EasyChair Preprint
№ 691

Thou shalt not fail - Targeting Lifecycle-Long Robustness while being vigilant for the Black Swans

Simo Huopio

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

December 23, 2018

23rd ICCRTS "Multi-Domain C2"

Topic 6: Interoperability, Integration and Security

Thou shalt not fail – Targeting Lifecycle-Long Robustness while being vigilant for the Black Swans

Researcher, MSc, Simo Huopio
Finnish Defence Research Agency
P.O. Box 10, FI-11311 Riihimäki, Finland
simo.huopio@mil.fi

Abstract

Software products used in the critical infrastructure (CI) and command and control (C2) realms have very long lifecycles and have many interfaces that are crucial for secure interoperability and networked use. When exposed to the shorter lifecycles of the commercial off-the-shelf (COTS) software used within, new approaches are needed to keep these products secure.

Many common software components have shorter lifecycles than the CI/C2 products using them. An inherent security debt develops if vendors creating the systems do not keep up updating underlying components. Newer security testing methods might also find new security issues on old software that are not anymore under constant development.

Another source for security debt are changes in environment in which the system is operated in, and the assumptions of the typical usage of the product: Adding new network links, bringing in new data streams and new ways of using the system may seem simple and straightforward changes but may bring the security of the whole system under serious threat.

Exploitation of security debt in the software can become a black swan event, highly unexpected, and with severe consequences, if the end user is not aware of the risk. To address the security debt in critical long-lifecycle software, this paper suggests a sustainable long-term approach: Firstly, a highly automated robustness testing setup is proposed to constantly assess the most critical interfaces of the system. Secondly, a periodical threat analysis is applied to the product to detect the subtle changes in the usage and the environment.

Keywords: black swans, critical infrastructure, software robustness, software lifecycle, technical debt, security debt, threat analysis

1. Introduction

Extended lifecycle systems

Traditionally specific fields of industry plan for extended usage of their products: e.g. defense, aircraft, maritime, power plants and grids, and the Critical Infrastructure (CI) in general. The emergence of software in workstations used for overall control and user interfaces, or multitude of controllers tightly integrated to almost every part of the system, have brought whole new kind of complexity to the scene.

Quality or lack of security requirements will continue to be one of the most significant challenges on the security posture of all software [1]. Deficiencies in security requirements can steer the development work to take too much technical debt[1] in the form of suboptimal security architecture or by the lack of applying any measures of security development life cycle (SDLC) altogether. If the technical debt related to security is not appropriately addressed during the development time, the result is an end product with uncontrolled amount of security debt[2]. [2]-[4] Very long product lifecycles - from ten to tens of years - can have a multiplier effect on the security debt catalyzed by the non-optimal requirements. [2]

In this paper two of these emphasized challenges are identified: Firstly the security effects of having limited lifecycle components – COTS or self-built – within complex long lifecycle products is discussed. Secondly, the effects of the changes in environment, usage, and functionality in the light of security posture of the product are debated.

According to Taleb in [5], a black swan event is “highly unexpected by the observer, carries large consequences, and is subjected to ex-post rationalization”. A large security incident caused by unmanaged security debt fits perfectly to the definition: The results can be catastrophic to the user who thinks everything was under control, and the particular event could have been avoided by fixing a single bug or configuration error.

In addition to the practical discussion of the very useful security debt metaphor, this papers main contributions include threat analysis approach adapted to the end user of CI system, and a long term robustness testing concept created to help reducing security debt in extended lifecycle products.

---

[1] Technical debt metaphor is defined by Ernst et al as “a design or construction approach that is expedient in the short term but that creates technical context in which the same work will cost more to do later than it would cost to do now, including increased cost over time” [10]

[2] Geer and Wysopal define the security debt as the measure of security flaws left unfixed or unpatched existing in the code that can be exploited maliciously. [2] A whitepaper by Whitehouse et al[4] provides a thorough analysis of the impact of security debt to the software development.

To illustrate the impacts of these challenges and the proposed solutions to them an abstract Long Lifecycle Product 1 (LLP1) in the defense industry is defined as follows:

- LLP1 has multiple physical subsystems with different roles. Each subsystem has at least one control target, which implements the desired end functionality of the system. The whole product is operated using at least one Control workstation (WS).
- LLP1 uses a mainstream embedded operating system (OS) on the subsystems, with custom software (SW) included on top of the OS
- LLP1 uses an mainstream workstation (WS) OS on control stations, with custom software included on top of the OS
- LLP1 uses IP communications on internal network
- LLP1 is used initially in a stand-alone fashion and is not supposed to communicate anything outside the network

LLP1 is shown as a block diagram in Figure 1.

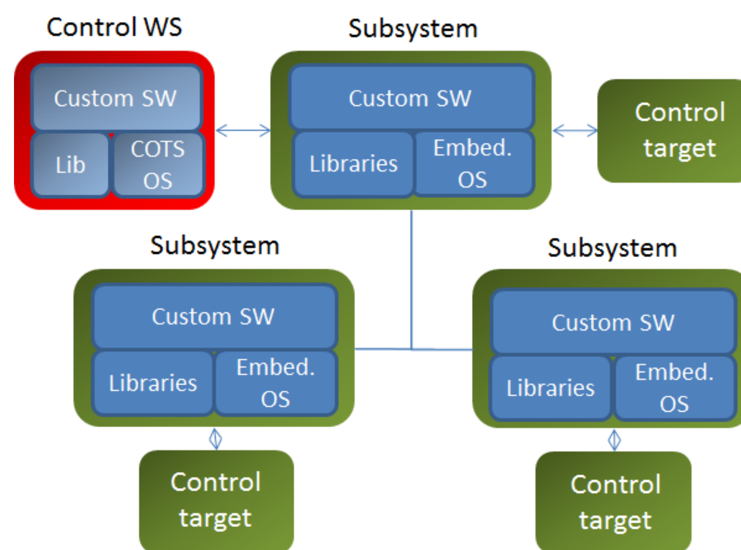

Figure 1: Long Lifecycle Product 1 (LLP1)

Short lifecycle components in complex long lifecycle products

On the creation of buildings, mechanics, and electromechanics of the extended lifecycle infrastructure, the longevity is traditionally assumed from the beginning of the project. The risk of obsolescence – the part becoming un-procurable from manufacturer [6] - is actively managed with long support contracts, and proper documentation. However, when the system involves electronics, embedded controllers, computers and especially software, the extended lifecycle has to be specifically addressed to ensure the component compatibility, bug fixes, and in general the maintainability of the system. According to Muñoz [7] software obsolescence can take place in three main areas: Skills, Media, and Software COTS effects. Skills area refers to skills, knowledge and information required creating, supporting, or modifying the software. For very long lifecycle products like aircraft, also the data storage formats and media are an issue if not properly managed and maintained. The most significant risk that could lead to obsolescence is the composition of the system software itself, including self-created, commercial-off-the-self (COTS) and open source (OSS) parts of software.

The COTS or OSS parts, subsystems, and software libraries are of primary concern, as they typically have their own shorter lifecycles that the CI system they are used in. When such component reaches the end of its lifecycle and it is not maintained anymore, its replacement might require different toolchain, and have different – even conflicting – dependencies, or might lack the support of the needed hardware, making it very hard, and costly to integrate[7], [8]. In practice, the developers may make a conscious decision to leave the component as it is, and counting on that new security bugs would not emerge to the old software. This act of taking the conscious risk decision is called taking technical debt [9], [10], or in case of security posture of the system, security debt [2]. In practice it is a widespread practice to leave some software modules behind without any direct maintenance. Sometimes there are already many old libraries present in the software on the moment it is released, as Eronen has found in his paper "Patched but still vulnerable - code rot in popular applications" [11].

