Wireless Personal Communications
https://doi.org/10.1007/s11277-023-10431-2

A Call for Mandatory Input Validation and Fuzz Testing

Geir M. Køien[1] · Lasse Øverlier[2]

Accepted: 6 April 2023

Abstract
The on-going digitalization of our critical infrastructures is progressing fast. There is also a growing trend of serious and disrupting cyber-attacks. The digital services are often fragile, and with many weaknesses and vulnerabilities. This makes exploiting and attacking the services a little too easy. If the services verifies all inputs, many security threats will be avoided. Similarly, if one diligently tests the services with malformed inputs, one will uncover many security and software quality problems. In this paper we investigate "input validation" and "fuzz testing" as a means to improve security. The aim is not exhaustive coverage, but to provide indications of usefulness and to serve as a call for action.

Keywords Security threats · Critical infrastructures · Input validation · Fuzz testing

1 Introduction

1.1 Background and Motivation

Increasingly, our critical infrastructures are *digital* critical infrastructures. Given that the infrastructures generally are accessible on the internet, directly or indirectly, they will also have global exposure.

The users may or may not be honest. And, there will also be criminals and nation-state actors looking for illegitimate access. We shall not investigate the threat actors as such, but note that their existence should in itself be an argument for strong security protection. This protection will generally include entity authentication and authorization (access control), to ensure that only legitimate users gain (authorized) access. Public services will tend to have many many users, and in general one cannot assume much trust in legitimate users. Neither can one assume that the legitimate users are trustworthy (as in being able to act honestly).

✉ Geir M. Køien
geir.koien@usn.no

Lasse Øverlier
lasse.overlier@ntnu.no

1 Department of Microsystems, University of South-Eastern Norway, Horten, Norway

2 Department of Information Security and Communication Technology, NTNU, Gjøvik, Norway

Invariably, the system must *always* check that input data is valid and appropriate. In our coverage, there are two complementary aspects to improving the handling of input data: *Input validation* and *Fuzz testing*.

1.2 Handling of Input Data

Every useful program needs to have input. Historically, a main culprit in erroneous input handling is missing input validation for *buffers*, causing a buffer overflow. The overflow vulnerability may be exploited by the intruder. An early example is the so-called Morris Worm[1], which exploited a buffer overflow. Another example is the famous Aleph One article [2], where the basic principles behind an overflow exploit were explained. In short, by overwriting a buffer with too much data, the overflowing data will be written to a part of memory that is reserved for other uses. In particular, it might affect the memory used for storing the *return address* of the calling function. The result is that the processing may *return* to execute code injected by the intruder, and not the code at the "normal" return point. This will potentially allow the intruder to run arbitrary code. A recent survey paper on techniques to bypass mitigation techniques for buffer overflows illustrates that buffer overflow exploitation is still a big problem [3].

Another input vulnerability type that has been around for many years is the SQL-injection database attack. The core of the problem is that textual input data *may* be interpreted as SQL code by inserting *escape* characters in the data. Thus, if you do not validate your input before using it in a database query, the malformed input may delete/insert/modify entries in the database, or in some cases even execute programs on your database server. A recent survey of web vulnerabilities highlights SQL injections as a main threat vector [4].

1.2.1 Input Validation

This covers efforts to check and validate the input data. We here address input to functions and input via protocol exchanges. The input handling code must know what kind of data to expect as input. One should now the *type* of the data. Additionally, one must check the *range* and/or *length* of the input data. For simple data types, this many be fairly trivial, but it can be challenging for more complex data types [5].

Then there is the common practice to "clean up" data by applying so-called *sanitizer* functions. Sanitizer functions are often used to remove redundant whitespace characters, etc, and to transform input to a uniform representation (for instance by ensuring that date formats are uniform). Sanitizers are also used for removing escape characters. Sanitizers are not a replacement for validation, and should be run prior to validation checks.

1.2.2 Fuzz Testing

Had input validation always been designed and implemented correctly and comprehensively, then there would be no problem with wrong or malformed inputs. Software is rarely perfect, and even if it were, in a dynamic world the requirements change over time. It is therefore increasingly likely that the input validation will come to be incomplete or inconsistent.

