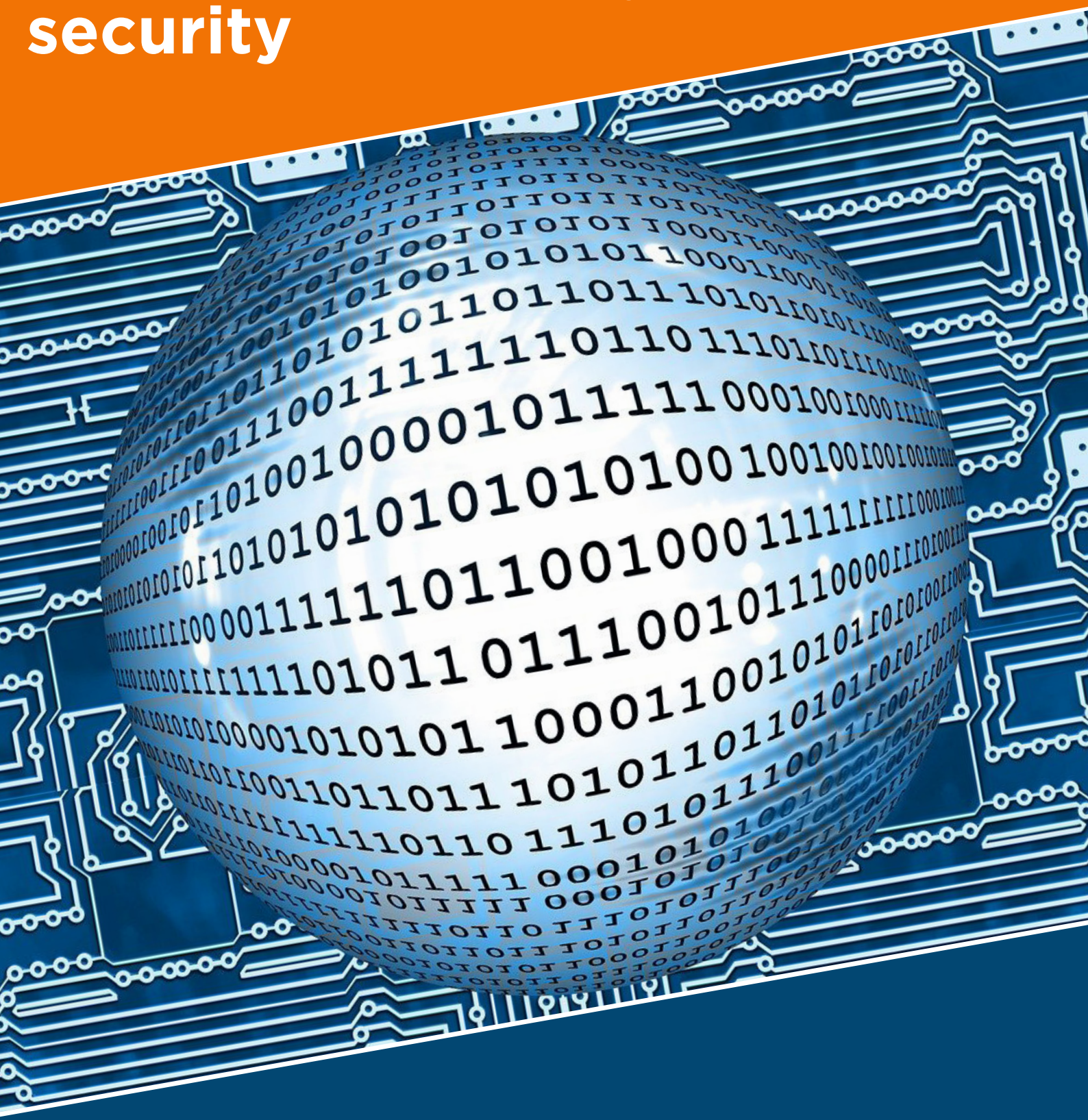
Industry best practices for software safety & security



Hardening software with security by design

TRUST IN SOFT

INTRODUCTION

"Software is eating the world!"