In-house built software without significant COTS subsystems can also accumulate security debt in a similar fashion. The reason to this is usually architectural as development of new features might lead to substantial unwanted refactoring of existing codebase. In this case the manufacturer might decide to take in technical debt to catch the planned time to release or in general to save in development costs. These kinds of risk decisions accumulate and if not mitigated can add up to significant risk during the extended lifecycle of CI system.

For many CI systems unmanaged software obsolescence and resulting security debt is an unsolved challenge; The CI projects are much better equipped to manage and minimize the possibilities of obsolescence in hardware and electronics than in software[7].

Figure 2 illustrates the situation of LLP1 after some time has passed from the deployment. There might be already outdated software within the subsystems, or there might be a long time since the custom software parts of the system have been tested with the newest robustness[3] testing tools.

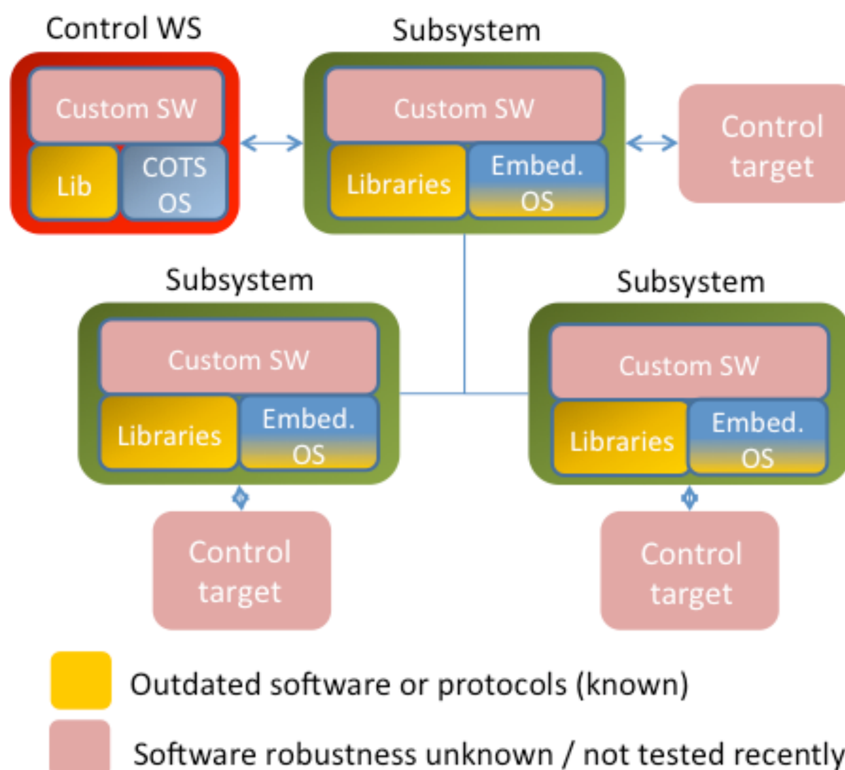

Figure 2: Situation on LLP1 after some degradation has happened

[3] IEEE has defined software robustness as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. [12]

To find ways to manage the effects of having software components with short lifecycles in complex long lifecycle products the following research question has to be answered:

RQ1: In which ways the end user of long lifecycle critical infrastructure (CI) system can address the security debt caused by software components with shorter lifecycles?

Changes in usage and environment

The changes in the operating environment and the usage are the central challenges of extended lifecycle systems. Examples of these could be added network connectivity and functionality, changes in physical operating area, changes in operator training and background, or changes in security landscape via emergence of new attack tools and malware, and crypto algorithm weaknesses.

Individually these changes can be small, but when accumulated over the time they can broaden the vulnerable attack surface of the system, undermine the security principles of the original systems, or even lay groundwork for a black swan -like event, which has not been anticipated at all by any of the involved.
To illustrate the types of changes that can occur, we can use our LLP1 -system as an example. Let's consider following changes to the system in a Middle Life Update 1 (MLU1):

- A network link has been introduced which connects the system with an external computer server for providing metrics and status information to the Network Operations Centre (NOC).
- A new wireless link technology has been introduced which allows more flexible configuration of the system in the field
- The physical housing of the product changes so that the control workstations can be co-located with other control equipment used in the field, which means that the control workstations are not physically as protected as before.

Figure 3: is an illustration of the LLP1 with the MLU1 changes

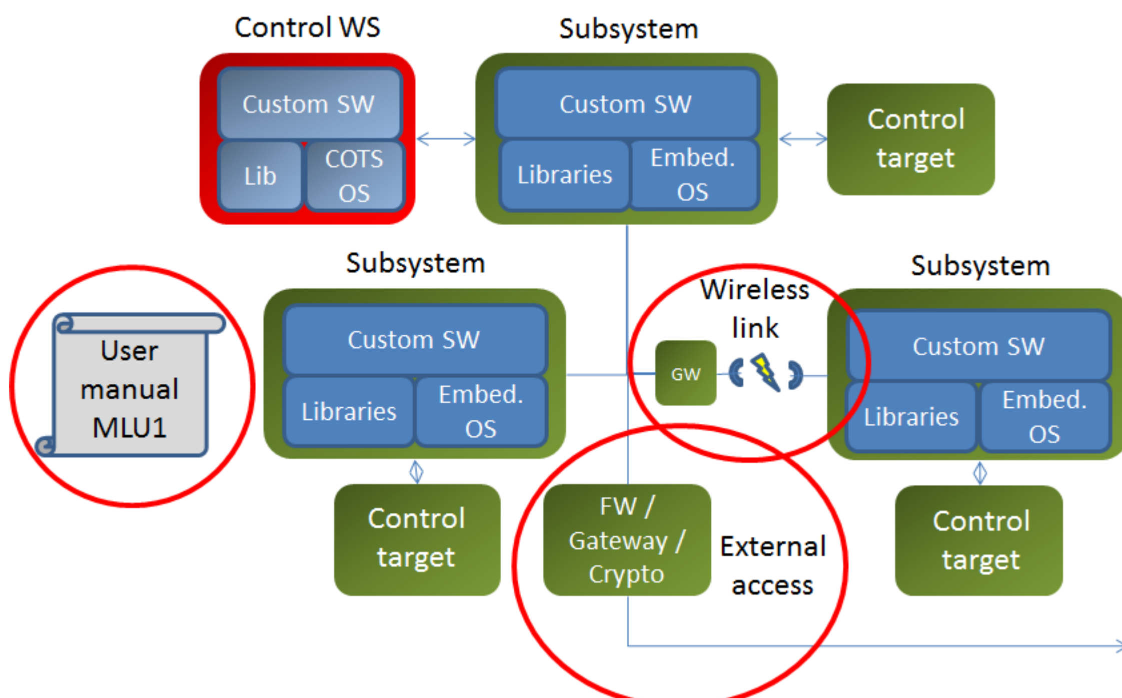

Figure 3: New configuration of the LLP1 system

To find concrete concepts for the CI system maintainer to address the effects of changes in the usage patterns and the systems working environment the following research question has to be answered:

RQ2: What are the best ways to manage the effects of significant changes in the CI system, its working environment or its usage?