Thus, there is an obvious need to test the input validation procedures. We shall assume that the input validation software is subject to all the normal software quality assurance methods, including function tests, etc. In addition, we strongly advocate "fuzz testing" the code. The fuzz testing concept dates back to 1988 [6]. At its basis, fuzz testing is a remarkably simple concept, focusing on providing malformed (or randomized) data to the input handler, and see how it tackles the challenge. The trick is to automatically generate lots and lots of invalid input data and run the software over and over again with different input data [7]. Chances are that the input handler will experience some kind of failure. Most of these failures will probably not be security sensitive as such, but they certainly point to a software quality problem. If the fuzz generator has knowledge about the expected inputs, it can focus its malformed input data to stress the input handler on areas that are known to be problematic.

1.3 Structure of the Paper

Chapter 1 includes the general introduction and motivation. Chapter 2 details the need for proper input validation. Chapter 3 is a high-level survey on modern fuzz testing techniques. Chapter 4 is a high-level survey on input validation techniques. Chapter 5 is a discussion, and chapter 6 contains a summary and a conclusion. Other works are cited throughout the paper.

2 On the Need for Input Validation

2.1 The Software Security Crisis

The term "software crisis" dates back to the early days of computing science, and was used to describe the difficulty of writing quality software in a timely manner. The crisis arose because existing development methods were insufficient, inadequate and inefficient. The term was coined at the first NATO Software Engineering Conference in 1968 [8]. Subsequently, there has been other "software crisis" moments.

Currently, we seem poised to approach yet another software crisis moment. The current crisis is not about the speed of developing new services per see, but rather about the fragility and vulnerability of the modern software. This comes along at the same time as societal dependency on software services is on the increase. Thus, our new digital infrastructures, relying heavily on software, are fragile and vulnerable to attacks. The crisis runs deep, as one now increasingly "softwareizes" network functionality and services that used to be performed by hardware. A case in point is the 5G networks, which crucially depends in Software Defined Network (SDN) and Network Function Virtualization (NFV) technologies.

2.2 Societal Dependency and Vulnerability

In an increasingly digital society, many of the infrastructures are becoming digitalized. This is beneficial in many ways, including providing always-on services and enabling a global reach. Of course, there is another side to this in that the global always-on aspects also means that the infrastructure now becomes globally exposed to cyber attacks 24/7/365. There is thus an extraordinary need for security protection of these services and infrastructures.

Despite this, the services are often poorly protected and vulnerable to many different types of attacks. In the last few years we have seen a pronounced trend in so-called ransomware attacks, in which the intruder takes over the services, encrypts the systems and leave them inoperable. Then the perpetrators demand a ransom to provide the decryption key. The ransomware actors are generally technically competent, and use a number of techniques and tools to gain access to the systems.

There is no single cause for the current state of affairs regarding the lack of strong security, and there is not going to be a single measure to fix it either. However, some measures will likely have a pronounced positive impact, and be both effective and efficient. For our purpose, we focus on improvement to the handling of input data. As per the “OWASP Top 10” list, incorrect handling of data input is featured in several of the top 10 categories [9]. Thus, if we are able to improve on the input data handling, many vulnerabilities will be effectively mitigated. And, of course, there are additional quality and reliability advantages to correct handling of input data.

2.3 Development vs. Deployment

Fuzz testing is, as the name implies, a test activity. As such it is part of the software development phases, and explicitly part of the security tests [10].

Input validation, on the other hand, is designed to be a permanent feature of the software. It will thus be part of the deployed software, and will be running as the software is used by end-user/clients. It is noted that it is generally orders of magnitude less expensive to fix a flaw during development than in production. It is additionally noted that the costs of fixing a flaw prior to deployment will primarily affect the software developer.

Additionally, it is generally considered useful to distinguish between design flaws and implementation “bugs”. In general, the impact of a bug is localized to a specific implementation and it can often be corrected fairly fast and without side-effects. Correcting a faulty design tends to be more challenging, and it may more easily cause side-effect and even backwards compatibility issues.

2.4 Client-side vs. Server-side

The most common software service model is to have a functionality split between software receding at the user platform (*client-side*) and software being hosted centrally (*server-side*). The server-side and client-side dichotomy indicates that we are dealing with distinct security domains. The standard adage is that one should never assume trust or trustworthiness. Thus, the server-side processes should not assume trust in the client-side input data. As such, it is a “Never Trust, Always Verify” case, a la the Zero Trust tenets [11].