More than ever, software is powering all essential operations in our life from vehicles to white goods. A modern car is said to contain more than 100 million lines of code. And with the ever more present 5G, functionalities are drastically virtualized and migrating to the application layers. 5G also contains more software than all previous network generations combined.
And it is not ready to stop! As autonomous activities are increasingly adopted in all industries, the software is set to continue its expansion at a rapid pace and will continue to pilot many aspects of our lives.

However, this trend raises an essential question: how to craft safe, secure, and reliable software? Bugs are and were always part of the vocabulary of a developer. "It's impossible to write bug-free software," they claim. But with software piloting more and more essential aspects of our lives, can we allow ourselves to compromise on the presence of bugs? (spoiler alert: no !)
In this white paper, we will review the industry best practices and will challenge them in the light of new trends in software: increased size, complexity, integration of third-party open-source bricks, criticality, and value of standards.

Security and safety are the new cornerstone for software

Writing software comes with an important responsibility. The program needs to work precisely according to specifications whatever the conditions and external inputs. No user input should break the integrity of the code or should lead to a program behavior that was not planned by the developer. Security and safety are the two obsessions that any developer should have when developing software. Security ensures the software has no vulnerabilities or no behaviors that could be used by an attacker to inject code or take the control of the program. In addition, safety ensures the software works in a reliable way, according to specifications, and never crashes.

IT security products are not enough to guarantee rock solid security

Software cybersecurity has always been an important concern and an area for focus. However, in recent years, as software takes an exponentially increasing place in our lives, cyberattacks have increased proportionally. Attacks such as SolarWinds in the US and on a hospital in Germany are just recent but dramatic examples and are setting the trend for the coming years.
Fortunately, many mechanisms are in place and efficiently protect software-based products.
For example, server hosting products can be protected by firewalls, proxies, or advanced logs analysis. Access and connection can be managed through password mechanisms or certificates. Communication with servers can be encrypted or use SSL/TLS.
But, let's think about this.

We can compare the software to a castle that is surrounded by high, impenetrable walls. Blocked by these walls, logically no intruder can get to the castle. Unless, of course:

1. A secret underground tunnel access exists, from the nearby forest. If the castle architects forget to block this entrance, the walls protecting the castle are useless, since the attacker can gain access by this other route, no matter the size of the walls. Thus, if a vulnerability in the core software exists, it can be exploited by an attacker no matter the other IT defenses.

2. The castle walls have flaws that allow the intruder to break in. IT defender software can only be as efficient as its own source code. The takeaway here is that the best way to ensure a product's security is to make sure its software and the software protecting it are as secure as possible, without 0 days that can be exploited.

Let's see what are the industry best practices to ensure software security.

The Story of Heartbleed

Heartbleed, discovered in April 2014, is a software vulnerability that became infamous for allowing cyberattackers to gain access to sensitive information. It affected thousands of web servers, including those that were running high traffic websites, like Yahoo. Heartbleed resulted from a flaw in OpenSSL, an open-source code library with Transport Layer Security (TLS) protocols that provide security for communications over a network. The flaw enabled a hacker to trick a vulnerable web server into sending information like usernames and passwords. Though this vulnerability was not in the TLS standards themselves, due to its widespread use OpenSSL triggered a major cybersecurity crisis. 17% of all servers in the world were affected once the vulnerability was exposed to the public, and served as a wake-up call for code security.

The flaw in OpenSSL that led to Heartbleed resulted from a bad implementation of the heartbeat functionality (how computers communicate with each other when a user is temporarily inactive, in order to reassure that a connection is still in place). With the OpenSSL implementation, however, the computer that received the heartbeat request did not check that the request had the same size as was stated in the request. This enabled extra KB of data to be extracted from the webserver by a hacker, containing sensitive information.

Why you can't rely on common processes to ensure the security of the software

90%

of cyberattacks result from vulnerabilities in the source code.

