Stack Overflow. Such sources have been hotbeds of vulnerable code examples in the past (see [Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security, Fischer et al, 2017](#)).

As always: garbage in means garbage out.

On the other hand, it doesn't help to put the responsibility for security on developers' shoulders while failing to give them the necessary resources and support for it.

[Bruce Schneier](#) pointed out in 2019 that even though 68% of security professionals believe it's a programmer's job to write secure code, they also think less than half of them can actually spot security holes.

GitLab's yearly [Global Developer Report](#) from 2022 underscored this as well: as DevOps transforms into DevSecOps, security is becoming the #1 concern. More importantly, now that 43% of "Sec" teams are fully responsible for security, despite the vast variety of tools at their disposal they feel much less optimistic and confident about this responsibility than the "Dev" and "Ops" part of the triad (56% vs 76%!). Automation is not going to solve the problem by itself. It isn't a coincidence that DevSecOps folks sometimes call SAST tools 'False Positives as a Service'.

Tools are handy and valuable, but there is no substitute for human expertise.

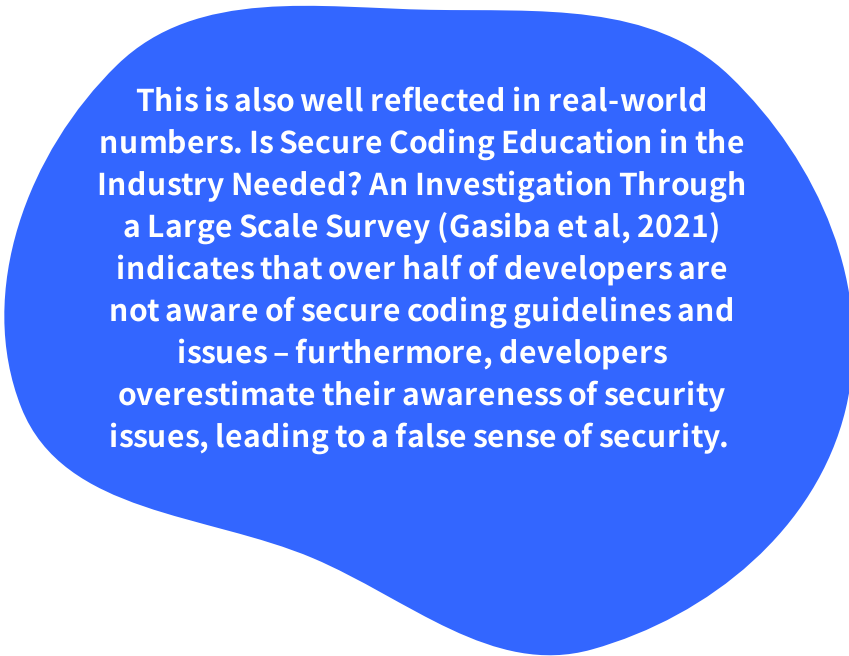


6. Invest in secure coding training

As we've seen so far, there are two challenges in cybersecurity today: how to deal with issues from the past (unknown vulnerabilities in existing code, legacy code, and third-party code) and how to deal with issues in the future (vulnerabilities in all code written by the developers from this point on).

For the first question, we have lots of answers: various code analyzers, testing tools, and vulnerability management. However, for the second question, the only realistic answer is writing code that is free of such vulnerabilities. And that's not something a tool can do for us.

The only solution is education: making developers aware of these security problems in all phases of the SDLC and giving them the necessary mindset and skills so they will be able to avoid them (and spot them in existing code).



This is also well reflected in real-world numbers. *Is Secure Coding Education in the Industry Needed? An Investigation Through a Large Scale Survey* (Gasiba et al, 2021) indicates that over half of developers are not aware of secure coding guidelines and issues – furthermore, developers overestimate their awareness of security issues, leading to a false sense of security.

The best method to address this discrepancy is through secure coding education supported with hands-on exercises. Developers need to see vulnerable code in action, see the (often devastating) consequences of vulnerability exploitation, and then actually fix the vulnerable code themselves. Only this way will they acquire the needed skills and fully understand and retain knowledge about these vulnerabilities.

CTF - Capture the flag

Capture the flag (CTF) events and platforms are popping up as a popular alternative in this area. CTFs are popular when it comes to improving the offensive skills of cyber security experts: they are fun (and gamified out-of-the-box), they provide realistic hacking scenarios, and they help establish the ‘hacker mindset’. But when it comes to defensive best practices and establishing company-wide secure coding initiatives, they have pretty clear deficiencies compared to real training: a relative inability to cater to developers without prior experience in security, weak (or even negative) motivation for developers less interested in competition, and poor coverage of ‘less cool’ (but still critically important) security issues.



Sometimes microlearning is also brought up as a possible solution: teaching about security issues in small bite-sized (even just 5- or 10-minute) videos or brief activities that programmers can check when they first encounter such an issue or just during their free time (if such a thing exists at all).

But secure coding is one of the areas where this doesn't really work. As per Amy Fox's 2016 article [Microlearning for Effective Performance Management](#):

“Microlearning is not a panacea for every training need. If an employee is learning something for the first time, particularly a complex skill, individual coaching or another form of more intensive training may be best. Microlearning often is best used for reinforcement to help learning stick and to build up employees' skills.”

In the context of secure coding, microlearning can be effective only as a reinforcement technique once developers already know about vulnerabilities and best practices – in other words, once they have already taken part in an in-depth training course.

And that's exactly what we believe in: with blended learning, developers should first establish a deep foundation for secure coding in their programming language(s) of choice via an instructor-led training course. And once this is achieved, they can follow it up with regular monthly 'bite-sized' e-learning modules to keep their skills sharp and up to date.

Finally, a note about gamified capture the flag (CTF) events and platforms. CTFs are popular when it comes to improving the skills of cyber security experts: they are fun (and gamified out-of-the-box), they provide realistic hacking scenarios, and they help establish the 'hacker mindset'. But when it comes to learning about secure coding, they have pretty clear deficiencies compared to blended learning: they tend to focus on 'fun' attack scenarios and thus ignore many common vulnerability types, they aren't adaptive to the needs of individual participants, and their competitive aspects can actually have a negative effect on motivation. On the other hand, blended learning also drives high engagement without having to lose the benefits of gamification. If you're interested in the details, we have analyzed these limitations in a separate article: [CTF in secure coding education – a critical look](#).



About Cydrill

Established in 2019 and recognized by Enterprise Security in 2021 as one of the top companies shaping the cybersecurity landscape, Cydrill is on a mission to tackle the root cause of poor cyber-defense: inadequate coding practices.

Cydrill's blended learning journey provides training in proactive and effective secure coding for developers from Fortune 500 companies all over the world. By combining instructor-led training, e-learning, hands-on labs, and gamification, Cydrill provides a novel and effective approach to learning how to code securely.

Learn more

about our courses and learning
environment