The software at the client-side may have been provided by the server. However, the server-side processes should still never trust the client-side software. This apparently contradictory position is due to the fact that the server-side will not normally have control over the client-side platform (hardware, operating system, browser, etc.). That is, the server-side will not normally have assurance of the integrity of the client-side software.

Of course, the server-side should always authenticate the client-side and verify the access rights, etc. However, the server-side must still check that the input data is according to the expect input. This encompasses both syntactical and semantic properties of the input data, and it explicitly concerns meta-data aspects of the data.

2.5 The Principal Parties

When it comes to software vulnerabilities and impact of a breach, we have chosen to highlight three principal parties. The costs associated with improving the software and the cost from experiencing system breaches are (unevenly) distributed amongst these parties.

- The *software developer*: This entity is responsible for designing and implementing the software. The software developer may be a single individual, an organization or a business.
- The *service provider*: The software *service provider* is the entity that deploys the software, providing services to users. For our purpose, this is a business or organization.
- The *user*: The *user* is the consumer of the software services. The user may be an individual, but it may also be a software agent.

We note that there will tend to be several orders of magnitude more users than service providers/software developers. For public services there will be regulatory and jurisdictional contexts for the entities to operate under.

2.6 The Cost of a Security Breach

2.6.1 Assume Breach

To “assume breach” is one of the philosophical underpinnings of the Zero Trust concept [11]. We shall adopt this as a premise. It is then imperative to have a look at the consequence of a breach.

2.6.2 Cost Range

The first thing to note is that the cost of a security incident will vary considerably. Some incidents will have negligible impacts, while others will incur multi-million dollar losses. The well-known story of how the NotPetya malware inflicted damage and losses illustrates the latter [12].

2.6.3 Cost Distribution

The cost of a breach will not be the same to all parties.

- The *software developer* will suffer costs associated with correcting the software, and handling of the security updating procedure. Due to “limited liability” clauses, the software developer will likely not share the burden of financial losses that stem from the consequences of the security breach. Thus, while the impact may be serious for the software developer, it is rarely going to be devastating.
- The *service provider* may suffer from minor loss of functionality (availability) and all the the way up to full-scale ransomware lockdown of the whole infrastructure. The costs can potentially be very high. For service providers that are ill prepared, a serious cyber attack may become an existential threat.
- The *user* may suffer from loss of functionality, and he/she may loose personal data (privacy) and credit card information, etc. Some of the end-user costs may be covered by

the service provider. The service provider may in turn rely on insurance to cover its costs.

The "limited liability" of the software developer is subject to jurisdictional and regulatory rules. It is increasingly likely to be subject to the software developer having demonstrated that they designed and implemented the software according to "best current practices"[1].

2.6.4 Disparity of Consequences

The disparity in the consequences of a successful cyber attack is to be noted. In business lingo, one may need to find ways to ensure that the costs of an exploit is internalized with the software developers. To put it another way, the developers do not have (as much) "skin in the game" (an extended and somewhat irreverent account of the "skin in the game" concept can be found in [13]). The cost/consequence disparity is probably a significant contributing factor to the current "security crisis". There needs to be incentives that forces the software developers to invest in improved software quality. Of course, the fact that many of the most commonly used pieces of software are open-source only adds to the complications of creating incentives.

As noted in the previous sub-clause, legislation concerning software and liability generally seems to move towards stronger requirements for compliance with "best current practices" and certification of the software development process. These requirements will generally also apply to open-source software. We expect that this will provide an impetus for software developers to document that they have had quality assurance processes in place and that they have conducted a minimum of (security) testing of the software.

2.7 Design and Development Costs

The cost of devising and running fuzz tests is normally covered exclusively by the software developer. The cost of developing input data validation code is also solely covered by the software developer. The cost of running the input data validation code, in terms of computational overhead, fall on the service provider. This cost is usually very low, but for protocol exchanges the processing delays *may* be a factor.

The incentives for the software developer to improve the software quality and security properties is clearly there, but potential development delays and increased man-hour costs will diminish the incentive.

3 Modern Fuzz Testing Techniques

Fuzz testing in its simplest form is a type of "black box testing", where you are able to send arbitrary input to a running program or to the initialization of a program and observe what happens. In Fig. 1 the "Target Software" is being given input from a "Feeder" that is given potential input from a "Generator".