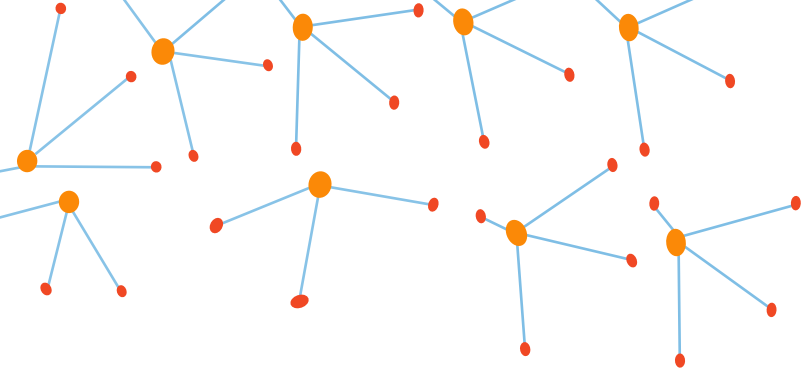
The processes mentioned in the prior section are great practices for integrating security into the software, but according to the U.S. Department of Homeland Security, 90% of cyber attacks result from vulnerabilities in the source code (DHS ,2015). The numerous recent cases in the news demonstrate that vulnerabilities such as buffer overflow can be exploited by cyber attackers at any level of the software stacks: on the electronic component, OS, communication or encryption software stacks, all the way up to the application level itself. And, according to MITRE, buffer overflow is the most common and critical software vulnerability, allowing attackers to execute arbitrary code remotely, read sensitive information, or cause the system to crash.

Why performing software tests in addition to system/functional tests is essential

Functional tests are very efficient in validating the overall software system against functional requirements and specifications. Using a "black box" approach, they allow testers and QA teams to validate the system for different inputs and check if the outputs match the specifications.Although efficient, this approach is not exhaustive and the functional tests can not guarantee the product against vulnerabilities.
Because of the black-box approach, developers have a more limited understanding of what exactly is happening during code execution, especially complex-ones. Although functional tests can validate the software in most common conditions, they might not cover all possible behaviors.
Another excellent way to harden the quality and security of software is to write software tests to assess individual chunks of code such as functions or modules and check they produce expected results. Although very efficient, it becomes more difficult to scale test coverage especially when programs are long and complex.

ISO and coding standards & the latest regulations are essential but they do not ensure perfect security & safety/reliability

ISO/IEC norms (like ISO 26262 in the automotive industry or DO-178 in the aerospace industry) and coding standards like CERT C specify a recommended or mandatory process to write better code, in order to better protect a product's users from safety and security concerns. Indeed, these norms, standards, and the latest industry regulations improve software security and reliability by limiting the possibilities for software weaknesses. However, these safeguards cannot ensure perfect security and reliability because just following their prescribed practices does not ensure the complete absence of vulnerabilities and unexpected software behaviors/ bugs in the source code itself.



Industry best practices to write secure and reliable

Eliminating bugs and unexpected software behaviors in the source code early in the development process ensures both security and safety while reducing costs

In the US, companies spent an estimated collective $607 billion on finding and fixing bugs in software in 2020 (CISQ, 2021). Identifying and correcting bugs in software already in production can cost up to 10 times more than fixing them during the development process. In fact, the earlier a bug is detected and fixed in the development cycle, the more a company saves. Moreover, it is not always easy or possible to update the software in the field. So, not only will you ensure software security and safety by resolving weaknesses early on, but you will also reduce long-run costs associated with vulnerabilities and defects.

Undefined Behaviors: the programmer's nightmare

Buffer overflows, non-initialized memory access, division by zero, access out of bounds, invalid pointers usage are just some examples of bugs that are software undefined behaviors. They are the software programmers' worst enemy.
C and C++ continue to be the most broadly used languages in embedded/electronic systems. Fast, efficient, and memory-optimized, C and C++ are powering the brains of modern products in various sectors including telecoms, IoT, medical, manufacturing, automotive & transportation, semiconductors, and more. But this power comes with a price: although C and C++ allow for great flexibility in programming and in managing memory, they can also lead to hidden defects if mismanaged. Those defects can either create reliability risks or leave the possibility to hackers to exploit them and crash the program, steal data or inject malicious code.

What tools allow to catch and eliminate Undefined Behaviors?