2. Related work and research methods

2.1 Research questions and hypothesis

The first chapter introduced the challenges and the related research questions:

RQ1: In which ways the end user of long lifecycle critical infrastructure (CI) system can address the security debt caused by software components with shorter lifecycles?

RQ2: What are the best ways to manage the effects of significant changes in the CI system, its working environment or its usage?

With the present knowledge of the field, a following hypotheses was drafted:

The maintenance of a complex Critical Infrastructure (CI) system has to take tools and methodologies from the software research and development in to use to address security debt and manage the changes of system environment during long lifecycle of the product.

The chosen methodology in this paper was to do an iterative literature review to find the existing work done within the area and to find whether the given hypothesis can be argued.

2.2 Methodology

Based on the research questions and hypothesis the literature search was concentrated on the following areas: Technical debt, Security debt, Software obsolescence and COTS in long lifecycle products, Threat analysis and emulation, Software robustness testing, Robustness testing automation, antifragility and Black Swan events in context of software products. The source databases used were IEEE Xplorer, ScienceDirect and Google scholar. The terms were cross-referenced with the IEEE Standard Glossary of Software Engineering Terminology when possible.[12]

As a result total of 74 articles were chosen by relevance. These include 12 Journal articles, four surveys, 36 conference articles, five magazine articles, eight books, four thesis and five web documents.

2.3 Results of Literature Review

The technical debt was a lot researched issue in extended lifecycle systems, as were different types of obsolescence. Software obsolescence, as opposed with mechanical part- and electronics obsolescence, has been widely ignored in practice. Software obsolescence was connected in many cases to the use of COTS software. The relationship between technology debt, security vulnerabilities and defect is analyzed thoroughly by Nord et al. in [13]. Security debt is brought as a new term as a subset of technical debt by Geer et al. [2], [3]. A whitepaper by Whitehouse et al. [4] provides a thorough analysis of the impact of security debt to the software development. As the security breaches and incidents have caused substantial impacts during the last decade, the security debt has much more quantifiable value than the technical debt, or product quality as an abstract term. [2], [3], [6]-[10], [13]-[22]

Most current threat modeling publications adopt at least partially the approach introduced in Microsoft Secure Development Lifecycle (SDL). In the SDL, STRIDE threat model is used to discover the threats using the following categories:

- Spoofing
- Tampering
- Repudiation
- Information Disclosure
- Denial of service
- Elevation of privilege

Further in the process, a DREAD model is used to further score the identified threats rating them in the following areas:

- Damage potential
- Reproducibility
- Exploitability
- Affected users
- Discoverability

Later on, Microsoft has tried to make the process more approachable for wider audience with the help of online, and even physical facilitation tools [23], [24]. Excellent walkthroughs of these methods are available from Microsoft and Open Web Application Security Project (OWASP) [23], [25], and they are discussed in approachable manner by Steven in [26]. A summary of further threat analysis methods is done by Hussain et al. in [27]. While threat modeling is considered to be an essential tool in product creation, it is rarely applied in customer perspective. It is to be noted that a large number of publications about threat analysis and modeling are case studies of applying the technique to individual product or a product class, and were excluded from the references. [21], [23], [25]-[33]

Software robustness testing publications are saturated with descriptions and further evolutionary modifications of distinct fuzzing[4] tools. The recent years have shown the effects of significant investment to the field: Now the approach is more scientific, concentrating to maximizing the ability to find execution paths through the application, as well as to maximize the code coverage for the

[4] Fuzz testing or fuzzing is a software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion

actual fuzz testing activity. There is also push for common methods for evaluating the fuzzers. [1], [34]-[54]

As the quality of the software still varies a lot, there are two distinct classes of applying the robustness testing tools: Low hanging fruits, like applying fuzz testing or other new methodology for the first time – this often yields swift and concrete results. The other distinct end of the spectrum is the push to maximize the tool performance in finding the rare but significant bugs in already very robust software with large fleets of fuzzing instances, innovative ways to instrument the target code and millions of executions. This end also drives for higher automation over interactive investigative approach, and has resulted also deeper integration of the tools to the development workflow, be it Continuous Integration (CI), DevOps[5] [55], or agile – or combination of all of these. [1], [11], [32], [43], [48], [55]-[65]

The Taleb's black swan theory applies well to the realm of complex critical software products: Software bugs are always present, they are hard to predict, and they can lead to high profile events beyond the realm of reasonable expectations of the product authors or the end users. These kind of black swans manifest in failures of large software projects, and also in the high profile security bugs, and massive data leaks. There are many approaches to tackle the black swan –like events: Increasing the robustness and ruggedness by changing the architecture, product creation methods and testing strategies. Taleb's own concept of antifragility takes the concept even further: Antifragile systems are designed to assume failures and external shocks, and to get stronger by them. [15], [66]-[73]

[5] DevOps as defined in [55] by Dyck et al "is an organizational approach that stresses empathy and cross-functional collaboration within and between teams in software development organizations, in order to operate resilient systems and accelerate delivery of changes."

3. Results

3.1 Overview

The literature review results support the research hypothesis: Tools typically used in R&D phase - like threat analysis and software robustness testing - can also be applied in the product end-user perspective.

The technical debt is typically known, and at least partially managed within a R&D process. Usually it does not manifest itself to the end user if managed properly. However, the effects of security debt are shared more evenly between the vendor and the end user. In fact, when unmanaged and unknown, large amounts of security debt can easily result in catastrophic, black swan –type events to the end user.

Especially in the realm of critical infrastructure, where there is a tradition to take good care of hardware obsolescence, these tools can help to create similar processes for the increasing number of software components that are in risk of obsolescence be it the lack of updates, compatibility, poor quality of code, or other sources of technical or security debt.

In this chapter, threat analysis is described based on the literature review in subsection 3.2. As a new contribution, threat analysis framework is introduced, modified to suit the end user perspective in (subsection) 3.3. Further on, software robustness testing in R&D is described in (subsection 3.4). Building on that knowledge, a lifelong robustness testing concept is laid out, which is meant as a tool for the end user to take active role in managing the hidden, yet known vulnerabilities of the critical software.

The identified challenges are addressed with the combination of integrating periodical threat analysis process to the CI system ownership, and by creating the ability to conduct sustained, highly automated robustness testing to the most critical interfaces of the system thereby fortifying these interfaces even further. The end user seldom has complete visibility to the internal complexities of the system, but with these tools it can have much more level and active relationship with the system manufacturer.

3.2 Threat analysis

Threat analysis in R&D

OWASP guide to Application Thread Modeling [25] introduces the subject as follows

> *"Threat modeling is an approach for analyzing the security of an software application. It is a structured approach that enables one to identify, quantify, and address the security risks associated with an application. Threat modeling is not an approach to reviewing code, but it does complement the security code review process"*

The process of threat modeling is described as follows in [25]:

1. **Decompose the system:** Gain an understanding of the system and how it interacts with external entities. It is considered a good practice to create data flow diagrams (DFDs) that show different paths through the system, highlighting the privilege boundaries.

2. **Determine and rank threats:** Using the DFD graphs as a starting point, identify the threats using a threat categorization methodology such as STRIDE, or Application Security Frame (ASF). The threats are ranked using a value-based risk model such as DREAD, or less subjective qualitative risk model based on general risk factors.
3. **Determine countermeasures and mitigation**: In this phase countermeasures and mitigations are identified to each of the risks. In this phase the threats and risks can be prioritized in a way that is most relevant to the case (e.g. business risk).

When implemented during the R&D phase in the way suggested by [25], [27] the threat modeling produces information that help the overall management and the communication of the security posture of the product. The arguments raised against applying the process are often related to the expenses, especially the human resource usage. The early versions of Microsoft proposal have been reported to require security professionals with specific expertise to succeed [26], [31]. In addition to find the right combination of people doing the analysis, the failure to scope the work properly can have ambiguous and inconclusive results.

There are many adaptations of the process, such as integrating it to agile product creation [32], using different tools like card games [24], abuser stories, as well as graphical and quantitative tools.[27]

3.3 Threat analysis framework for CI end product

The premise for threat analysis from the customer perspective

When applying the threat emulation process from the software product customer perspective there are apparent limitations when compared to what the product creator can do in the R&D phase:

- The lack of knowledge from the internal architecture of a product: Typically the end user is aware of the external interfaces of the system, the ways to configure the system, and the external dependencies and main modules of which the product is constructed (at least as far as the configuration is concerned). The detailed architectural choices may be hidden from the view.
- The end users deal with the finished product: The chosen security architecture with the controls, countermeasures and mitigations cannot usually be changed.
- The end user has limited set of mitigations and countermeasures. In practice the end user is limited to tools external to the target system, the system configuration and how the operators of the product are trained.

It would seem that the actions the end user can take would be insufficient. However, the threat emulation can make a big difference to the security posture of the overall system, and it can help the maintainer to make much more informed choices during the lifecycle of the product:

- Even when treating the system as a black box, the threat emulation work can give essential information about the product: The focus of the process can be on the unique usage environment where the system is used in, and on evaluating the assumptions of the known security controls.
- Threat analysis process can help clarifying the assumptions of the privilege and trust boundaries of different parts of the system, and of the different user roles that interact with the system. This knowledge helps identifying the most critical interfaces of the system, robustness of which can be challenged by own robustness testing processes.

- The analysis made from the customer perspective gives the owner and maintainer excellent material for the dialog with the product creator and vendor. It also helps maintaining the security posture in the requirements for middle life updates planned for the product
- The information helps considerably handling the potential security incidents as in many cases the impact of breach in different parts of system is already known.

Suggestion: The periodical threat analysis

The suggestion of threat analysis work for the extended lifecycle CI system can be laid out in two phases: More extensive *Initial threat analysis* (phase 1) which is executed during and right after the procurement, and iterative *threat analysis update (phase 2)* which is performed preferably periodically, or at least when system is updated with new functionality or with new usage patterns.

The initial threat analysis (phase 1) is conducted in the process of the system procurement and integration, preferably before the system is taken into use as the results of it can be used to add threat mitigations to the usage training, and the production configuration of the system.

To conduct the initial threat analysis it is needed to gather together knowledgeable personnel from the following categories: End-user/instructor, Product owner, product maintenance representative, manufacturer representative, and possibly a facilitator who will help to ensure the comprehensiveness of the process. It is advisable to cycle new people to get new ideas. Enough time and peaceful setting should be given for the team to help the process. During the workshop, the following steps should be followed:

Step 1: The primary system requirements are captured in the form of use cases for the system. The use cases should cover all the primary functionality and the operators.

Step 2: The threat modeling process is executed as described in [25] but by treating the target system as a black box with external interfaces with the environment, neighbor systems and the operators interacting with the system. The use cases of the system are used as help to determine the impacts of the threats. If known at the time of the analysis, the top-level threat models of the other systems communicating with the current target can be used as a reference.

a. For the mitigations, concentrate on the operator instructions, the end user configuration, and robustness testing.
b. The risk list is compiled so that it can be shared and further discussed with the system manufacturer. Even though some risks would be escalated to the manufacturer, they should always be mitigated or accepted by the product owner.

Step 3: A specific list of the most critical interfaces is compiled for guiding the robustness testing efforts.

Step 4: The results of the initial threat modeling are augmented with the Bill of Materials (BOM) analysis of all the system software. The result of this work should be the list of internal library components and their versions with the current vulnerability status. This data is also composed to be shared with the manufacturer and the robustness testing actors

The threat analysis update (phase 2) is more about the review of the security posture, and to comprehend the impact of the changes that apply to the overall system since the last iteration. The composition of the team doing the analysis should be similar as in the initial phase.

Step 1: If any parts of software have changed, redo the BOM analysis (phase 1 step 4). Augment the results with the latest vulnerability feeds.

Step 2: Go through and acknowledge the evident changes in the system, system dependencies, usage and its usage environment. Specifically look for

a. Software obsolescence and code rot
b. Added architectural complexity

Step 3: Go through the old identified threats and state the validity in the perspective of the changes in first two steps.

Step 4: Brainstorm the possibility and content of new threats.

Step 5: Go through the identified threats and quantify them in an agreed fashion

Step 6: Go through the results of the long-term robustness testing work that has been done since the last iteration of threat analysis.

a. Augment the list of threats with the results
b. Scope the long-term robustness testing work for the next period.

Step 7: Make decisions in the light of the resulting list of threats.

In addition to the quantified and accepted list of threats to the system, and the risks or security debt they pose, the end result of the threat analysis work can be

- Changes in documentation, training and the constraints set for the use of the system
- Changes in requirements toward next version or more substantial middle life update (MLU)
- Shortlist of interfaces which should be under constant robustness testing
- Decisions about end of operational life of the system with suggestions of procedures of replacing it

In our LLP1 example system one result of the threat analysis work performed after the MLU1 changes identified some interfaces to be critical in the sense that security vulnerability on their implementation could directly expose the internal functionality of the system to possible attacks from external sources. Therefore they are good candidates for long-term robustness testing effort. The chosen interfaces are illustrated in Figure 4:

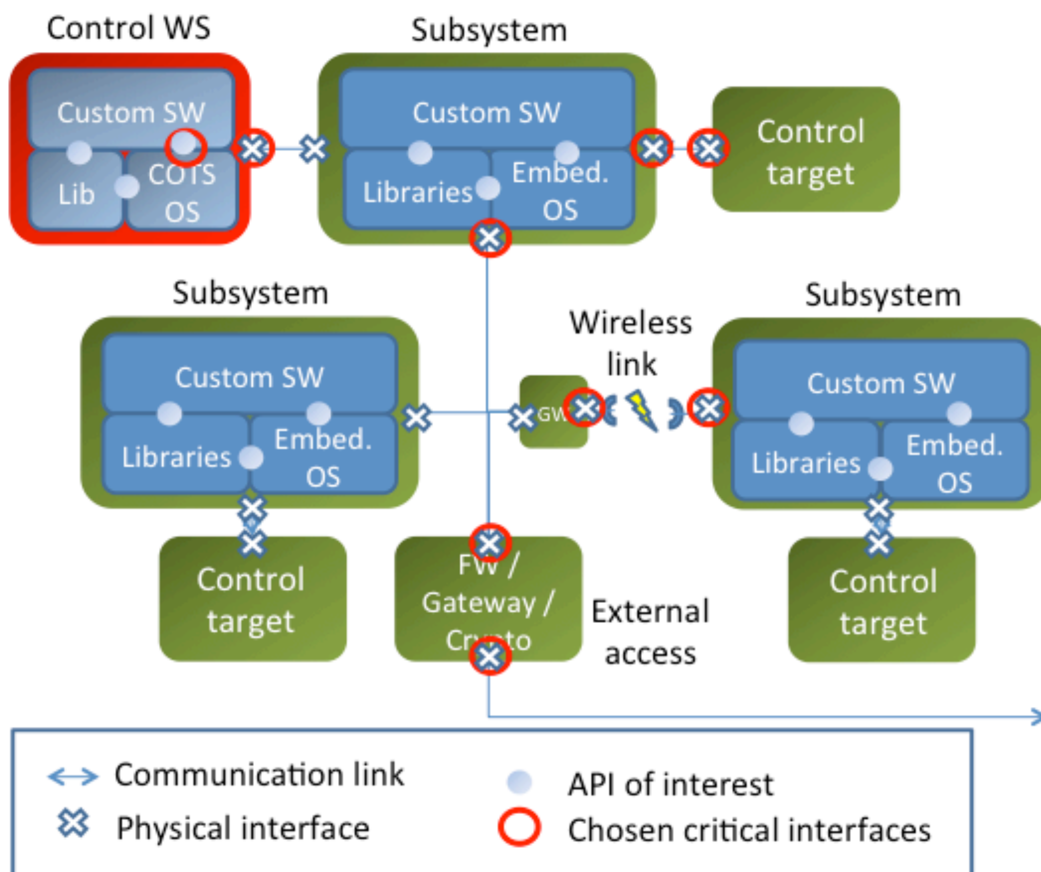

Figure 4: Critical interfaces identified in threat analysis work after MLU1

3.3 Lifecycle-long robustness testing

Software Robustness testing

Software robustness is defined in [50] as follows:

> *"Robustness testing. Informally, robustness can be defined as the ability of a software to keep an "acceptable" behavior, expressed in terms of robustness requirements, in spite of exceptional or unforeseen execution conditions (such as the unavailability of system resources, communication failures and invalid or stressful inputs. Such a feature is particularly important for software critical applications those execution environment cannot be fully foreseen at development time."*

The term "robustness testing" was first used by the Ballista project at Carnegie Mellon University [57] Most of the modern fuzzers (robustness testing tools) use the same basic principle than in Ballista for finding actual errors: supplying the target software interface with input that is intentionally broken. This approach was developed further by PROTOS project and its successors experimenting with different ways to infer, model, and break the inputs and with significant results. [33], [45], [46], [49], [63], [74]. Later on, the effectiveness of the fuzzing tools has been significantly improved with the emphasis on the execution speed in test automation and the target of maximizing the code coverage reached with the fuzzing tool.

An example of "bleeding edge" of the fuzzing tools can be considered to be American Fuzzy Lop (AFL) tool, and libfuzzer and their variants [42], [54], [59]. The critical success factors of the tools have been

the ease of use and integrated automation of code coverage maximization through target code instrumentation or emulation. These tools have been used to find new high profile security vulnerabilities, like Heartbleed [62], even in the software that has been considered very robust, and been exposed to very hostile treatment for decades.

The modern way to use fuzzing tools is to integrate them into the product creation environment. In this way all the external interfaces in the software have undergone at least some elementary robustness testing. It is speculated that in the future the testing work can be automated to the degree that the computer systems could test and fix their own code. For practical applications this is still out of reach, or applied only in simplified laboratory environment. Nevertheless, the events like Defense Advanced Research Projects Agency (DARPA) Cyber Grand Challenge [75] have been showcasing this technology and its progress.

3.4 Automating the robustness testing for the CI systems

The suggested approach in this paper is to concentrate the customer robustness testing efforts to the most critical interfaces that should be in the very robust state in the first phase. The new software faults found in the implementation of those interfaces could be considered as very improbable but with the very high impact – typical black swan events.

Following the hypothesis, one should be able to apply robustness testing to the critical interfaces of CI system for extended periods of time. As noted before the automation and the effectiveness of the testing tools are increasing. Nevertheless, there are still specific challenges to overcome:

The test automation optimization for very long test runs: To achieve very long robustness testing runs the test setup has to be very robust by itself, while still being scalable and flexible. This means keeping memory and other resource consumption in control, and being able to autonomously and seamlessly save the state of the fuzzing effort to continue with fresh environment should the resource anomalies arise. The performance drawbacks from this approach could be compensated with scaling the number of parallel test runs and the sustainable extended execution of the test.

Minimizing the false positives and ambiguous results: The traditional robustness testing is interactive work, and longer test runs on unstable interfaces may produce lots of noise: High number of false positives and weird but harmless behavior that is hard to replicate. Should the system be prone to generate a lot of false positives significant efforts has to be made to the testing automation and the automated processing of the raw results.

Ways to minimize these kinds of findings include

- Strict monitoring of the target system resource usage
- Separate training period monitoring the normal usage of the product
- More in-depth instrumentation of the code which could help detecting preconditions for complex errors
- More profound isolation of different test cases.

Should the amount of ambiguous cases stay high, or should the needed isolation slow down the testing too much to be feasible, might argue for using more interactive, intensive and shorter test strategy instead of lifecycle-long mostly automated testing.

Choosing the right metrics for the extended test runs: The metrics gathered from the testing have to contribute to the long-term goals in addition to just state the obvious findings: Depending on the target system one can accumulate the statistics based on code coverage, input space coverage, secondary indicators (resource usage, responsiveness, generic error handling). Eventually, the metrics collected from the testing should validate that the continuation of the activity has theoretical possibilities to find new errors.

The triage and practical follow-up of the results for old parts of the software: Many components of the extended lifecycle CI system software can be considered to be in the maintenance mode, and the manufacturers support on technical issues can be slow or nonexistent. If the response times to triage new findings from an old COTS library grow considerably, one has to make decisions when to call the library obsolete and start refactoring even if the issues found in testing would not be conclusive.

In the context of our example system, the Figure 5 illustrates the interfaces that were chosen for long-term robustness testing using the threat analysis results of LLP1 MLU1: The external network interface of the Gateway, the external network interface of control workstation, and the interface facing the internal network of the Subsystem.

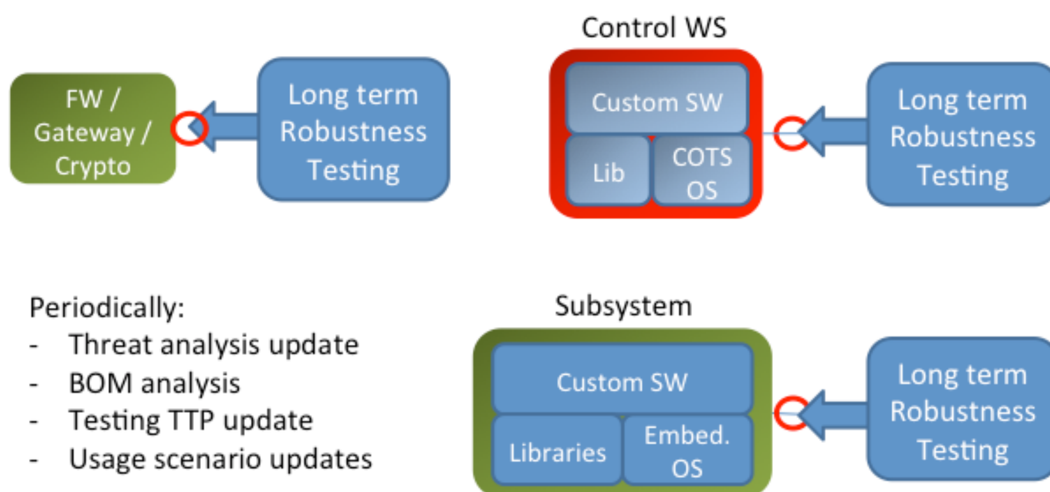

Figure 5: The chosen long-term Robustness testing setups for the critical interfaces

Each of the chosen interfaces would be tested in isolated test setup. The control workstation would be tested on virtual machine (VM), the target Subsystem and Gateway might need a particular setup including the instrumentation of physical device.

Independent on the actual target, the robustness test setup would be built up incrementally in following way:

I. First the robustness of the chosen interface would be validated with interactive robustness testing with traditional fuzz-testing tools, which are able to generate network traffic. In this setup, the stability of the test tooling itself and the target would be confirmed.
II. During the first sessions, also the resource usage is monitored for the more extended test runs. Ability to save the state/seed of the testing and to be able to continue from the same situation is verified, as the target system might have to be periodically reset if the fuzzing slowly consumes resources.
III. The leap to really long test periods requires choosing the metrics, which would confirm the slow progress of the testing effort through the input and target state space. The metrics to consider: Target code coverage and fuzzing passes executed per known path, amount of input space covered.

The steps are illustrated in Figure 6

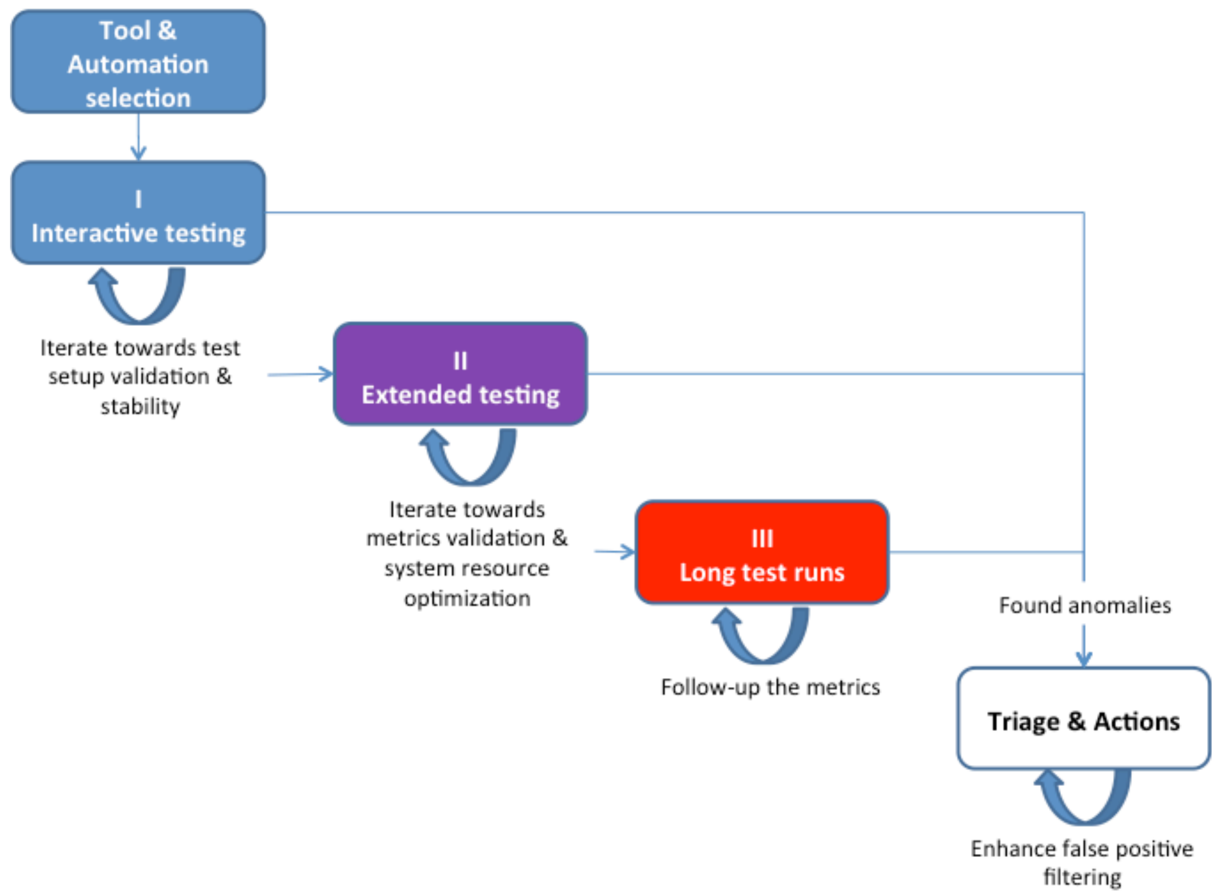

Figure 6: Incremental process of building robustness testing setup for a chosen interface

4. Discussion

4.1 Targeting lifecycle-long robustness

To aim for realistic security posture for long-lifecycle system it is needed that the security is taken seriously into the account already on the product creation phase with the manufacturer. In practice this means that the manufacturer uses some form of secure development lifecycle (SDLC) in its R&D efforts. Even though this phase would be in order, the end user hast to take active role to target lifecycle-long robustness of the system.

In the role suggested in this paper, the usage and maintenance phases of the lifecycle have to be augmented with periodical threat analysis work, critical and proactive approach with the software updates of different parts of the system and extended robustness testing of the most critical interfaces. The overall process is illustrated in figure 7.

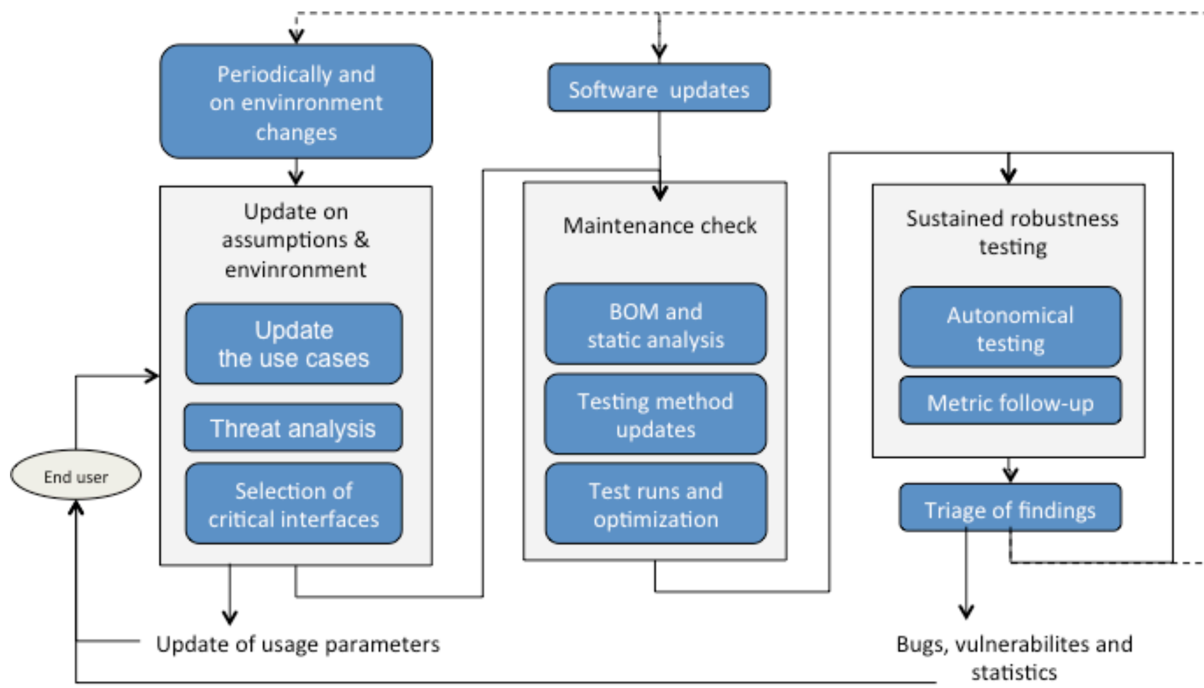

Figure 7: Lifecycle-Long Robustness testing process

Important factor is the general attitude of being vigilant, and look at the system with the fresh mind after each cycle. This means the manufacturer and the end user should not be caught unaware of the impact of environmental or technological changes to the protected system. Therefore at least some of the lurking black swans could be turned gray, with the time to prepare for the impact.

4.2 Contribution

The main contributions of this paper are as follows:

1) Threat analysis approach adapted from the industry best practices to the end user of Critical Infrastructure system. The data from previous cycles of threat analysis is used during the following periodical iterations to detect and weigh the subtle changes in the system use.

2) A long-term robustness testing concept created to help reducing security debt in extended lifecycle products. The most essential interfaces of a critical system should be on constant scrutiny, using the newest tools possible.

3) Practical discussion of little known but very useful security debt metaphor based on up-to-date literature review. The metaphor is essential to internalize when managing and supporting extended lifecycle products.

4.3 Impacts

When implemented, the process can have following impacts to the CI system maintenance:

- The security posture of the system is better known to the owner and maintainer of the product. Especially the technical and security debts are managed in much more informed fashion.
- The dialogue with the manufacturer regarding the security issues of the product can take place at deeper level. Both manufacturer and the owner know that the reciprocal side is taking the security seriously.
- The end of the system lifecycle is managed in much better and informed way as the update and end-of-life decisions can be made in controlled fashion based on facts.
- More resources are needed for the CI system maintenance. In addition to the personnel costs, the approach adds technical resources necessary for the sustained testing activity.

4.4 Conclusions

The long lifecycle critical infrastructure systems have many challenges. This paper has suggested a solution to two of them: Incompatible life cycles of COTS components and the whole CI system, and the hidden effects of changes of usage and the usage environment of the system. The solutions proposed in this paper are implementing a process of threat analysis that is done in the end users perspective. The analysis would be augmented with the extra insight given the BOM analysis of the software. In addition to that, the selected critical interfaces would be subject to sustained robustness testing effort.

Acknowledgments

I'd like to thank my employer Finnish Defence Research Agency (FDRA), the colleagues who have provided comments and advice, and professor Juha Röning of University of Oulu for coaching my research to this point.

References

[1] M. M. Hassan *et al*, "Testability and software robustness: A systematic literature review," in 2015, Available: https://ieeexplore.ieee.org/document/7302472. DOI: 10.1109/SEAA.2015.47.

[2] D. Geer and C. Wysopal, "For Good Measure - Security Debt," *;Login:,* vol. 38 no. 4, pp. 62-64, Mar. 2013.

[3] D. Geer and D. Conway, "Foor Good Measure - The Price of Anything Is the Foregone Alternative," *;Login:,* vol. 38 no. 3, pp. 58-60, Jun. 2013.

[4] Whitehouse Ollie and Vaughan James, "Software Security Austerity - Software security debt in modern software development," *Rexc Whitepaper,* vol. 1, 2012.

[5] N. N. Taleb, *The Black Swan: The Impact of the Highly Improbable.* (2nd ed.) NY: Random House, 2010.

[6] B. Bartels *et al*, *Strategies to the Prediction, Mitigation and Management of Product Obsolescence.* 2012*87*.

[7] R. G. Muñoz *et al*, "Key Challenges in Software Application Complexity and Obsolescence Management within Aerospace Industry," *Procedia CIRP,* vol. 37, pp. 24-29, 2015.

[8] S. Rajagopal, J. A. Erkoyuncu and R. Roy, "Software obsolescence in defence," *Procedia CIRP,* vol. 22, pp. 76-80, 2014.

[9] B. Curtis, J. Sappidi and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt,* 2012, pp. 49-53.

[10] N. A. Ernst *et al*, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering,* 2015, pp. 50-60.

[11] J. Eronen and M. Laakso, "Patched but still vulnerable - code rot in popular applications " in *Cybersecurity Symposium, Idaho, USA. * 2016, .

[12] J. Radatz, A. Geraci and F. Katki, "IEEE standard glossary of software engineering terminology," *IEEE Std,* vol. 610121990, *(121990),* pp. 3, 1990.

[13] R. L. Nord, "Software vulnerabilities, defects, and design flaws: A technical debt perspective," in *Fourteenth Annual Acquisition Research Symposium,* 2017, pp. 67-75.

[14] F. J. Rojo Romero *et al*, "Key challenges in managing software obsolescence for industrial product-service systems (IPS2)," in *2nd CIRP IPS2 Conference,* 2010, pp. 393-398.

[15] E. Jang *et al*, "Unplanned obsolescence," in Jun 22, 2017, pp. 93-101.

[16] P. A. Sandborn, "Editorial Software Obsolescence-Complicating the Part and Technology Obsolescence Management Problem," *Tcapt,* vol. 30, *(4),* pp. 886-888, 2007. Available: https://ieeexplore.ieee.org/document/4383342. DOI: 10.1109/TCAPT.2007.910918.

[17] T. Besker, A. Martini and J. Bosch, "Impact of architectural technical debt on daily software development work - A survey of software practitioners," in *43rd Euromicro Conference on Software Engineering and Advanced Applications,* 2017, pp. 278-287.

[18] G. Digkas *et al*, "How do developers fix issues and pay back technical debt in the apache ecosystem?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER),* 2018, pp. 153-163.

[19] H. Ghanbari *et al*, "Looking for peace of mind? manage your (technical) debt: An exploratory field study," in *ESEM 2017: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ISBN 978-1-5090-4039-1,* 2017, .

[20] Z. S. Hossein Abad *et al*, "Understanding the impact of technical debt in coding and testing: An exploratory case study," in *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice,* 2016, pp. 25-31.

[21] D. Landoll, *The Security Risk Assessment Handbook: A Complete Guide for Performing Security Risk Assessments.* (2nd ed.) CRC Press, Inc., 2011.

[22] Z. Li, P. Avgeriou and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Software,* vol. 101, pp. 193-220, 2015.

[23] B. Potter, "Microsoft SDL threat modelling tool," *Network Security,* vol. 2009, *(1),* pp. 15-18, 2009.

[24] A. Shostack, "Elevation of privilege: Drawing developers into threat modeling." in *3gse,* 2014, .

[25] Anonymous "Application Threat Modeling," *Https://Www.Owasp.Org,* May 31, 2017.

[26] J. Steven, "Threat modeling-perhaps it's time," *IEEE Security & Privacy,* vol. 8, *(3),* pp. 83-86, 2010.

[27] S. Hussain *et al*, "Threat modelling methodologies: a survey," *Sci.Int.(Lahore),* vol. 26, *(4),* pp. 1607-1609, 2014.

[28] D. Dhillon, "Developer-driven threat modeling: Lessons learned in the trenches," *IEEE Security & Privacy,* vol. 9, *(4),* pp. 41-47, 2011.

[29] J. A. Ingalsbe *et al*, "Threat modeling: diving into the deep end," *IEEE Software,* vol. 25, *(1),* 2008.

[30] R. Khan *et al*, "STRIDE-based threat modeling for cyber-physical systems," in *Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), 2017 IEEE PES,* 2017, pp. 1-6.

[31] P. Torr, "Demystifying the threat modeling process," *IEEE Security & Privacy,* vol. 3, *(5),* pp. 66-70, 2005.

[32] A. Vähä-Sipilä, "Software security in agile product management," *Software Security in Agile Product Management,* 2011.

[33] R. Kaksonen, "A Functional Method for Assessing Protocol Implementation Security." , University of Oulu, 2001.

[34] M. Böhme *et al*, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security,* 2017, pp. 2329-2344.

[35] S. K. Cha, M. Woo and D. Brumley, "Program-adaptive mutational fuzzing," in *Security and Privacy (SP), 2015 IEEE Symposium On,* 2015, .

[36] J. DeMott, "The evolving art of fuzzing," *Def Con,* vol. 14, 2006.

[37] D. Duran, D. Weston and M. Miller, "Targeted taint driven fuzzing using software metrics," 2011.

[38] P. Garg, "Fuzzing – Application and File Fuzzing," *Application Security,* Jan 4, 2012.

[39] P. Garg, "Fuzzing-mutation vs. generation," *Exploit Development,* Jan 4, 2012.

[40] M. Gustafsson and O. Holm, "Fuzz Testing for Design Assurance Levels." , Linköping University, 2017.

[41] Y. Li *et al*, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering,* 2017, pp. 627-637.

[42] P. Gutmann, "Fuzzing Code with AFL," *;Login:,* pp. 11-14, 2016.

[43] S. Ognawala, A. Petrovska and K. Beckers, "An Exploratory Survey of Hybrid Testing Techniques Involving Symbolic Execution and Fuzzing," *arXiv Preprint arXiv:1712.06843,* 2017.

[44] B. Shastry *et al*, "Static exploration of taint-style vulnerabilities found by fuzzing," *arXiv Preprint arXiv:1706.00206,* 2017.

[45] A. Takanen, "Fuzzing: The past, the present and the future," in *Actes Du 7eme Symposium Sur La Sécurité Des Technologies De L'Information Et Des Communications (SSTIC),* 2009, pp. 202-212.

[46] A. Takanen, J. D. Demott and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance.* Artech House, 2008.

[47] P. Tsankov, M. T. Dashti and D. Basin, "SECFUZZ: Fuzz-testing security protocols," in *Automation of Software Test (AST), 2012 7th International Workshop On,* 2012, pp. 1-7.

[48] B. West and M. Wengelin, "Effectiveness of Fuzz Testing High-Security Applications - A Case Study of the Effectiveness of Fuzz-Testing Applications with High Security Requirements." , KTH Royal Institute of Technology, 2017.

[49] J. Viide *et al*, "Experiences with model inference assisted fuzzing." in *2nd USENIX Workshop on Offensive Technologies (WOOT),* 2008, .

[50] J. Fernandez, L. Mounier and C. Pachon, "A model-based approach for robustness testing," in *IFIP International Conference on Testing of Communicating Systems,* 2005, pp. 333-348.

[51] S. M. A. Shah *et al*, "Robustness testing of embedded software systems: an industrial interview study," *IEEE Access,* vol. 4, pp. 1859-1871, 2016.

[52] A. Shahrokni and R. Feldt, "A systematic review of software robustness," *Information and Software Technology,* vol. 55, *(1),* pp. 1-17, 2013.

[53] M. Susskraut and C. Fetzer, "Robustness and security hardening of COTS software libraries," in 2007, pp. 61-71.

[54] N. Stephens *et al*, "Driller: Augmenting fuzzing through selective symbolic execution." in *Ndss,* 2016, pp. 1-16.

[55] A. Dyck, R. Penners and H. Lichter, "Towards definitions for release engineering and devops," in *Release Engineering (RELENG), 2015 IEEE/ACM 3rd International Workshop On,* 2015, .

[56] P. Godefroid, M. Y. Levin and D. A. Molnar, "Automated whitebox fuzz testing." in *Ndss,* 2008, pp. 151-166.

[57] N. P. Kropp, P. J. Koopman and D. P. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium On,* 1998, pp. 230-239.

[58] C. Miller, "Babysitting an army of monkeys," *CanSecWest,* 2010.

[59] K. Serebryany, "Continuous fuzzing with libFuzzer and AddressSanitizer," in *Cybersecurity Development (SecDev), IEEE,* 2016, .

[60] J. Wisnowski, "Advanced Automated Software Testing Implementation Guide - STAT COE-Report-01-2017," *Stat T&e Coe,* 2017.

[61] S. Huopio, "A rugged nation," in *The Fog of Cyber Defence*, J. Rantapelkonen and M. Salminen, Eds. Helsinki: National Defence University, 2013, pp. 127-135.

[62] Z. Durumeric *et al*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference,* 2014, .

[63] J. Eronen *et al*, "Software vulnerability vs. critical infrastructure-a case study of antivirus software," *International Journal on Advances in Security,* vol. 2, *(1),* pp. 72-89, 2009.

[64] S. SORSA, "Protocol Fuzz Testing as a Part of Secure Software Development Life Cycle." , Tampere University of Technology, 2017.

[65] J. Stark, "Product lifecycle management," in *Product Lifecycle Management (Volume 1)*Anonymous 2015, .

[66] B. Flyvbjerg and A. Budzier, "Why your IT project may be riskier than you think," *Harward Business Review,* pp. 23-25, "Sep ". 2011.

[67] D. Russo and P. Ciancarini, "Towards antifragile software architectures," in *4th International Workshop on Computational Antifragility and Antifragile Engineering,* 2017, pp. 929-934.

[68] N. Taleb, "The black swan: Why don't we learn that we don't learn," in *Highland Forum 23,* Las Vegas, 2004, .

[69] W. W. Wu, G. M. Rose and K. Lyytinen, "Managing black swan information technology projects," in *System Sciences (HICSS), 2011 44th Hawaii International Conference On,* 2011, pp. 1-10.

[70] N. N. Taleb, *Antifragile: How to Live in a World we Don'T Understand.* 2012*3*.

[71] D. Russo and P. Ciancarini, "A proposal for an antifragile software manifesto," in *3rd International Workshop on Computational Antifragility and Antifragile Engineering,* 2016, pp. 982-987.

[72] M. Monperrus, "Principles of antifragile software," in *Companion to the First International Conference on the Art, Science and Engineering of Programming,* 2017, pp. 32.

[73] J. S. Levin, S. P. Brodfuehrer and W. M. Kroshl, "Detecting antifragile decisions and models lessons from a conceptual analysis model of service life extension of aging vehicles," in *Systems Conference (SysCon), 2014 8th Annual IEEE,* 2014, pp. 285-292.

[74] P. Pietikäinen, A. Kettunen and J. Röning, "Steps Towards Fuzz Testing in Agile Test Automation," *International Journal of Secure Software Engineering (IJSSE),* vol. 7, *(1),* pp. 38-52, 2016. Available: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/IJSSE.2016010103. DOI: 10.4018/IJSSE.2016010103.

[75] J. Song and J. Alves-Foss, "The DARPA Cyber Grand Challenge: A Competitor's Perspective, Part 2," *IEEE Security & Privacy,* vol. 14, *(1),* pp. 76-81, 2016.