[1] A *Best Current Practice (BCP)* is a de facto performance standard used in engineering and information technology. It is a moving target, and not always well defined.

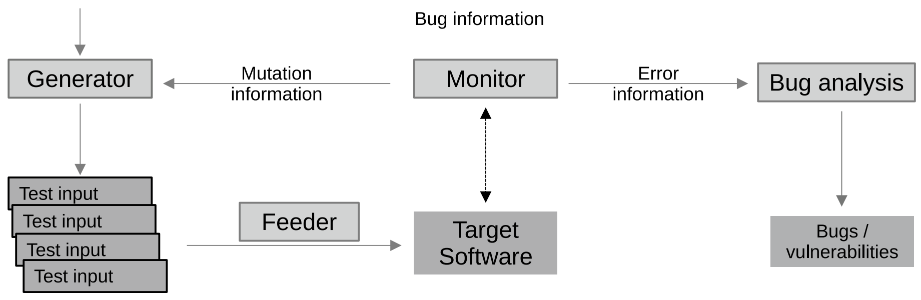

Fig. 1 The principles of a simple fuzzer

The Generator may create random input values, but may also use feedback from the Monitor which is based on the happenings in the target software for the specific input or for a combination of input. Often the Feeder may give input to multiple parameters in the Target Software, or it may also need to feed a specific sequence of input values for the Target Software. This may be required for the Target Software to receive all information it requires to complete its tasks, or optionally malfunction and terminate with an error. The Generator may also utilize feedback from deeper analysis of bugs to create more crafty and specially adapted input. There are several different views of fuzzer architecture and fuzzing techniques, and we will not provide extensive coverage here. The interested reader is referred to [14] for further information.

3.1 Fuzzing Level

We classify the types of fuzz testing into two main types: *open environment* and *restricted environment*.

In the *restricted environment*, comparable to the term "black-box testing", we include a server running programs where the tester has no access to the binary or the source code of the software. This is often the case for larger cloud services where the tester only has access to the observable results through the provided interface, or optionally through other channels, like events (not) taking place elsewhere. Only the server provided feedback, directly or indirectly, in addition to potential error messages given through this interface or observations of service availability will provide information to the fuzz tester.

In the *open environment*, "white-box testing", we have the best analogy with client-side software. Here we include all software where the tester has access to the binary and/or source code, for running his/her own tests with complete control of the environment where the software is running (tested). The essential part is not what type of software it is, but the control of the environment that limits the fuzz testers abilities and possibilities.

In most cases the fuzzing environments is a *hybrid scenario*, more comparable to a "both black and white-box testing", a combination of restricted and open environment. This is not to be confused with "gray-box testing", in which some partial knowledge of the fuzzing environment may be known. Our hybrid scenario is easier to compare with a two-level opportunity with some test objects where the fuzz tester has complete environment control, and some test objects where the fuzz tester does not have any control over the system.

Common hybrid scenario variants occur within many of the services in the *app universe*. These client side apps are often constructed to be local interfaces to some server side program or a cloud service. These services are often built with a centralized service where the tester neither has control over the environment nor access to the source code or binary code. The client side app may be run by the tester in an open environment, but the tester may not have access or knowledge of the interface between the locally controlled client and the remotely service in a restricted environment.

This means that the hybrid scenario fuzz tester may actually have two levels of opportunities/challenges, the cloud service with the more limited interface protocol, and a local app with all the variations the app developer wants the user to have access to. In addition the protocol between the local app and the cloud service may often be more extensive and have many more possibilities of communicating with the server than what the app makes the user believe. This extra functionality may still be available on the cloud service, but was likely never intended to be used by a normal user and may therefore be vulnerable to protocol fuzzing[15]. These added features could be extra services developed for testing and debugging, new functionality for future app upgrades, old functionality not removed, etc.

3.2 Fuzzing Techniques

When an input is being fuzzed there are several techniques that may be used depending on many different scenarios. There will be specific techniques for testing a desktop application, an app on a mobile device, a web service, a protocol interface, a graphical user interface, a command line application, a file format, etc. For each specific input there are many possible values, so-called *input vectors*, that can be given to the running software. That is, the input state-space is a combination of a all possible values of all variables concerned. The *input space* are frequently so large that testing them all is impossible within the lifetime of the universe.

The fuzzer needs to select which vectors to test by evaluating for example:

- What type of input is expected? Numbers, strings, binary values, specific file formats, file names, encrypted data, etc.
- What are the expected "limitations" of the input? E.g. a string of up to 32 characters, a number between 1 and 10, a PDF file, password rules, etc.
- What types of errors are previously known to this type of input format? E.g. a 50 character string, a negative value, a manipulated PDF-file not conforming to the standard, etc.
- What types of variations of the input field are possible? Can we input a longer string than expected, or a negative number, or an invalid PDF, etc.
- Earlier successful and unsuccessful attempts. How to evaluate these earlier results and how to integrate new results with previous experience?
- How to select the next input vector to try? Based on the feedback shown in Fig. 1 we can use both information from the monitoring the process and aggregate information from the analysis of potential bugs/errors to improve and prioritize the list of next input.

In real fuzzing software there are many more input formats, variants, limitations, known problems, etc. But within each of the input fields a fuzzer must know the input space for the input vectors, have some selection and variation criteria, and some way of observing the result of using the selected value.

4 Input Validation

This is only a fairly superficial exposition of the input validation categories and techniques. It is intended only to highlight the importance of input validation. We choose to classify input according to:

- *Syntactical aspects*
- *Semantic aspects*

The semantics are typically harder to validate, and is rarely validated by basic input data validation methods. We shall therefore not investigate this further in the present paper. It is noted that fuzz testing may additionally be used to test semantic aspects. The value of such test cases presumes that one can verify that the response to the input data is appropriate.

4.1 Syntactical Input Validation

The input data to be validated, whether the input to a function call, a command-line argument or a protocol element, will have syntactical properties. The syntactical properties will concern aspects such as distinguishing between items/objects, the order of the items/objects and the formatting/encoding of the objects.

For syntax checking to be meaningful, there must exist a specification of the abstract syntax of the input data. Unfortunately, one often finds that these aspects are poorly defined or only implicitly defined.

It is noted that syntactical input validation is reminiscent of type checking in programming languages. It is also somewhat reminiscent of the guard construct (design-by-contract) in the Eiffel language [16]. Here the input data have explicitly defined type properties (including ranges), etc., and the Eiffel compiler will verify that all conditions are meet.

Another example is the explicit and embedded syntax of the Abstract Syntax Notation 1 (ASN.1). The basics of ASN.1 is captured in the ITU-T X.680 [17]. An introduction and an overview article can be found in [18, 19]. At its most basic, ASN.1 defines all data items by the **T**ype, **L**ength, **V**alue (TLV) concept. There are pre-defined atomic types and one may construct complex types and nested types. The explicit nature of ASN.1 data is a clear advantage. Irrespective of the notation and encoding scheme, one should in general have ways to uniquely determine *type* and *range* of the input data.

For commonly used data elements, such as email addresses, postal codes, date-time information (age/birth-date), etc., one may have dedicated verification code that recognizes the data types and valid input ranges, etc. Internationalization requirements may require that the code recognize national currencies, time-zones, specifics concerning delimiters, etc.

Other common data types, like names (people/places) also need proper handling. One should be specific about the permitted maximum data length, but it may be hard to impose strict rules for the general case. In fact, secure handling of free-form text handling is quite a complex subject [20, 21].

Regular expressions (regexp) methods are commonly used for analyzing complex patterns in the input data. Regexps were defined during the 1950ies by Stephen Kleene [22], and popularized by use in Unix text-processing utilities like *awk* and *sed*. Provided that the input data is sufficiently well specified, then regular expressions will be very well suited for verifying compliance. Regexp methods is particularly well suited for checking patterns in the data. Regexps are also well suited for generating invalid data for fuzz testing [23].

On the larger context of validation of input data strings, we note that there are a number of techniques and methods available. The reader may find an extensive treatment of the subject in the book "String Analysis for Software Verification and Security" [24].

4.2 Input Filtering and Input Sanitization

It is bad practice, but it is nevertheless quite common for web services to interface directly with SQL databases. This is often done for inquiries and look-ups, for instance to check inventory, etc. The problem is that this may very easily lead to SQL injection attacks (see an extensive treatment in [25]). In [26], the authors provide an overview over sanitizer and their efficiency.

Finally, one must always subject the output data of these efforts to input validation checking.

5 Discussion

In this paper we have outlined a input related software vulnerabilities. The vulnerabilities may be exploited by criminals, hacktivists or even nation-state actors that actively try to subvert the systems.

Designing and implementing secure software is a large topic, and we have chosen to only focus on the correctness of input data. That tiny part is quite important, and it is a foundational part of secure software. Specifically, we have investigated fuzz testing and input data validation. We have also seen that there are important asymmetries concerning the costs associated with fuzz testing and input validation. Here, we find that there are distinct differences concerning the costs of:

- Running fuzz tests and devising strong input validation methods
- Experiencing breach due to inadequate input data handling

The software developer will generally have to cover the costs of improving the security of the software. To exacerbate the problem, software developer does not have a strong incentive to do so as there are high upfront costs and limited benefits to be had.

The service provider will generally be responsible for the services it runs, and this includes covering costs associated with security breaches. This includes immediate costs, in the incident handling, but also post-fact costs associated with remediation and with fines, etc. On the other hand, the service provider generally wants to reduce its costs in procurement of the system. Thus the service provider, in effect, does not normally provide incentives for the software developer to improve the software.

6 Summary and Conclusion

6.1 Summary

Input validation is an activity that ensures better quality and improved security. There are indications that software that has been designed and implemented with sound input validation coding practices may be more trustworthy [27]. However, it is hard to design input validation functions that catches all cases. That is, input validation is necessary, but it will in practice inevitably be incomplete with respect to filtering out all invalid inputs.

Fuzz testing is similar to input validation in that it can ensure better quality and improved security. But, while input validation seeks to avoid accepting erroneous inputs in the first place, fuzz testing tries to trigger faults by deliberately injecting invalid inputs. And, similar to input validation, fuzz testing is in practice incomplete with respect to triggering all possible error cases.

It is a maxim in software development that errors and flaws are much more expensive to fix late in the development cycle that at an earlier stage. Flaws and errors that end up creating security vulnerabilities may turn out to be very expensive indeed. The incurred cost of such errors will reflect back to the software developers, but generally the main cost of security incidents will fall to the services that runs the exploitable software. For instance, security vulnerabilities that leads to a ransomware attack may incur large expenses to the organization being attacked (and not so much to the software developers).

Software that has been deployed will be "late", and it is therefore in general cost-effective to find as many flaws as possible prior to deployment. Thus, the importance of including input validation and performing fuzz testing can hardly be overstated.

6.2 Conclusion

Software included in public services and critical infrastructures must be as secure as possible. These services are subject to attacks and the cost of breaches may run high. Fuzz testing and input data validation are amongst the many methods that improve the security and general quality of the software services. These are important measures, and we argue that they should be implemented and conducted according to best current practices. To ensure that this happens, one will need to address the asymmetries concerning the cost of improving the software services. The software developers needs to be held accountable for the security performance of software and the service provider (or indeed any software consumer) must contribute to make this happen.

Funding Open access funding provided by University Of South-Eastern Norway. The authors have not disclosed any funding.

Data Availibility Statement There is no supporting data for this article.

Declarations

Conflict of interest The authors have not disclosed any competing interests.

References

1. Spafford, E. H. (1989). Crisis and aftermath. *Communications of the ACM, 32*(6), 678–687.
2. One, Aleph. (1996). Smashing the stack for fun and profit. *Phrack Magazine, 7*(49), 14–16.
3. Butt, M. A., Ajmal, Z., Khan, Z. I., Idrees, M., & Javed, Y. (2022). An in-depth survey of bypassing buffer overflow mitigation techniques. *Applied Sciences, 12*, 6702.
4. Kaur, J., & Garg, U. (2022). *State-of-the-art survey on web vulnerabilities, threat vectors, and countermeasures, 3–17*. Springer Singapore.
5. Di Zio, M., et al. (2016). Methodology for data validation 1.0. *Essnet Validat Foundation*.
6. Miller, B. P., Fredriksen, L., & So, B. (1990). An empirical study of the reliability of unix utilities. *Communications of the ACM, 33*(12), 32–44.
7. Godefroid, P., Levin, M. Y., & Molnar, D. A. (2008). *Automated whitebox fuzz testing, 8*, 151–166.
8. Naur, P., & (eds), B. R. (1968). Software Engineering; Report on a conference sponsored by the NATO SCIENCE COMMITTEE (07-11 October 1968. Report, NATO Scientific Committee, Garmisch, Germany.
9. van der Stock, A., Glas, B., Smithline, N., & Gigler, T. (2021). OWASP Top 10 - 2021. https://owasp.org/Top10/.
10. Saad, E., & Mitchell, R. (2020). OWASP Web Security Testing Guide; Version 4.2. OWASP Webpage.
11. Rose, S., Borchert, O., Mitchell, S., & Connelly, S. (2020). Zero trust architecture. Special Publication 800-207, NIST. https://csrc.nist.gov/publications/detail/sp/800-207/final.
12. Greenberg, A. (2018) . The untold story of notpetya, the most devastating cyberattack in history. *Wired* **22** .
13. Taleb, N. N. (2018). *Skin in the game: Hidden asymmetries in daily life*. Random House.
14. Chen, C., et al. (2018). A systematic review of fuzzing techniques. *Computers & Security, 75*, 118–137.
15. Wen, S., Meng, Q., Feng, C., & Tang, C. (2017). Protocol vulnerability detection based on network traffic analysis and binary reverse engineering. *PLOS ONE, 12*(10), 1–14.
16. Meyer, B. (1992). Applying, "design by contract." *Computer, 25*(10), 40–51.
17. ITU-T. (2021). X.680 : Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. Recommendation X.680, ITU-T. https://www.itu.int/rec/T-REC-X.680-202102-I.
18. ITU-T.(2022). Introduction to ASN.1. https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx.
19. Neufeld, G., & Vuong, S. (1992). An overview of asn 1. *Computer Networks and ISDN Systems, 23*(5), 393–415.
20. Unicode.org. (2014). UNICODE SECURITY CONSIDERATIONS. Unicode Technical Report 26, Unicode.org . https://unicode.org/reports/tr36/.
21. Unicode.org. (2021). UNICODE SECURITY MECHANISMS; v14. Unicode Technical Standard 39, Unicode.org. https://unicode.org/reports/tr39/.
22. Kleene, S. C., et al. (1956). Representation of events in nerve nets and finite automata. *Automat Studies, 34*, 3–41.
23. Shahbaz, M., McMinn, P., & Stevenson, M. (2015). Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions. *Science of Computer Programming, 97*, 405–425.
24. Bultan, T., Yu, F., Alkhalaf, M., & Aydin, A. (2017). *String Analysis for Software Verification and Security* (Vol. 10). Springer Nature.
25. Halfond, W. G., Viegas, J., Orso, A., et al. (2006). *A classification of sql-injection attacks and countermeasures*, Vol. 1, 13–15 IEEE.

26. Song, D., et al. (2019). *Sok: Sanitizing for security*, 1275–1295 IEEE.
27. Lemes, C. I., Naessens, V., & Vieira, M. (2019). *Trustworthiness assessment of web applications: Approach and experimental study using input validation coding practices*, 435–445 IEEE.

Geir M. Køien received his PhD from Aalborg University, on access secuirty for mobile systems. He has also worked for many years in industry, including LM Ericsson Norway and Telenor R &D. During these years he worked extensively with mobile systems and with security and privacy. He has also worked with the Norwegian Defence Research Establishment and with Norwegian Communications Authority on various security and communications related projects. Currently, he is a professor with the University of South-Eastern Norway (USN).

Lasse Øverlier PhD, Siv.ing. (MSc), Master Technology Management NTNU/Stanford Associate Professor, Department of Information Security and Communication Technology, Norwegian University of Science and Technology (NTNU), and Principal Scientist, Norwegian Defence Research Establishment (FFI) since 2002. He has a PhD Information Security from University of Oslo in 2007 and a Siv.ing. (MSc) degree from NTH/NTNU in 1993. Lasse has been working as a scientist, researcher and lecturer at NTNU and FFI since 2002 in the area of Information Security, and was a founder, co-founder, Technical Manager and IT Manager in the private sector prior to that. He has also worked as a Research Scientist for the US Army Research Laboratory in 2015-2016 and as a Research Scientist for the US Naval Research Laboratory in 2005-2006.