Detecting and catching vulnerabilities and Undefined Behaviors once the software is in production is time-consuming, costly, and bad for the reputation. It is therefore essential to ensure that those possible bugs are eliminated during the development process. Compilers are able to detect very basic issues such as simple syntax errors.
Static analyzers reason on the syntax of the code and are able to catch more complex code constructs. However, this comes with a price on numerous false positives and false negatives increasing verification time.

Dynamic analyzers such as sanitizers can detect some additional memory management-related issues, however, they remain quite complex to implement and use and their analysis is not exhaustive. Although these tools have limitations, it is an excellent best practice to use them as first steps to limit the presence of bugs. But again, the lack of exhaustivity needs to be carefully taken into account.

On the importance of exhaustively testing the source code to ensure it has a deterministic behavior

What would be the consequences if software crashed or did not work as planned? We always think about potential health/life impacts in "critical" industries, for instance in the automotive and medical industries, industrial cobotics, etc. Though in fact, it is in all industries that software safety and reliability may be heavily impacted by undefined behaviors.
For example, unexpected software results/behavior may lead to a wrong billing in the Smart Metering industry, a wrong decision when unexpected inputs are handled by the algorithm in manufacturing, or service discontinuation or bad user experience in the Telecom, IoT or security industries when software crashes.

In the end, it is about efforts versus gain. Depending on the target market, it may be necessary to detect all undefined behaviors exhaustively and ensure none of them are left in the code. In other industries, it is a matter of optimizing the time and resources spent during verification efforts to ensure that Time to Market is achieved while most bugs have been fixed. Because C and C++ languages allow for interaction with hardware at a very low level, another essential aspect to factor in is material defects. A material defect in the field may provide wrong information to inputs of the software. Is the software robust enough to cope with a material defect? Do the tests performed during the design stage ensure the software is able to resist the infinite complexity of real-life field conditions? Ensuring software does not crash whatever the inputs start with detecting and fixing undefined behaviors.

Likewise, ensuring software has a deterministic behavior (ie, for the same inputs, the output will always be the same) starts with detecting and fixing undefined behaviors. Being able to test as many inputs as possible during the software design stage is key to ensuring software safety and reliability.

Why software tests on third-party software are essential to ensure the safety and security of the overall product

More and more companies across industries rely on integrating third party software stacks around their own software. Some may argue that bugs and undefined behaviors present in the third party software are to be fixed by the third party. It is important for vulnerabilities in third-party software to be tracked and fixed by companies integrating them in their own software. Wouldn't it be a shame to have invested massively in the security of the software developed in-house and face an attack due to a flaw in the third party software?

Cyber-attackers also run their own ROI analysis and focus on widespread software stacks in order to maximize their impact. As an example, vulnerabilities revealed in June 2020 in the Ripple 20 Treck TCP/IP library could impact hundreds of millions of IoT devices in the field. In any case, even when using third-party software stacks, companies also often write some piece of code around them, for instance to provide inputs and handle the outputs from the third-party software. This part of code around the third party stack is especially sensitive and needs to be carefully checked.

Want to learn more?

In this white paper, we addressed the current software testing best practices that play an important role in the safety and security of software, but cannot ensure it. Popular tools including syntax-driven static analyzers, dynamic analyzers, and sanitizers are helpful for partially detecting software defects. However, they are not enough to guarantee software security and safety.

Interested in

- Ensuring your software is immune from security flaws, behaves in a deterministic way and does not crash whatever the inputs?
- Learning how to drastically reduce your software test efforts and quickly generate the equivalent of billions of tests just by generalizing the inputs of one of your current tests?
- Hearing about how to get mathematical guarantees on software security and safety?
- Not only exhaustively detecting the bugs but also getting useful hints on their root cause?
- Or just willing to chat more about software best practices?

We would love to hear from you!

Contact us at contact@trust-in-soft.com

2415 Third Street, Suite 231
San Francisco, CA 94107,
USA , +1 (408) 829-5882

222 cour Avenue du Maine
75014, Paris, France
+33 1 84 06 43 91

Reference

Consortium for Information and Software Quality (CISQ). (2021). Cost of Poor Software Quality in the US: a 2020 report. CISQ.

Department of Homeland Security - Cybersecurity. (2015). Software Assurance. https://us-cert.cisa.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf