TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Software Engineering Department

Michailas Ornovskis 204790IVCM

Secure Software Development Lifecycle Reference Meta-Architecture

Master Thesis

Supervisor

Dr. Hayretdin Bahşi

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia Teaduskond

Michailas Ornovskis 204790IVCM

Turvalise tarkvaraarenduse elutsükli referents metaarhitektuur

Magistritöö

Juhendaja

Dr. Hayretdin Bahşi

Tallinn 2022

Author's declaration of originality

I hereby certify that I am the sole author of this thesis and that it has not been presented for the examination or submitted anywhere else for defense purposes. All used materials, references to the literature, and work of others have been cited.

Author: Michailas Ornovskis

14.05.2022

Abstract

The ability to choose information security controls for the development pipelines and other types of software development models based on clear Secure Software Development Lifecycle ((S)SDLC) reference meta-architecture, its components, software development methodologies criteria, and other elements will enable organizations to achieve measurable levels of security in an optimal fashion.

The main objective of this study is to create such a reference meta-model, that would be then transformed into a reference architecture for the DevSecOps methodology and to calculate the most optimal controls for specified selection of software errors. The calculation method is based on the Analytic Hierarchy Process (AHP) and the usage of software development methodology principles as an input for its criteria.

During the control prioritization process for each selected software error, the optimality factor is found in the overall weighted sum value across all errors in general. This way will enable organizations to find a suitable finite selection of top controls that would be effective for a given scenario – based on the methodology, coding language, software errors, and unique threats applicable to the operating environment.

The results show that reference architecture model follows industry controls selection in general but is more precise in case of specific organization and given software errors selection. The overlap with industry architectures and their control selection is significant, yet the resulting architecture is specific, and each control selection is justified.

This study could be further expanded into calculating specific controls and defining architectures for other methods and even become more specific in automating security controls selection for them.

The thesis is written in English, and it comprises 58 pages of text, 6 chapters, 19 figures and 11 tables.

Annotatsioon

Võimalus valida tarkvara konveierite ja muud tüüpi tarkvaraarendusmudelite jaoks infoturbe kontrolle, mis põhinevad selgel turvalise tarkvaraarenduse elutsükli ((S)SDLC) referents metaarhitektuuril, selle komponentidel, tarkvaraarenduse metoodika kriteeriumidel ja muudel elementidel, võimaldab organisatsioonidel saavutada mõõdetav turvatase optimaalsel viisil.

Selle töö põhieesmärk on luua selline referents metamudel, mis transformeeriks seejärel DevSecOps metoodika referents arhitektuuriks ning arvutada tarkvaravigade kindlaksmääratud hulga jaoks kõige optimaalsemad infoturbe kontrollid. Arvutusmeetod põhineb analüütilise hierarhia protsessil (AHP) ja tarkvaraarenduse metoodika põhimõtete kasutamisel selle kriteeriumide sisendina.

Iga valitud tarkvaravea infoturbe kontrolli prioriseerimise protsessi käigus leitakse optimaalsustegur kõigi vigade üldise kaalutud summa väärtuses. See võimaldab organisatsioonidel leida sobiva lõpliku valiku infoturbe kontrolle, mis oleksid antud stsenaariumi jaoks üldiselt tõhusad – lähtudes metoodikast, programmeerimiskeelest, tarkvaravigadest ja operatsioonikeskkonna uniklaasetest ohtudest.

Tulemused näitavad, et referents arhitektuuri mudel järgib üldiselt tööstusharu kontrollide valikut, kuid on täpsem konkreetse organisatsiooni ja antud tarkvaravigade valiku korral. Kattuvus tööstuse arhitektuuride ja nende infoturbe kontrollide valikuga on märkimisväärne, kuid magistritöö meetodist tulenev arhitektuur on spetsiifilisem ja iga infoturbe kontrolli valik on õigustatud.

Seda tööd võiks veelgi laiendada spetsiifiliste infoturbe kontrollide arvutamisele ja muude meetodite põhinevate arhitektuuride määratlemisele ning isegi nende jaoks infoturbe kontrollide valiku automaatsel valikul.

Lõputöö on kirjutatud inglise keeles ja sisaldab 58 lehekülge teksti, 6 peatükki, 19 jooniseid ja 11 tabeleid.

List of abbreviations and terms

SAST	Static Application Security Testing
SCA	Software Composition Analysis
DAST	Dynamic Application Security Testing
NIST	National Institute of Standards and Technology
SDLC	Software Delivery Lifecycle
(S)SDLC	Secure Software Delivery Lifecycle
ISC^2	The International Information System Security Certification Consortium
TOGAF	The Open Group Architecture Framework
AHP	Analytic Hierarchy Process
XP	Extreme Programming
RUP	Rational Unified Process
OWASP	The Open Web Application Security Project
CWE	Common Weakness Enumeration
API	Application Programming Interface
CR	Consistency Ratio
CI	Consistency Index
RI	Random Index
DOD	Department of Defense (United States)
PCI-DSS	Payment Card Industry Data Security Standard
COBOL	Common Business Oriented Language
SQL	Structured Query Language
XSS	Cross-Site Scripting
HTML5	HyperText Markup Language, version 5
J2EE	Java 2 Platform, Enterprise Edition
TLS	Transport Layer Security
RASP	Runtime Application Self Protection
IAST	Interactive application security testing
IDE	Integrated Development Environment
CI/CD	Continuous Integration / Continuous Deployment
MCDM	Multiple Criteria Decision Making

Table of Contents

List of Figures

List of Tables

1 Introduction

The following chapter introduces the research questions, thesis motivation, and scope novelty. It formulates the goals of the thesis and supplements the reader with the background information necessary for understanding the topic.

The following sections also include a literature review and gaps analysis sections.

1.1 Motivation

The current situation in the software development, secure deployment, and operation shows that security controls across all three control areas of information security – human, processes, and technology, are suggested to developers and infrastructure engineers. The majority of papers and Internet articles tend to suggest using a described selection of controls (such as Static Application Security Testing (SAST), Software Composition Analysis (SCA), Dynamic Application Security Testing (DAST), etc.) that are proposed in the studies themselves [1][2][3][62]. NIST in its withdrawn technical publication, focuses more on phases, different activities, and SDLC (Software Development Lifecycle) control gates. However, the controls and security activities suggested in the implementer tips do not have another basis than just being suggested [63]. Moreover, the security controls tend to be tied to DevOps or DevSecOps software development methodologies only, ignoring that different criteria, such as control effectiveness, coverage, implementation time, measurability, and simplicity, might be used in other models and methodologies. They could also have different weights and suitability depending on the methodology used – some manual controls such as granular security review might fit perfectly well into projects that follow classic methodologies, such as Waterfall, and have lengthy release cycles.

Up to the author's knowledge, there is no comprehensive study or literature source that would suggest a method to calculate security controls and activities suitability depending on the software development methodology, the coding language used, and possible security flaws they are meant to protect against.

The problem that the industry might face is that the same security controls with similar configurations are not used effectively across various methodologies, languages, and projects. This might result in controls not being used optimally, coverage being inadequate, and as a result – production deployments being vulnerable and insufficiently defended against the security flaws identified earlier in the threat modelling phase.

This approach is often one-sided as it is more likely to suggest technological controls over others, such as process-related (effective change management and defect management processes, for example) and human-related (job hiring procedures, comprehensive application security education). Also, such a suggestion for selecting a particular set of controls without knowing the exact environment, programming

language(s), or software development specifics does not seem right – vulnerabilities that are introduced due to software errors are different, and there is no "one size fits all" approach that can be used. For example, buffer overflow vulnerabilities are most common in the languages with manual memory allocation and management and do not apply to other languages that use a different approach, such as interpreted languages [61]. In addition, more security controls can be chosen compared to the limited selection presented in various study guides and grey literature articles.

Since code runs in the target environment, situation differs depending on which deployment method is chosen (artefact–based deployment, container, serverless function, package, or library) and which target environment code will operate in – container worker node with orchestrator and bare metal server versus smartphone have different security features; since the operation is part of the modern development process, it should be considered as well (but it often is not) [4][5].

As a consequence, there are many software errors and vulnerabilities in various products. The author believes that common thinking and selecting tailored controls from all control areas can reduce this number. It also must be noted that academic pieces echo the need for understanding how security fits into DevOps; to generalize this question, how security fits into generic Software Development Lifecycle [6][7].

1.2 Research Problem and Questions

The research problem and area are the rationale and methods behind a selection of particular security controls based on development needs. A particular research outcome is the Secure SDLC meta-architecture model that can be converted to a simpler security architecture for a particular methodology based on the selected number of security controls deemed to be most optimal in the current situation for a given development model, language, and most common security flaws. In addition, an example security architecture (the model to arrange control gates and security controls) for DevSecOps methodology is created.

The term meta-model, or in this case, a meta-architecture model, is captured in TOGAF (The Open Group Architecture Framework) and is defined as *"A model that describes how and with what the architecture will be described in a structured way"* [64].

As defined by Mohamed Sami, a meta-model in architecture is an abstraction layer upon the system, and a meta-meta-model is an abstraction layer of a meta-model that contains meta-entities [65].

For SDLC, such entities might will contain methodology, criteria, and security controls. This meta-model, or in this case meta-architecture model, could later be used to create an SDLC architecture system-level model for a given methodology.

This research hypothesizes that it is possible to create a working meta-architecture model that would fit together security controls from all three control areas and that it can convert to the particular software development methodology of DevSecOps.

In this research, the author will answer the following research questions:

- RQ1: Which most common security controls are possible in different stages of the development lifecycle?
- RQ2: Which security controls bring the most value based on software development methodology input to their cost and how to convert architecture meta-model into DevSecOps SDLC architecture model?
- RQ3: How to calculate and identify the most important security controls that take care of various software errors?

The first research question lies in understanding which possible security controls from the three pillars of information security – people, processes, and technology are applicable at which particular stage of SDLC, as defined in the United States Department of Defense Enterprise Reference Design [8]. Generalized control selection can still be used in other methodologies, but the scope is to tie them to DevSecOps stages.

The second research question focuses on identifying a method to understand controls with the most value, given the DevSecOps methodology and criteria that define it.

The third research question focuses on calculating the most important security controls from all three pillars for a generalized set of software errors as defined in the Seven Pernicious Kingdoms taxonomy [9].

1.3 Limitations and Assumptions

The following thesis results and methods are applicable in an enterprise that the author works in (Swedbank Group) since the method used to assess security controls selection, their placement across different stages of SDLC, DevSecOps criteria ratio, criteria values for different security controls versus software errors is Swedbank Group expert-based opinion with results validation. Results in other companies, applied to other methodologies, specific languages, and specific software errors might differ.

In order to apply a given method to a particular enterprise, all meta-architecture model steps need to be revisited, and new architecture needs to be calculated, given input to all relevant security controls and criteria.

Using the AHP [10] multiple-criteria decision-making methodology to align criteria and calculate the most effective security controls, the author will be able to show a method of their identification. Controls ranking is done according to a generalized set of most common (32) software flows based on languages listed in the Seven Pernicious Kingdoms taxonomy. The controls selection method depends on the number of security controls that the enterprise can afford and its risk appetite. At the same time, some security controls are effective against multiple software errors. By calculating their appearance rate as effective against software flaws, it is possible to

determine the mathematical mode – count of appearance. The assumption is that the controls that are effective against more software flaws should be preferred since using specific controls (against specific flaws) in the pipeline leads in unnecessary pipeline sophistication, which could otherwise be avoided by using more common controls that are effective against multiple software flaws at once.

1.4 Scope and Goal

The goal and main deliveries of this study are:

• (S)SDLC Reference Meta-architecture model

• DevSecOps Reference Architecture and its graphical representation

• Most optimal security steps and controls identification for DevSecOps model

• A method to identify these controls for DevSecOps model and its library of controls.

The scope of this study will stay within the boundaries of (S)SDLC, security controls, software error taxonomies, and reference architecture models. The financial cost of controls implementation, detailed review of the pipelines, their architectures, security controls, software errors, and their descriptions is out of the scope.

There are two main limitations – security controls selection will be based on expert-driven opinion; also, quantitative evidence of security controls implementation and effectiveness is not possible within such a short timeframe – the adoption of security architecture models varies in different organizations but usually takes quite some time; however, the measurement should base upon the number of software security-related errors that reach production environment; therefore outcome and effectiveness are measurable.

One minor gap is the method flexibility – some additional efficiency-related aspects might be taken into account based on the organization – such as risk appetite, uniqueness of the environment, programming language, existing controls, and tolerance for having vulnerabilities in the production environment.

1.5 Novelty

The novelty of a master thesis lies in a combination of controls selection applicable against software errors based on given threat taxonomy and usage of this selection to create particular SDLC security architecture based on defined criteria, language(s), and methodology. While there are other studies that use similar security control selection methods [43], controls and their alternatives selection are not applicable against particular software errors affecting the SDLC. Controls and control gates selection might depend more on the identification of software flaws affecting the product and its different pipeline phases, while precise controls suggestions in

this thesis are driven by methodology criteria as defined in the methodology description.

2 Background Information and Literature Review

This chapter provides background information necessary for understanding the thesis, such as Software Development Lifecycle and its phases, Analytic Hierarchy Process (AHP) multiple-criteria decision-making method, Software Security Errors Taxonomy, and SDLC Reference Architectures. This chapter explains why particular methods and taxonomies were chosen and provides a methodology basis.

In addition, it also discusses related work in the field.

2.1 The analysis of identified studies

The reviewed literature mainly focuses on analyzing practices of embedding security into the existing software development process, separately analyzing challenges [11][12], tools [13], security controls, review of taxonomies [14], and others. The author found it interesting that there is a lack of the original research in this field – mostly one or another approach references back to some book or comprehensive study (in the best case scenario) – such as ISC^2 recommendations, Department of Defense publications [1][8][15][16], other grey literature sources [4][17] or simply "internet artifacts" [18].

2.2 Software Development Lifecycle

Before reaching the target operation environment, the software undergoes different phases of the development process. Depending on the methodology or development method chosen, these phases can be different and might include various software quality gates and security controls.

Moreover, the overall logic of development methodologies sometimes suggests the usage of different quality gates and controls, depending on the project nature, its risk appetite, timeframes, and other criteria.

There are various methodologies that have been developed during the last 50 years – examples include Waterfall, Spiral, V-Model, Agile, DevOps, and others [19]. Different methodologies have different advantages and disadvantages and are commonly chosen based on organization needs, projects and concerns.

2.3 Waterfall model

Phases, their iteration, and overall logic depend on the methodology chosen. Some methodologies, such as the Waterfall development model, do not have feedback

loops – software release and related processes are moving strictly in one direction from the requirements phase towards the maintenance phase, as shown in Figure 1.

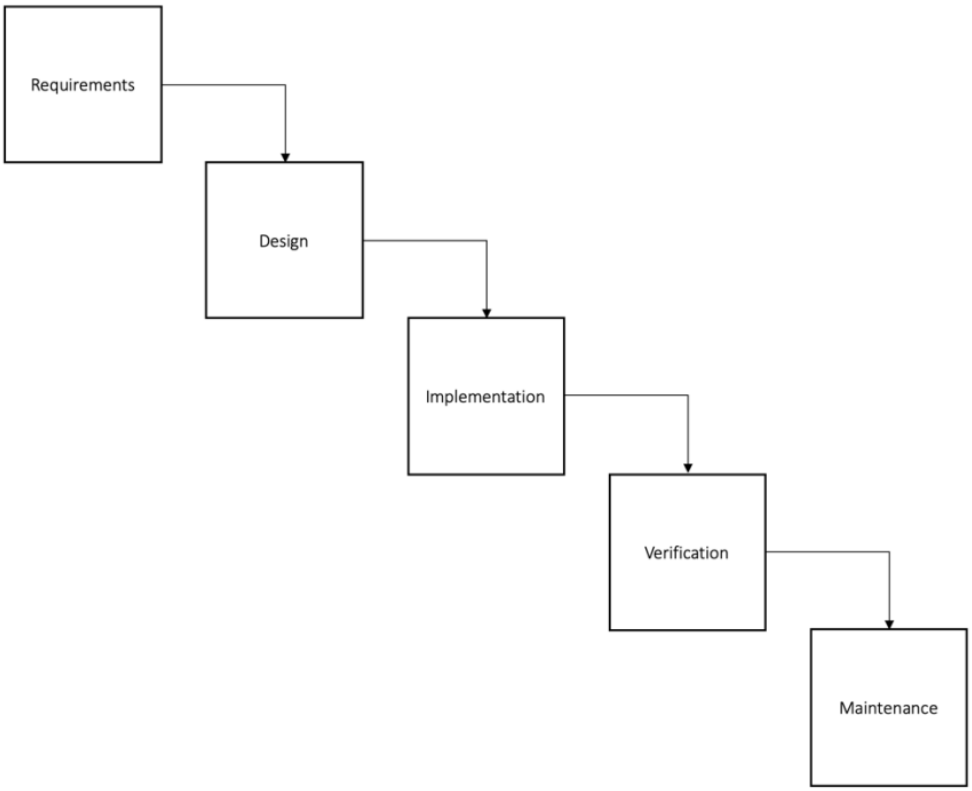

Figure 1. The Waterfall development model [20]

The waterfall development model does not allow moving back in its phases and expects the previous phase to be complete before proceeding further to the next phase, thus being sequential.

This causes a problem in the development process since the classic Waterfall model expects starting each software release from scratch – back from the requirements phase. In case software errors are discovered in the process – correcting a mistake or moving backward in one or two phases is not ordinarily possible.

The waterfall model has derivatives that fix mentioned flaw and introduce feedback to one or two phases backward. One example of such an advanced Waterfall model is the Waterfall model with Royce's iterative feedback, as shown in Figure 2 [21].

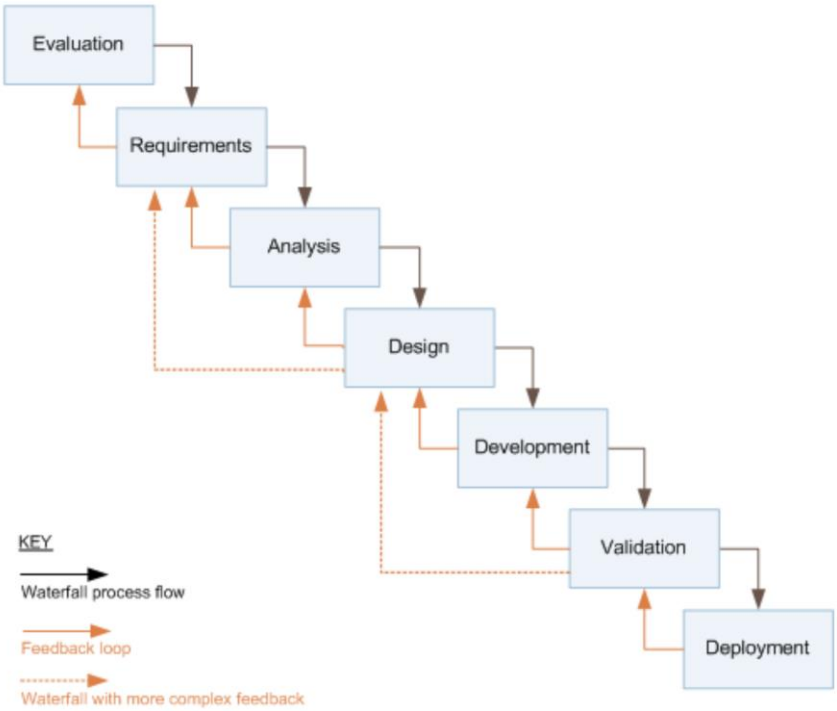

Figure 2. The Waterfall development model with Royce's iterative feedback [21]

This model allows iteration between adjacent phases. In some versions, it allows moving backward more than one phase. For instance, if one or more flaws are discovered during the validation phase, the issue will again be corrected in the previous development phase. Similarly, more severe issues related to software design can be addressed from the validation phase in case this is required.

The Waterfall model has some advantages compared to other ones – such as having a clear structure and defining goal it is aimed to achieve early, but it also has significant disadvantages, such as fixed requirements, inability to adapt to the changes, project overall high risks and costs [20][22]. The software itself is only produced during the late phases of the Waterfall model [19].

There are other models based on Waterfall, such as the V-Shaped model, which is mainly seen as a Waterfall extension with additional validation and verification steps, but all these models have a fatal flow – they are sequential and require starting from the beginning each time a new product, release or version are developed [23].

The necessity to start each project back from the requirements phase and incompliance with modern development practices and business needs, where standard products typically have many versions, subversions, and single release trains, has created a need for iterative models.

2.4 Iterative models

Iterative models are meant to eliminate initial steps from the iterative process cycle and concentrate on product development with fixed initial requirements. These SDLC

models commonly have the initial planning stage done once for the entire project, and the iterative process itself concentrates on the actual release planning, requirements, design, development, and testing stages. Initial planning can indeed take a significant amount of time, but once the project's initial requirements are set, they do not change in time [23]. Figure 3 shows common SDLC iterative model steps.

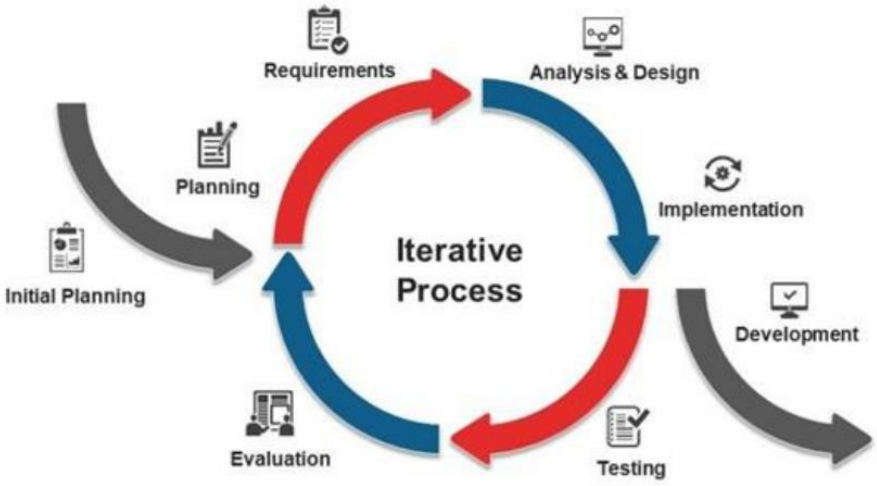

Figure 3. The SDLC iterative model [23]

Different iterative models are fit for different project needs. For example, the Spiral model combines the advantages of Waterfall's rigorous controlled features and evolutionary nature of the iterative process and allows to start small by creating a proof of concept prototype and later feed it with necessary features, still going through iterative cycles of the same major project. Figure 4 displays the steps and phases of the Spiral model.

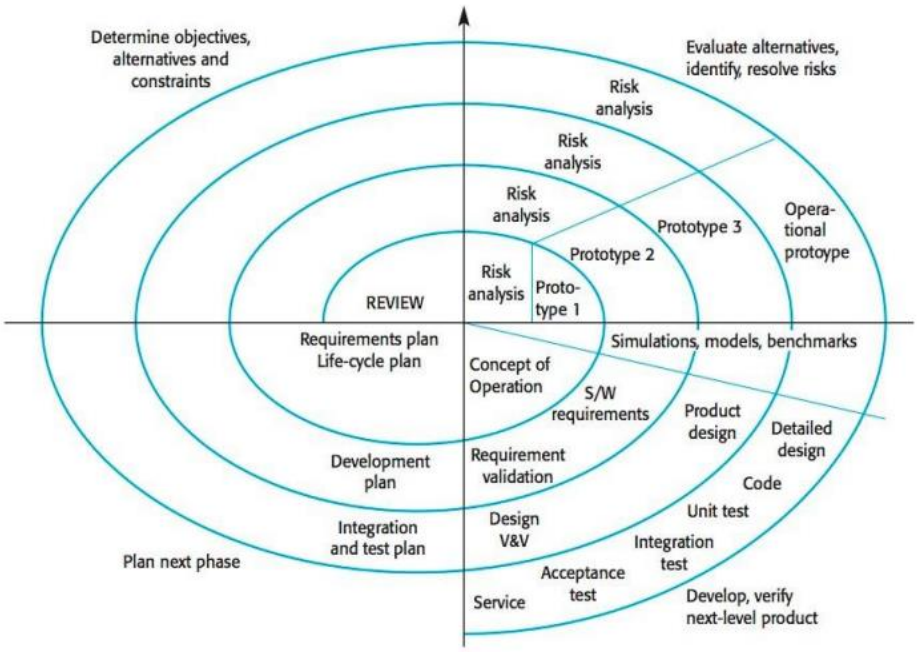

Figure 4. The Spiral development model [23]

Spiral models favor the development of large and complex systems [24].

Due to its various advantages of moving through the same phases in each prototype release – thus a possibility of quality gates automation, producing a working prototype by the end of each release and performing risk analysis at the beginning of each sequential release step the Spiral model has become popular. The last prototype of the Spiral model is called the operational prototype. After passing final quality gates, including but not limited to unit, integration, and acceptance tests, it reaches production deployment and service operation.

The Spiral model is complex and requires manual user involvement at each cycle, at least at the stage of risk analysis. While it might be a well-balanced fit for the development of complex mission-critical systems with a low-risk appetite, it is considered to be heavy and complex for smaller teams and projects. In addition, Spiral model interoperability (including sharing of libraries, quality gates configuration, and security controls) between different projects is limited due to the high degree of customization.

RUP model is also considered to be a heavyweight in the industry and shares the same complexity disadvantage as the Spiral model [25].

2.5 Modern development methods and methodologies

Disadvantages of development methodologies discussed in the previous chapter, such as the inability to change requirements during the development phases, model complexity and overall velocity, and other similar factors, made them unsuitable for smaller teams and projects. It resulted in the development of lightweight methodologies, methods, and derivatives – such as Agile process model, Extreme Programming (XP) design method, Scrum/Kanban methodologies and brand new SDLC methodologies – such as Dev and DevSecOps.

One of the essential milestones of realizing this unsuitability and seeking better ways to develop software resulted in creating the Agile Manifesto in 2001 [26]. The Agile Manifesto has four core principles:

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan [26]*

Focus on responding to change implies ability to bring in changes in the development process; otherwise, previously impossible change of the initial requirements during the process. None of the previously discussed models allowed that since the initial setting of the requirements is done only once during the early stages of the project.

Working software over comprehensive documentation eliminates the usage of the Waterfall model since the model is formal and implies producing extensive documentation in each stage [27].

Scrum methodology has raised on top of the Agile Manifesto and suggested changing the way of working, including roles, methods, and iterations. This resulted in the creation of iterative cycles - sprints, where features to the product are added from the common backlog, and requirements might change between the sprints. The development itself is continuous and is monitored in often daily feedback meetings.

Figure 5 displays the Scrum flow.

Figure 5. The Scrum methodology flow [28]

Modern development methodology requires the same modern SDLC methodology that would adopt changes to the requirements, cater for sprint iterations, and the ability to deploy rapidly – preferably with a high degree of automation. Thus, DevOps and its derivative DevSecOps were born.

DevOps methodology stands for a combination of development and operations, while DevSecOps adds a security aspect to the methodology.

The DevOps methodology, similarly to the Agile Manifesto, stands on four principles:

- Culture
- Automation
- Measurement
- Sharing

These principles together assemble the acronym CAMS [6].

Together, these principles imply multiple practices and shift industry towards certain choices, as DevOps is broadly adopted and has become the de-facto standard methodology of modern development and operation, with up to 88% of organizations adopting it [29].

The culture principle requires a higher degree of trust between development and operation teams, removing the silo-based approach.

The automation principle implies automating releases, quality gates, and overall increasing software development velocity by automating all possible steps to a high degree. Automation also implies better customer and developer experience through self-service phases in release pipelines, as displayed in Figure 6.

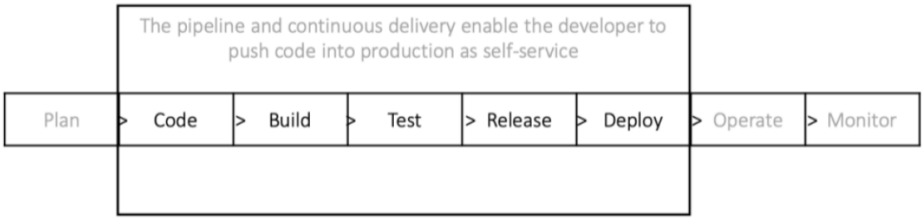

Figure 6. The DevOps methodology developer self-service phases [6]

The measurement principle is a derivative of automation and allows monitoring of important metrics directly from the pipeline with the goal of adjusting the process. Measurement combined with automation produces automated real-time reporting of software builds and indicates overall release health.

Sharing principle primarily implies sharing good and bad experiences across the organization – this includes several important atomic security controls to be seen later, such as sharing successful software libraries and learning from mistakes.

DevOps does not directly involve security aspects and mandate for security controls in its pipelines – initially, security was bolted on instead of being an integral part of this SDLC methodology itself [30].

Such an approach resulted in an unacceptable situation of not being able to conduct security and risk management activities of these high-velocity pipelines.

To fix the matter, security was added into various phases of the DevOps, resulting in the creation of the DevSecOps SDLC methodology.

2.6 DevSecOps

Due to being based on the same automation principles, DevSecOps prefers the usage of automated security checks as much as possible. Pipelines favor automated tools that provide security controls that can run in a predetermined time. Both DevOps and DevSecOps advocate for having so-called "coffee tests "or "coffee builds "– being able to run software build through the pipeline while drinking one or two cups of coffee, equivalent to five to ten minutes [31].

DevSecOps is characterized by putting security controls and activities in the early stages of the pipeline and preferring automated controls over manual ones [32].

There are multiple reasons for doing this; one of them is identifying security flaws in the software as early as possible until the build reaches the late stages of development and operations. This reduces the cost of correcting mistakes.

The cost of repairing software bugs or security flaws once the software release reaches the production stage can be very high compared to the early development stages. The study by Muhammad Asad and Shafique Ahmed shows that this cost might vary 30-60 times, as shown in Figure 7 [33].

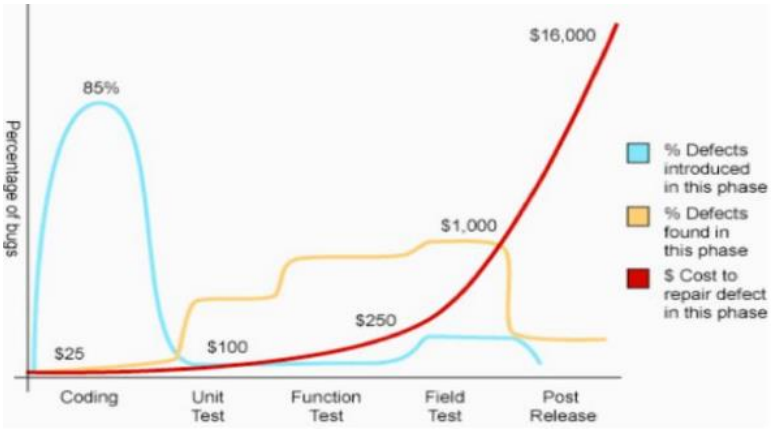

Figure 7. Cost of the software bug in different phases of development [33]

Inherited from the four DevOps principles, DevSecOps also advocates for the usage of automated metrics in the key focus areas for both development, operations, and security purposes. While there are different types of controls, they usually get suggested in different studies as industry best practices without tailoring their selection to the organizational needs.

DevSecOps is portrayed similarly to DevOps, with added security controls in different stages of development and operation. US Department of Defense, in its Enterprise DevSecOps Reference Design document, thoroughly addresses the topic of using DevSecOps methodology in a large-scale enterprise, describing topics of software factories, Continuous Integration and Continuous Delivery (CI/CD) pipelines, and suggested security controls.

In an earlier version of the same document, the DevSecOps software lifecycle model already contained various suggested security controls placed against different DevOps stages, as shown in Figure 8.

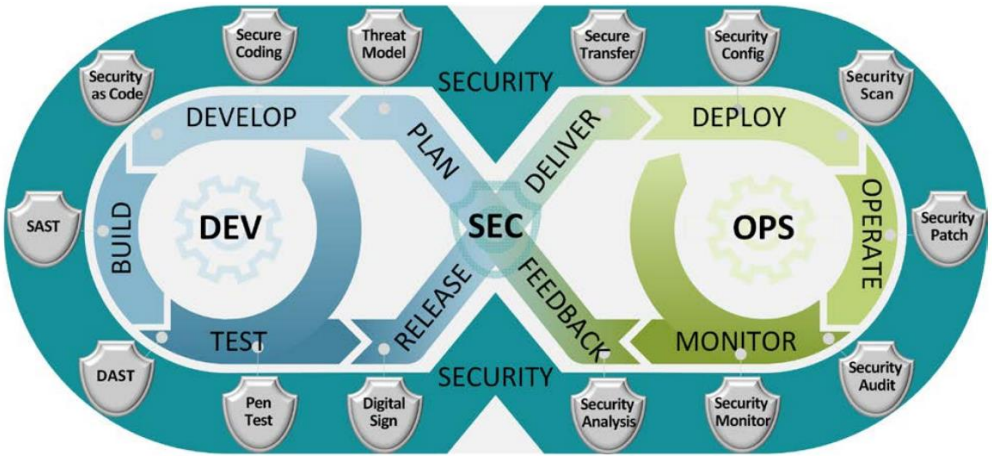

Figure 8. The United States Department of Defense DevSecOps software lifecycle model [8]

The document also states that *"Security is not a separate phase of the DevSecOps lifecycle; rather security activities occur in all phases. This DevSecOps security practice facilitates automated risk characterization, monitoring, and mitigation across the application lifecycle. "[8]*. However, these activities and security controls are again suggested as possible, and the document does not state exactly why they became industry best practice or industry default selection and whether they are applicable to all programming languages, companies, and pipelines.

2.7 Security control types

It is common to divide security controls into different categories – the industry agrees that security controls can be either preventive, detective, or corrective and either administrative, technical, or physical if divided by their type [34].

Preventive controls are meant to prevent security events from happening, detective controls are used to detect security events taking place, and corrective controls are meant to correct or remediate damage caused by the security events once they took place.

Security controls functions division largely depends on the area of their application – for instance, technical controls depend on technology – such as usage of antivirus against malware threats. Administrative controls might include security education and awareness training – these are the controls that consist of policies, guidelines, and expectations, combined with enabling users to do something. Physical controls are a separate family of controls and are tangible. For example, mantrap preventing entry into a building, smoke, heat detectors, and fire suppression systems are all physical controls of either detective or corrective nature, depending on their goal.

Similar logic applies to software and application security – controls can be divided into different areas depending on their field of application and nature. Moreover, different security controls help against different types of software vulnerabilities and threats posed in the production environment.

2.8 Software Security Errors Taxonomy

While security controls can prevent, halt or remediate vulnerabilities and damage caused by them, application security uses several well-known taxonomies to classify these. One of the most well-known taxonomy is the OWASP (Open Web Application Security Project) [36]. The project started with issuing the famous top ten most critical web application vulnerabilities that need to be addressed while building applications. It later brought more complete classification, expanded the scope, and started producing guidelines on how to develop applications securely [37].

The second famous taxonomy that is used in the industry is Mitre CWE (Common Weakness Enumeration). CWE enumerates weaknesses in multiple areas, including hardware design and software development. The list of weaknesses (or software flaws) is more comprehensive compared to OWASP and contains 699 entities at the time of writing the thesis [38]. The list is community-developed.

In 2005, the Seven Pernicious Kingdoms software security errors taxonomy was published [9].

Under this taxonomy, the errors are divided into seven kingdoms (or classes) with an additional class of environment errors. The ranking is done in order of importance to application and software security:

1. Input Validation and Representation

2. API Abuse

3. Security Features

4. Time and State

5. Errors

6. Code Quality

7. Encapsulation

**. Environment [9]*

Each kingdom from the taxonomy represents separate field from where software security errors can come from.

Input Validation and Representation kingdom focuses on injection, encodings, validation and other types of errors that result from trusting input.

API Abuse kingdom focuses on API-related types of errors, such as bad language practices, API misuse, dangerous functions, and restrictions.

Security Features kingdom focuses on software security errors that come from unsafe or incorrect usage of security functions – such as violating the privacy, least privilege concepts, mishandling passwords, and using weak cryptography.

Time and State kingdom enumerates weaknesses related to abusing time and state – such as race conditions, insecure temporary files, and mishandling threads and sessions.

Errors kingdom focuses solely on insecure error handling – such as catching overly broad extensions, being too verbose during error conditions, leaving stack traces visible to users etc.

Code Quality kingdom focuses on software errors caused by poor code quality – often forgotten aspect – memory leaks, usage of obsolete functions, uninitialized variables, and similar flaws.

The encapsulation kingdom focuses on software flaws caused by incorrect or insufficient encapsulation – the creation of boundaries to separate sensitive functionality and data from the rest of the environment.

Finally, the Environment kingdom, which is often presented separately, focuses on issues caused by production environment misconfiguration and lack of proper security controls. This kingdom described flaws related to Dockerfile misconfiguration, insecure deployment, insecure storage etc.

Since Seven Pernicious Kingdoms taxonomy provides comprehensive insight also into environment issues, I use it to classify software flaws and applicable security controls in this thesis. The taxonomy has evolved into Fortify Taxonomy due to one of the original authors moving to work in this company. At the time of writing this thesis, it contains a list of 1079 classified weaknesses that can be filtered by language, framework, kingdom, applicable security standard, and other categories [39].

2.9 Multiple Criteria Decision Making

There are several alternatives when it comes to decision-making methodologies. A typical relationship between major alternatives is a unique goal that needs to be achieved based on multiple criteria with different weights and several alternatives to consider. Criteria selection is based on the problem nature, interconnection of different criteria, criteria weights, and others. [40].

There are multiple studies that address the problem of method selection [41],[42] and a wide variety of alternatives to consider from.

However, since one of the research questions is application security controls prioritization based on the MCDM method, such studies and comparisons have already been made for information security controls selection and prioritization [43].

The method used in the study is called Fuzzy AHP of Fuzzy Analytic Hierarchy Process. The only difference between this method and the AHP is the use of so-called triangular numbers, which tend to show more precise results in calculations [44] [45].

However, the difference for this thesis is not that high since proposed security control efficiency values, as will be shown later, differ significantly. This fact advocates for the usage of a method with more simple calculus – the AHP.

2.10 Analytic Hierarchy Process

Analytic Hierarchy Process is a quantitative method of multi-criteria decision making used for solving various problems across different fields of study, including economics, information technology, and particularly information security controls prioritization, risk management, resource allocation [46], and others [47][43].

In an original publication by R.W. Saaty, the inventor of the method, it is described as the general theory of management that uses ratio scales and paired comparisons [48].

The goal of the method is to select the best alternative using scales of criteria and sub-criteria (in case they are introduced) among multiple choices available.

Figure 9 shows the AHP scheme of hierarchy levels in its simplest form, with a single level of criteria.

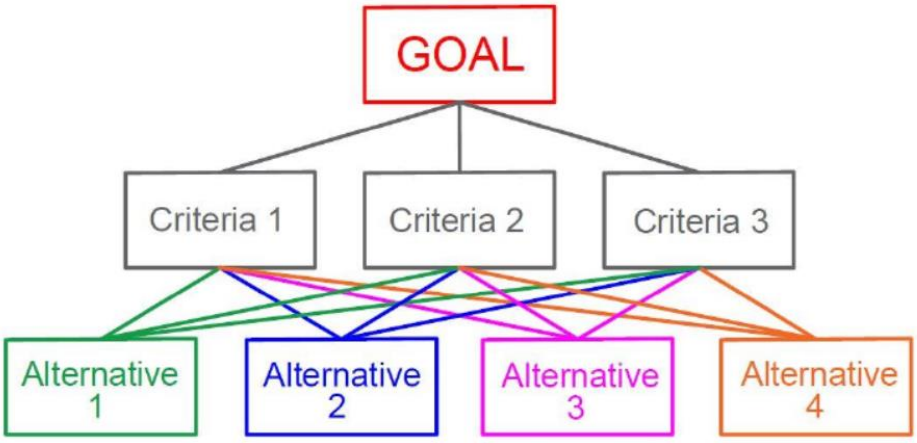

Figure 9. The AHP process scheme example [49]

The idea of the AHP method is described in six steps:

1. Definition of the unstructured problem,

2. Development of the hierarchy of interconnected decision-making elements that describes the problem,

3. Comparison of pairs of decision elements, using the Saaty comparison scale, to obtain input data,

4. Prioritization by calculating the relative weights of decision-making elements, which are afterwards combined into the total priority alternatives,

5. Checking of the consistency of the decision maker,

6. Obtaining of the overall ranking [49].

Saaty comparison scale is filled with measurement numbers between each pair of criteria, including reciprocal ratio for inverse comparison. The measurements are usually done during questionnaire interviews with experts and rating their opinion using an understandable scale of importance expressed with words and compared to numerical ratings, as shown in Figure 10.

Scale	Numerical Rating	Reciprocal
Extremely importance	9	1/9
Very to extremely strongly importance	8	1/8
Very strongly importance	7	1/7
Strongly to very strongly importance	6	1/6
Strongly importance	5	1/5
Moderately to strongly importance	4	1/4
Moderately importance	3	1/3
Equally to moderately importance	2	1/2
Equally importance	1	1

Figure 10. Saaty comparison scale for the AHP process [50]

If there are multiple experts involved in the questionnaire, usually the geometrical mean from several ratings is taken into account. However, other techniques are also used, such as arithmetic mean. Each criterion is then compared to each other criteria, including itself, and the ratio is calculated.

Ratio calculation results in the creation of the table, and after performing normalization, the normalized pair-wise matrix with criteria weights is calculated [49].

The resulting matrix needs to pass the CR test (Consistency Ratio). First, the consistency index needs to be calculated according to the formula in Figure 11.

$$CI = \frac{\lambda_{max} - n}{n - 1}$$

Figure 11. AHP Consistency Index calculation [49]

Lambda max value means the maximum eigenvalue of the calculated comparison matrix. The consistency ratio is then calculated according to the formula in Figure 12.

$$CR = \frac{CI}{RI}$$

Figure 12. AHP Consistency Ratio calculation [49]

RI is the value of the random consistency index determined by the table given by Saaty, as shown in Table 1.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RI	0.0	0.0	0.58	0.9	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

Table 1. Saaty Consistency Index table [50]

The *n* value stands for matrix dimension, and the appropriate consistency ratio test needs to be below *0,10* to be treated valid result. The consistency ratio test shows that the result is sufficiently accurate and that the weights can be used later in the calculations.

The overall ranking calculation in step six is a standard weighted average summation for each particular alternative, where the weights are taken from the matrix calculus.

2.11 SDLC Reference Architecture, Meta-architecture, and related work

There were multiple approaches to defining SDLC architecture views, its stages, security controls, and common components [33][51].

According to DOD reference [8], application DevSecOps processes are placed across different phases of the development and operation, and there are several best practices to follow, including involving multidisciplinary teams in the process design effort, automating most of the processes, and treating the entire DevSecOps lifecycle as iterative closed loop [8]. Both DOD and OWASP have published DevSecOps maturity models [52][53], which allow measuring team maturity by grading answers to various questions related to secure coding, organizational structure and so on. Each particular process architecture or, in other terms, process design description follows the same logic – the process is divided into phases, then the phases are covered with security activities and security controls, as shown in Figure 13.

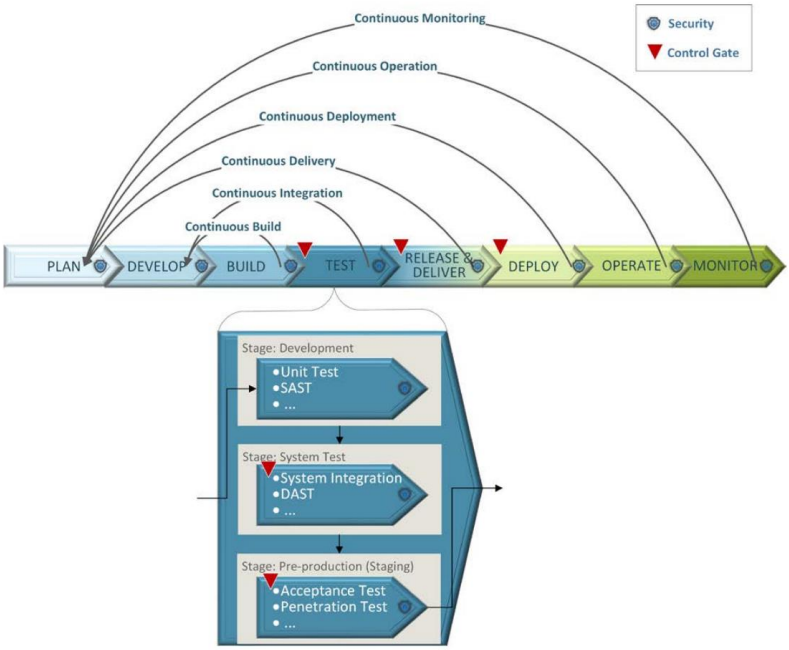

Figure 13. United States Department of Defence DevSecOps phases and security controls [8]

In DOD reference, DevSecOps architecture highlights eight distinct phases – plan, develop, build, test, release & deliver, deploy, operate and monitor. Since it is treated as closed loop, system decommission is not part of the architecture.

After the division, process continuity is highlighted by pointing arrows back to the planning phase since both DevOps and DevSecOps require continuous tolling feedback, as it was mentioned earlier.

Then, the same figure highlights activities, security processes, and controls in each phase. For example, the development phase requires SAST tooling and Unit testing to be included, while the test phase runs DAST tool and system integration tests.

MDA-SDLC architecture overview has a similar approach but different phases and controls, as shown in Figure 14.

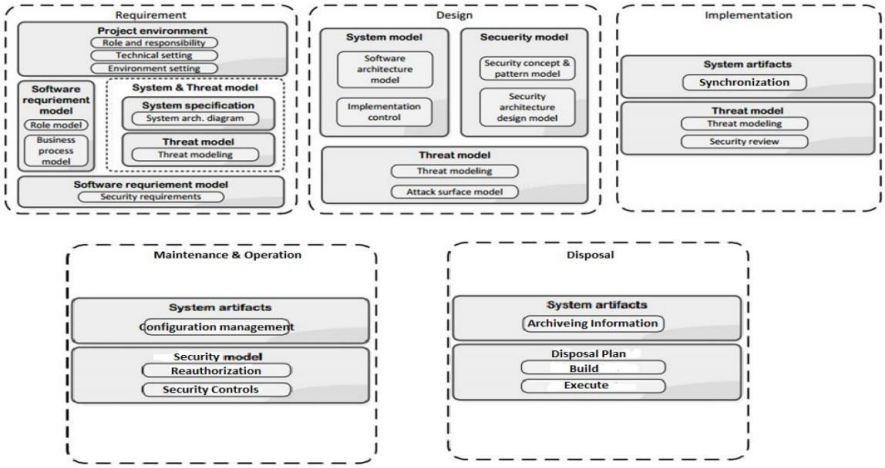

Figure 14. MDA-SDLC architecture overview [54]

It has five phases – requirement, design, implementation, maintenance & operation, and disposal. All stages contain activities such as software requirement modelling, threat modelling, disposal process, configuration management etc.

While such modelling construction is justified by referencing different case studies and best practice guides in selecting stages and controls, it does not explain the meta-architecture which is used to derive these, nor does it explain the justification of given phases and selected processes and controls. The meta-architecture is the meta-model that defines a particular architecture for a given scenario based on multiple criteria.

Therefore, there is a gap in the studies – no meta-architecture model with a method to derive phases, security controls, their number, and coverage based on multiple criteria such as the language used, software flaws, and security threats that can affect selected phases and other important factors.

3 Research methodology

The research methodology chapter explains how the results were achieved and which methods were in use. The methods include an interview with experts, expert-driven opinion about the controls, expert-driven validation, and both quantitative and qualitative measurements.

3.1 Research design

This section explains what research design looks like and which steps are included. The research design is built on the inference of creating a Secure SDLC meta-architecture, or (S)SDLC for short, which combines multiple steps needed to assemble the process of creation of other architectures applicable to SDLC based on different criteria, methodologies, and software error taxonomies.

The meta-architecture and created SDLC architecture for DevSecOps methodology are then validated together with calculated controls. The meta-architecture model with its derived DevSecOps Reference Architecture model is shown in Figure 15.

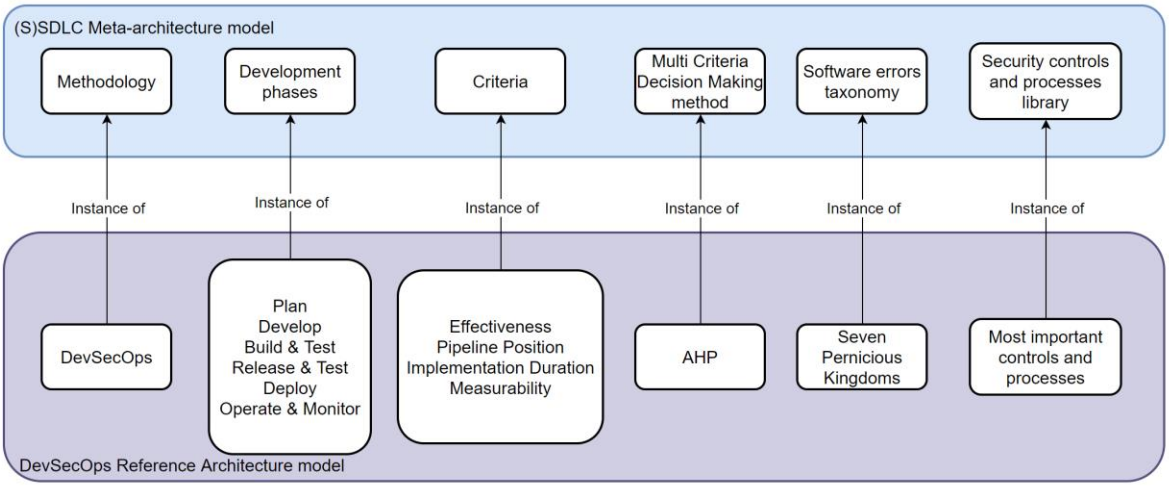

Figure 15. (S)SDLC meta-architecture model with derived DevSecOps architecture model

Such a model, using steps described in further sections, will be able to create DevSecOps Reference Architecture.

3.1.1 The controls library

I have assembled the library of security controls and practices based on my knowledge of the domain and have validated them together with the reference group of Security Architects and Offensive Security Officers from the Swedbank Group to see if any important controls are missing. The controls divided into categories are located in the Github repository [35]. These are the controls we see as applicable and acceptable in the area of application security. These controls and practices will be the ones used in the thesis.

3.1.2 The expert group

The expert group consists of seven security architects and two offensive security officers (penetration testers) from the Swedbank Group company. The architects have experience in different information security domains, including application security; their regular task is to consult business and development teams in matters of making SDLC pipelines secure and compliant with both internal security standards and industry regulations, such as PCI-DSS. The security architects also are responsible for performing regular security reviews of application development projects, target environments, and application delivery platforms.

The offensive security officers have substantial experience in testing for various flaws in developed applications in different languages – such as Java, Python, Node.js, and COBOL.

3.1.3 Development phases identification, methodology selection

For this research, it was decided to run trial against DevSecOps methodology. The phases of the methodology were aligned to reflect Swedbank Group view on the development process. In the original version presented by Department of Defense, DevSecOps methodology has eight phases [8]. For Swedbank Group use, these eight phases are converted into six, as shown in the following Table 2.

Swedbank DevSecOps phases	**DoD DevSecOps Reference Design phases**
Plan	Plan
Develop	Develop
Build & Test	Build
	Test
Release & Test	Release & Deliver
Deploy	Deploy
Operate & Monitor	Operate
	Monitor

Table 2. Swedbank Group versus US DoD DevSecOps phases

3.1.4 Applicable criteria identification and selection

Since the selected methodology is DevSecOps, both DevOps and DevSecOps important criteria are selected from the methodology definition. As was mentioned earlier, both methodologies have several important statements which imply the usage of automated controls over manual ones, which states the significance of controls measurability.

Since both methodologies also advocate for having coffee builds and modern development release cycles are frequent, security control implementation duration (time of executing checks) is a significant criterion as well.

DevSecOps advocates for the usage of security controls early in the pipeline, the same goes for other methodologies as well. Since the cost of repairing software errors grows multiple times while moving towards final phases of the development cycle security control placement in the phases, or pipeline position criteria in other words, is also important and needs to be taken into account.

The last criterion identified is security control effectiveness against given software security flaw – different controls are effective to a different degree. The effectiveness can be seen as the subjective value of handled security flaws – 80% effectiveness would mean that 4 out of 5 security flaws trigger a control reaction. The effectiveness criterion is important as well.

The author has identified four criteria – Effectiveness, Pipeline position, Implementation duration, and Measurability. After identification, the list of criteria was presented to the expert group for validation. It was agreed that they derive from DevOps principles, and it makes sense to use these four to proceed with the AHP process.

The AHP process layers were created, as shown in Figure 16.

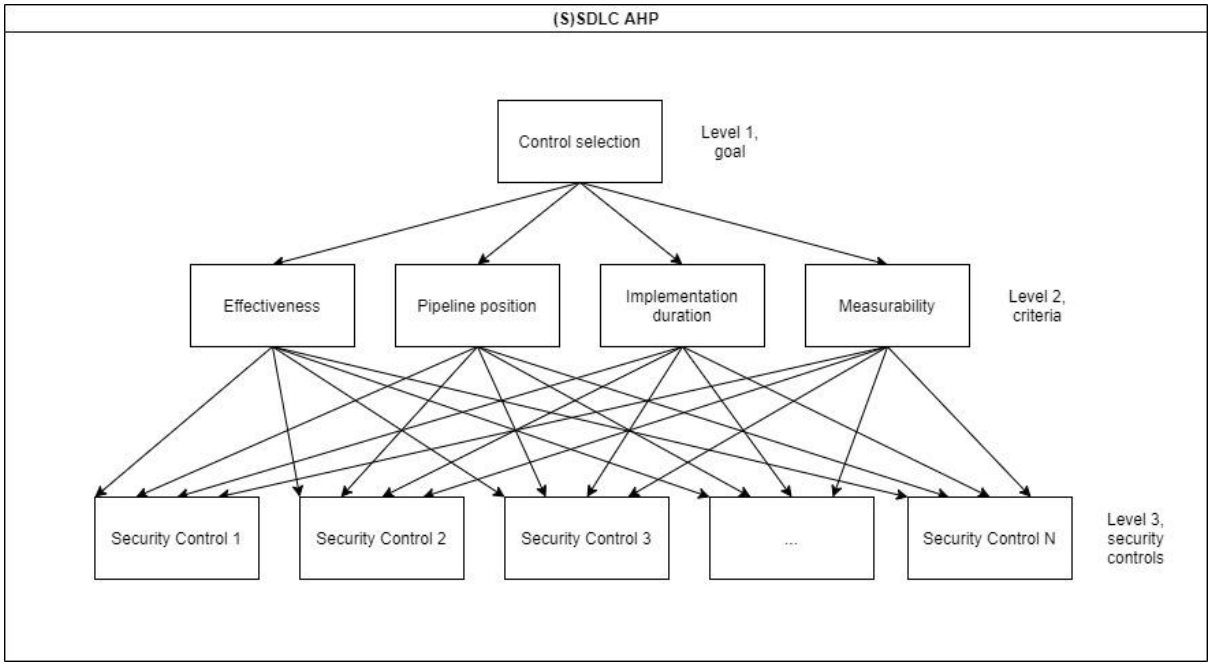

Figure 16. DevSecOps SDLC AHP process layers

The goal is to select suitable controls out of alternatives in Level 3; the Level 2 criteria correspond to the ones selected and verified using expert opinion.

The AHP process used in this thesis contains only a single layer of criteria (Layer 2). There are more sophisticated AHP models, where criteria might have sub-criteria, such as the division of measurability into two separate sub-criteria of completeness of coverage and usefulness of the results. In such a model, further calculation of criteria weights would entirely depend on the weights of their sub-criteria. According to the definition of four DevOps principles, there was no sense in dividing them into sub principles; therefore, the AHP process layers have single criteria layer.

3.1.5 AHP process to determine pair-wise comparison to criteria

Once the AHP structure was set, each expert reference group member got a task to compare each criterion as AHP requires in order to build a pair-wise comparison matrix. Using the Saaty scale from Figure 10, each representative compared each criterion according to the AHP process. The results were similar to Table 3, but individual values were collected.

The results were later calculated using the geometric mean formula and rounded against the closest measurement value from the reference scale.

$$C_i = \sqrt[n]{C_1 \times C_2 \times \ldots \times C_n}$$

Since there were nine experts in total, the *n* value is *9,* and each distinct pair-wise value was calculated separately. Reciprocal (inverse) values were taken from the reference scale, as shown in Figure 10.

The resulting pair-wise comparison matrix is shown in Table 3. This matrix shows relative criteria importance when compared to each other. For example, pipeline position is relatively two times more important compared to effectiveness.

Pair-wise comparison matrix	Effectiveness	Pipeline position	Implementation duration	Measurability
Effectiveness	**1**	**0,5**	**1**	**2**
Pipeline position	**2**	**1**	**1**	**3**
Implementation duration	**1**	**1**	**1**	**3**
Measurability	**0,5**	**0,333**	**0,333**	**1**
Sum	**4,5**	**2,833**	**3,333**	**9**

Table 3. AHP pair-wise comparison matrix

Using this table, the normalized pair-wise comparison matrix was calculated by dividing each value by the sum of the values in the respective column. The normalized pair-wise comparison matrix is shown in Table 4.

Normalized Pair-wise comparison matrix	Effectiveness	Pipeline position	Implementation duration	Measurability	Criteria weights
Effectiveness	**0,222**	**0,176**	**0,300**	**0,222**	**0,230**
Pipeline position	**0,444**	**0,353**	**0,300**	**0,333**	**0,358**
Implementation duration	**0,222**	**0,353**	**0,300**	**0,333**	**0,302**
Measurability	**0,111**	**0,118**	**0,100**	**0,111**	**0,120**
Sum	**1**	**1**	**1**	**1**	**1**

Table 4. AHP normalized pair-wise comparison matrix

Lambda value was then calculated for all criteria:

$$\lambda_{criteria} = (C_1 + C_2 + C_3 + C_4)/W_{criteria}$$

where sum of respective criteria values is divided by criteria weight [48].

The maximum lambda is then calculated as an arithmetic average of all lambdas:

$$\lambda_{max} = (\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4)/4$$

Since we have four criteria, there are four lambdas.

The lambda max value turned out to be *4.04584115*. Then, the Consistency Index and Consistency Ratio were calculated according to formulas in Figure 11 and Figure 12, respectively.

Consistency Index (CI) turned to be *0.015280372* and Consistency Ratio (CR) *0.016978191*. Since the Consistency Ratio value is significantly smaller than the reference value of *0.1*, the weights and the matrix itself are trustworthy and consistent [48]. In turn, it means that there are no mistakes made during the calculation process and that the resulting matrix accurately represents the given input.

The resulting criteria weights can then be used in later calculations. Each criteria weight shows how important it is compared to others in the evaluated model by assigning a criteria weight multiplier to each identified control.

3.1.6 Software errors taxonomy selection, applicable software flaws identification

It was decided to use the Seven Pernicious Kingdoms taxonomy due to it being more comprehensive for this study – it contains the environment kingdom, which is deemed important in the organization the model will be used in (Swedbank Group). Due to its comprehensive insight and the coverage of the operational environments, it is the most suitable for the Swedbank Group. This taxonomy usage was agreed upon with the reference group used for this master thesis.

In order to identify which software errors to cover, the author has used Fortify Taxonomy webpage [39] and has selected the most common software flaws applicable to a majority of languages. The Fortify taxonomy is a continuation of the Seven Pernicious Kingdoms taxonomy that contains a more comprehensive list of software flaws divided into the same kingdoms. This generalized set contains 32 software flaws, where some environment flaws were added manually to better reflect Swedbank Group concerns (such as disgruntled employee inject, which was not initially present). The result was then validated against the reference group as suitable to continue with and the one that brings value to the company. The complete list is presented in Table 5.

Kingdom	Software Security Flaw
Input Validation and Representation	SQL injection
	Command injection
	Cross-Site Scription (XSS)
	Dynamic Code Evaluation (unsafe deserialization and script injection)
	Denial of Service
API Abuse	Often misused
	Code Correctness
Security Features	Access Control
	Cookie Security
	Insecure Transport
	Key Management
	Password Management
	Privacy violation
	Privilege management
	Weak cryptography
Time and State	Race condition
	J2EE Bad Practices
Errors	Poor error handling
Code Quality	Code correctness
	Dead code
	Poor style
	Unreleased resource
Encapsulation	HTML5
	Insecure storage
	System information leak
*Environment	Build misconfiguration
	Dockerfile misconfiguration
	Target environment misconfiguration
	Disgruntled employee inject
	Insecure deployment
	Usage of third-party components
	Test environment data leakage

Table 5. Seven Pernicious Kingdoms selected software security flaws

Flaws in bold font are the ones added manually, and there are three of them – disgruntled employee inject, usage of third-party components, and test environment data leakage.

Disgruntled employee inject flaw indicates the ability of a developer or operations team member to inject malicious code into production in a way that other security controls such as peer review will deem to be ineffective.

Usage of third-party components combines two weaknesses – these components being vulnerable and with unsuitable licenses. This might result in system compromise or a penalty paid to the license owner.

Test environment data leakage is a specific threat of leaking data from test environments in cases where this data has some value (for example, is anonymized

production set). This particular flaw is not applicable to the company I work in but is still considered.

All other descriptions are taken from Fortify Taxonomy the way they are explained [39].

3.1.7 Ranking and criteria weights estimation of suitable controls

The security controls and processes are previously agreed upon and are known in advance [35]. Also, their pipeline position does not change and is fixed – this way, Threat Modelling is performed in the planning phase, at least for DevSecOps methodology [8]. It was proposed to the experts and later agreed upon to rank controls pipeline position and assign values gradually starting from *60* to *10*, depending on the pipeline position [55]. All intermediate values are multiples of *10*. This way, the closer control is to the left side (planning phase), the larger the value gets, as seen in Table 6.

Pipeline phase	Plan	Develop	Build & Test	Release & Test	Deploy	Operate & Monitor
Control pipeline position value	60	50	40	30	20	10

Table 6. Control pipeline position value for AHP calculation

Similarly, implementation duration and measurability criteria do not depend on a particular software flaw addressed. Their values are the same for all flaws. Implementation duration criteria got three grades – either *30*, *60,* or *100*. The idea to grade them this way was to separate control values as much as possible within the scale of *100* – this way, the distance between controls in the scale is *30 (30-0)*, *30 (60-30)*, and *40 (100-60)*. The less time it takes to perform the security control or process, the higher the value is. Since results are quantified, it will affect computation in a way that controls with faster implementation will be chosen among slower ones. This decision was taken and agreed upon beforehand within the experts' team to separate controls into distinct groups and not to use a continuous scale for these. This way, all slow manual controls will get significantly fewer points than automated ones in the implementation time. This is also justified if looking at the time to perform a manual task such as a security review – it might take up to a week. Technical control such as SAST scanning in the Swedbank Group environment will not take longer than a couple of hours for a single project. If to measure implementation time directly, the difference scale would be inefficient as one would need to know on an average – how much time does the scanning take and how much time regular security review takes. These values might differ significantly depending on the project; therefore, it makes more sense to grade such activities within scale groups to separate them.

Measurability grading is multiple of *25* and has three values – either *0*, *25,* or *50*. It was decided to try this scaling with three groups with the expert team, each control group being at the same distance of *25* value points from each other. Controls that

have the best measurability got a grade of *50,* and the ones with bad or no measurability got a grade of *0*. Measurability got a lower grade value since criterion weight is the lowest, and in practice, security controls reporting is not that well-established in the Swedbank Group to assign high values, to begin with.

Implementation duration and measurability were calculated using the same principle as for the AHP – all members of the reference group assigned their values to each control in the list for these two criteria, then the geometric mean was taken and rounded to the nearest possible value agreed upon.

Control effectiveness had to be calculated separately for each given software flaw. This process took the most time since 32 flaws were considered. Security controls effectiveness was estimated in values from *0* to *100* with a step of *10*. If the control was not effective at all, it was given a value of *0,* and if the control had complete effectiveness against a given type of flaw – it was assigned a value of *100*. After calculating the geometric mean and rounding to the nearest allowed value from *0* to *100*, the result was recorded.

In total, 122 security controls and processes were evaluated against 32 software flaws. For each software flow, the control weighted sum average value was calculated, and the top 20 controls were selected, starting from the highest value and moving to the lowest [56].

3.1.8 Creation of applicable SDLC architecture with a list of security controls and processes

This step's goal is to map and highlight the connection between all controls and phases in the SDLC methodology. Since controls are known, the main task is to interconnect them in a way this is aligned with Swedbank Group's development methodology and tooling possibilities. The resulting (S)SDLC reference architecture with all possible controls is located in the Github repository because it is too big to be inserted into the thesis document [57].

Operate & Monitor phase shows a separate circle of Operational Blueprint – this indicates that particular security controls might vary based on the operational environment and need to be considered separately for each. It made sense to pull environment controls into a separate area. For example, container orchestrators might have image immutability protection which terminates containers that have changed in time, but serverless deployments do not need this feature nor have it.

The controls and processes in the architecture are connected in a logical manner across different phases of the DevSecOps methodology.

3.1.9 Calculation of most important controls, validation of architectures

This meta-architecture step is to use calculated top 20 controls for 32 software flaws addressed, estimate their mathematical mode – rate based on occurrence across all software flaws, and then determine the top controls that need to be present in the final architecture and highlight them in red color in that architecture. These will be the ones ranked top 20 across all software flaws.

Since the goal of an enterprise is to select a necessary number of controls and due to the nature of presented software flaws, cumulative scoring is introduced along with mathematical mode – e.g., for each software flaw that appears more than once, results in individual scores are added together, and the final ranking table needs to be considered while looking into top 20 controls overall.

Once the architecture is complete, it needs to be validated against a peer group of experts – whether it makes sense to rank controls this way and what could be done differently. If the produced architecture is valid against selected software flaws, it will also mean that meta-architecture and methods used within meta-architecture are also validated, thus making it verified for use.

4 Results

This section describes calculation and ranking results for all software flaws and controls. It also provides a final ranking table and final Secure SDLC architecture for Swedbank Group use, given top 20 controls overall need to be selected against the top 32 identified software flaws (both taxonomy-based and three manually added).

4.1 Calculating results for individual flaws

The calculation results for all 32 software flaws and top 20 security controls are located in the Github repository [56]. For an example of SQL injection, the following top 20 controls were calculated as shown in Table 7.

Suitable controls	Score
Shared Security Patterns & Libraries	75,59
Security Education	73,29
IAST (Interactive Application Security Testing)	67,16
SAST (Static Application Security Testing)	66,13
IDE SAST (Integrated Development Environment SAST)	65,11
DAST (Dynamic Application Security Testing)	64,86
Input Validation	60,96
Application Security Risk Matrix	60,76
Regression test	60,25
SQL Stored Procedures	60,01
Database Firewall and Activity Monitoring	60,01
Vulnerability testing	57,95
Continuous Security Verification	57,71
RASP (Runtime Application Self-Protection)	57,71
Logging and Security Monitoring	55,40
Workloads Micro-segmentation	55,40
Data Leakage Prevention	55,40
Web Application Firewall	55,40
TLS Interception	55,40
Canary Breach Detection	55,40

Table 7. SQL injection top 20 security controls

As we can see from the list, due to pipeline position criteria weight, both security education and shared security pattern & libraries are selected as the top two controls. This is due to the fact that shared patterns & libraries are already controlled and are good to use since they do not contain any known software flaws at the time of usage. It is indeed much better to share pieces of code among multiple teams and control them once. Security education as an administrative type of control is also very important – it happens even prior to the design and plan phase and is indeed considered one of the most important controls there could be in any development process [58].

The main idea of having top controls and a list to select from is to provide alternatives with different values. Some controls are considered to be best based on their ranking, but there are alternatives available, and the end result is not mandated to be strictly followed. The values overall show rankings in the given methodology.

The following controls are technical – IAST provides white box testing in a test environment and is likely to find SQL injection flaws. It is usually integrated as a module inside of the application; therefore, it can make tests against all available interfaces. SAST tooling checks source code in the previous phase and is likely to find the same errors just as well.

There are specific types of controls as a defense against SQL injection, such as using stored procedures in the database. It is justified to appear in this table, but its overall value while looking at the entire set of 32 software flaws is insignificant since it is specific and only protects against SQL injections.

Production or Operate & Monitor phase controls [35] are still very effective, but not as much compared to the top five controls. It is also notable that if such an event as SQL injection happened in production, the cost of mistake would be much higher; therefore, it is still preferred to have multiple control coverage against a single security flaw and prefer controls from the initial phases of SDLC.

For specific environment types of flaws, looking at the disgruntled employee inject and its controls as shown in Table 8.

Suitable controls	Score
Configuration compliance	67,16
SCA (Software Composition Analysis)	66,13
SAST (Static Application Security Testing)	63,83
Attack Surface Minimization	63,06
Vulnerability testing	62,56
Separation of Duties	60,75
Complete Mediation	60,75
Workloads Micro-segmentation	60,01
Privileged Session Management	57,71
Supplier Chain Management	57,18
Job interview & Hiring Strategy	55,70
Compliance & Policy Checks	55,40
Continuous Security Verification	55,40
Anomaly Detection	55,40
Peer Review	53,60
Data Leakage Prevention	53,10
Canary Breach Detection	53,10
TLS Interception	53,10
Database Firewall and Activity Monitoring	53,10
Image Admission Control	52,21

Table 8. Disgruntled employee inject security controls

Control distribution also seems logical. Overall scoring still shifts towards the compliant configuration of the target environment (top control in the list), using software composition analysis to prevent malicious injects of vulnerable libraries, having use of such concepts as vulnerability testing, separation of duties (in the production environment), complete mediation (to prevent sessions mismanagement) and even taking into account supplier chain management issues along with privileged session management and so on. The difference in scoring value between the top ten controls does not exceed ten points, making them relatively close in a matter of choice against particular software flaws and threats.

Two security architects raised the topic that this type of scenario, such as disgruntled employee inject might be done differently, therefore, some controls might prevail over others. For example, if the inject is done through a malicious third-party library that comes from an unscanned repository, the image admission control would probably be more effective than data leakage prevention in case the ultimate goal was not to exfiltrate the data. At the same time, the group agreed that the vast majority of controls that would apply in the general scenario are present in Table 8. It might be argued in which particular order could they be arranged – depending on the scenario; still, all controls seem very reasonable.

One offensive security officer noted job interview and hiring strategy being on this list and questioned whether this would be effective control against a disgruntled employee. After discussing the nature of this control with the group, we came to a consensus that during the hiring process within our company, employee background and criminal records are thoroughly checked; therefore, this control is effective in case of similar events taking place in the past during the employment in other companies. Therefore, having this control in the list is also justified, despite it being of administrative nature.

4.2 Final table with mathematical mode and cumulative scoring for top 20 controls

The final table with all controls, their cumulative scoring, and mathematical mode for the top 20 controls (count of appearance across all 32 software flaws) is located in the Github repository [56]. Out of 122 controls proposed, only 87 controls were used more than once. This is due to the fact that the remainder was not deemed effective to be highlighted during the interview section and filled in the table.

The table is sorted by controls total scoring value, starting from highest to lowest in the top 20, as shown in Table 9.

Suitable controls	Total score	Mathematical mode
Security Education	2153,01	29
Shared Security Patterns & Libraries	1963,07	26
SAST (Static Application Security Testing)	1687,88	25
Coding Best Practices	1552,46	28
Peer Review	1539,91	30
CI/CD pipeline design	1441,57	31
IDE SAST (Integrated Development Environment SAST)	1410,97	21
Continuous Security Verification	1281,03	23
Penetration testing	1227,13	30
Application Security Checklists	1225,50	22
Learning from Mistakes	1224,53	27
Pair Programming	1097,53	20
Vulnerability testing	1090,68	18
DAST (Dynamic Application Security Testing)	1061,17	17
Security review	1045,73	29
IAST (Interactive Application Security Testing)	1019,33	16
Configuration compliance	1007,00	16
Application Security Risk Matrix	985,04	17
Red Teaming	980,95	30
RASP (Runtime Application Self-Protection)	916,39	16

Table 9. Top 20 security controls with mathematical mode

It turns out that the security education is deemed to be the most important control if taking all security flaws into account – it appears 29 times. In other words, it is effective against 29 types of security flaws out of 32. The second most effective control is shared patterns & libraries. Overall controls ranking looked correct and was validated together with a team of experts. Initially, one security architect and one offensive security (security verification) officer explained their concerns about the second top control in this list, but after careful consideration and understanding of the value of this control, no other objections were raised.

Cumulative effectiveness differs more than two times between the first and the last control in the table. Figure 17 shows all controls with their total score (cumulative values).

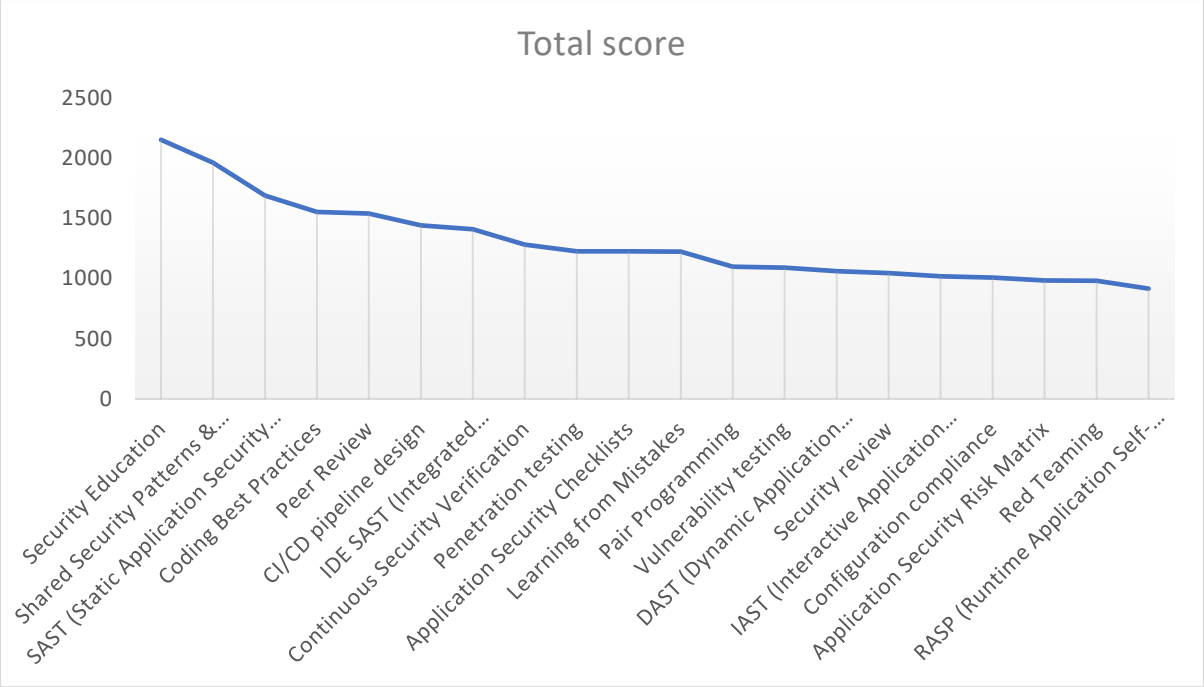

Figure 17. Top 20 security controls chart

The second way to understand this table is to calculate the average score per appearance – divide the cumulative score with a mathematical mode for each control. Figure 18 shows this representation and reorder.

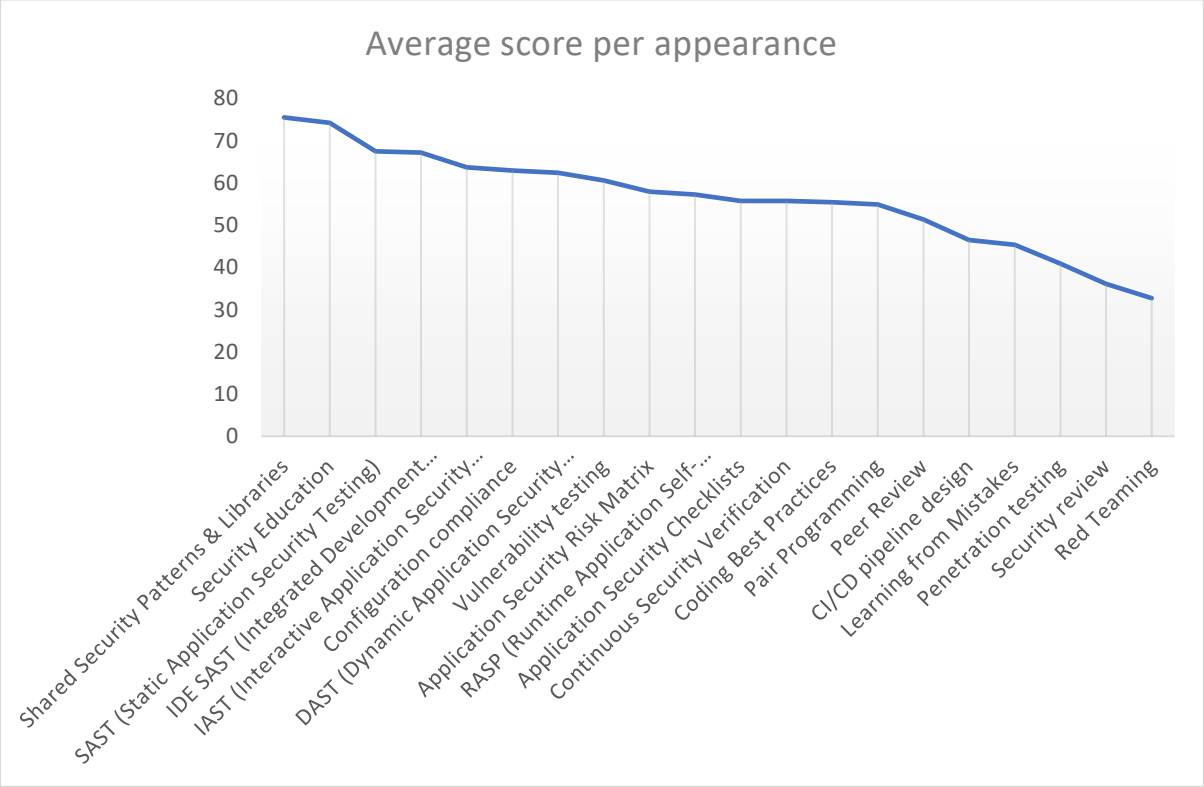

Figure 18. Top 20 security controls reordered based on their rate of appearance

Such reordering tends to place manual controls towards the end of the graph with lower values and brings automated controls closer to the middle. This, in turn, means that, on average, against a single software flaw – automated controls are preferred over manual controls, which is logical since the implementation duration criteria value is higher. Hense, RASP has moved from place 20 to place 10. At the same time, the top 5 controls do not change significantly.

This ranking might be more suitable for fully automated pipelines where time is important since the top 8 controls are fully automated.

4.3 Validation against other peers

Threat modelling as a security practice did not end up in the top 20; its overall position is top 22. The experts' group opinion was that it would be included in the top 20 under the condition that environment kingdom software flaws and threats would be considered during the threat modelling process. Usually, threat modelling does not take the operational environment into account; therefore, it was not deemed effective against environment kingdom software flaws and threats. If to validate this result against common, well-known reference examples by either Schleen or Microsoft [2][59], the majority of controls overlap. This means that control selection is justified, and at the same time, validation from expert reference group together with their input means that this particular selection is better adapted to the selected scenario and enterprise. Also, the Microsoft example does not rank controls based on their importance though DevSecOps assumes that this ranking has already taken place prior (since controls in the early phases impact the price of mistake correction).

Simplified Microsoft DevSecOps controls architecture is presented in Figure 19.

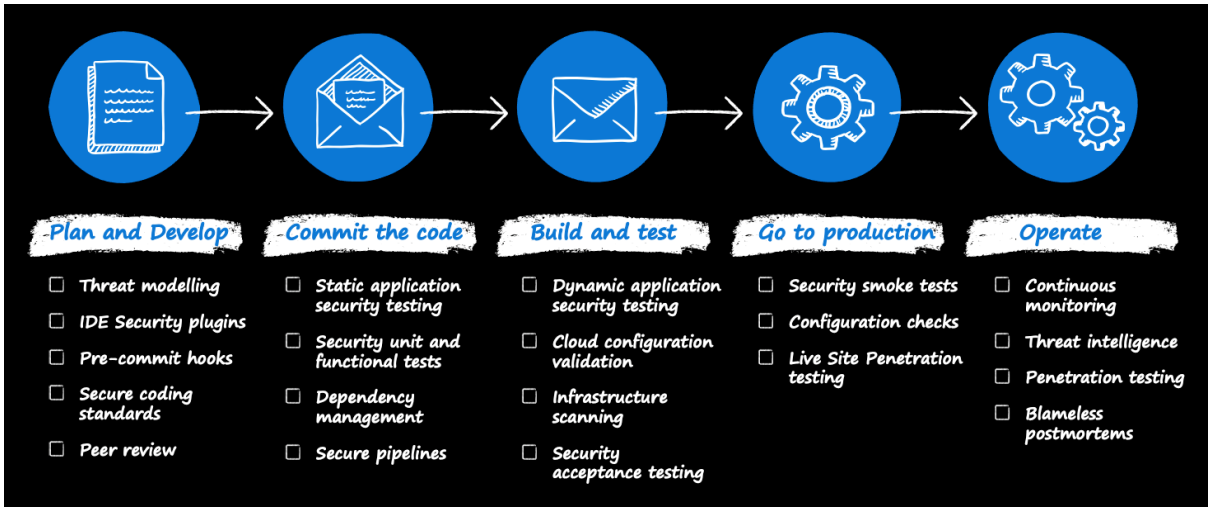

Figure 19. Microsoft DevSecOps controls architecture [59]

Though some control names might be confusing since they are named differently than in the thesis example, the reference table is presented below in Table 10.

Top 20 Microsoft DevSecOps controls	**Top 20 thesis DevSecOps control name**	**Top 20 thesis DevSecOps control place**
Threat modelling	Threat modelling	22
IDE Security plugins	**IDE SAST**	**7**
Pre-commit hooks	Pre-commit hooks	29
Secure coding standards	**Coding best practices**	**4**
Peer review	**Peer review**	**5**
SAST	**SAST**	**3**
Security unit and functional tests	Unit tests, Functional testing	21, 79
Dependency management	Supplier Chain Management, SCA	36, 42
DAST	**DAST**	**14**
Cloud configuration validation	**Configuration compliance**	**17**
Infrastructure scanning	**Vulnerability testing, top 13**	**13**
Security acceptance testing	**Security review, top 15**	**15**
Security smoke tests	-	-
Configuration checks	**Configuration compliance**	**17**
Live Site Penetration testing	**Penetration testing**	**9**
Continuous monitoring	Logging and security monitoring	38
Threat intelligence	Threat & Risk Landscape	Not scored
Penetration testing	**Continuous security verification**	**8**
Blameless postmortems	**Learning from mistakes**	**11**

Table 10. Top 20 security controls comparison

Overall, there are 12 out of 20 controls that overlap or 60%. If controls selection is to be expanded to cover the top 22 controls, it will also cover Threat modelling and Security unit testing, which will give 14 out of 20, or 70%.

The main reason why controls did not match up to 100% is that in the security flaws selected, there are ones that apply to the Environment Kingdom, some of them being not related to the development process, but rather to operation. For example, threat modelling was not deemed effective against disgruntled employee inject since it normally does not cover platform administration means and privileges access issues. Therefore, its value has decreased, and it ended up being in the 22 place.

The threat intelligence control proposed by Microsoft is technical – its operation most likely focuses on sharing indicators of compromise and scanning for their presence in the deployment environment. The counterpart threat & risk landscape control is an administrative control whose main aim is to supply risk management process with increased threat values for given threat actors based on annual reports by the industry.

Software composition analysis is indeed important and would have been found in the top 20 list if there had been more software flaws related to vulnerable third-party libraries.

These are good results compared to the reference given by the vendor; although Microsoft's method of controls identification is not revealed, it is stated that this is a vendor recommendation [59].

4.4 DevSecOps architecture with top 20 identified controls

The resulting architecture with highlighted top 20 controls is presented in the Github repository due to its size [60].

Security controls distribution over different phases is shown in the following table 11:

Plan	Develop	Build & Test	Release & Test	Deploy	Operate & Monitor
4	5	2	6	0	3

Table 11. Security controls distribution across different phases of DevSecOps

It is clear that controls in the earlier phases of the model prevail – there are nine controls out of 20 in the first two phases and 17 out of 20 in the first 4 phases. Since early involvement of security controls was an important criterion, this table reflects the result.

5 Discussions

This study dealt with the certain literature gap on covering the subject of choosing suitable application security controls based on organizational needs, different methodologies in use, and their criteria of choice.

The results showed that reading literature and methodologies principles allows to figure out criteria, and later usage of AHP method can rank these criteria so that organizational needs are covered in the best way. The chosen security controls are well in line with industry best practices, but their method of selection is validated together with peers and has a scientific basis.

The selection of the top 20 controls is reasonable, but it can be either expanded or downsized. Controls number depends on a budget, risk appetite, and other factors which derive from the organization's needs. The control list is applicable for software flaws selection and subjective ranking of their effectiveness in a given organization. It can be tailored for a specific coding language by following the same method proposed. The number of resulting controls, their ranking, and effectiveness will then most likely change, but the result can be used to build a specific pipeline for a specific language and organization. Not all organizations can afford to have ten controls in a row effective against each proposed software error, but it is possible to select the top 5, top 10, top 20, or top 50 using this method. By knowing risk appetite and calculating software error coverage by all controls in a pipeline, it is possible to know what the residual pipeline risk is and which software errors need more attention or more effective controls.

The entire method resulted in the creation of (S)SDLC meta-architecture (Secure SDLC) can be used to create SDLC architectures for DevSecOps based on the organization's needs. It also can be used to create architectures for different methodologies and phases and with different security controls. Though it has not been validated yet, it would be interesting to see the end result of an architecture for a Waterfall methodology using OWASP taxonomy and how it fits into industry best practice, if there is any.

Since the meta-model was used to derive the reference model for DevSecOps and reference architecture itself got validated, at least for DevSecOps methodology - this also means that the meta-model is correct since all attributes are derived directly.

This, in turn, answers research questions. The first question was related to the used controls pool, but they are still shared across different software development methodologies – they just have different values based on the criteria chosen.

This thesis could have been extended with additional research questions by bringing price perspective into account – since different controls might have different price tags. Some solutions that perform SCA (Software Composition Analysis) are open source, while some might be pretty expensive for a small enterprise. Bringing the cost of mistakes into the equation might result in a next thesis – for example, by calculating pipeline security controls that would make the pipeline "profitable "– by extrapolating the ideal assumption that sufficient control coverage can be effective

against selected types of software security flaws. Various risk management and risk calculation ideas might also grow out of this study.

However, it is outside of the study goal and scope, and therefore, not addressed.

6 Summary

In this thesis, the author has created a Secure SDLC meta-architecture model that was able to produce successful DevSecOps SDLC architecture with security controls that are based on methodology criteria and chosen software errors taxonomy.

The validation has shown that control selection is reasonable and that proposed architectures make sense to be used in the production to suit organization needs.

The study has shown that usage of AHP can indeed be expanded on the application security domain, which it previously did not consider.

While control estimation and calculation tasks take much time – the end result is still thriving. Most probably, such a method could be improved using automation and suggesting particular controls and configurations for a specific software model or pipeline.

References

[1] United States of America Department of Defense. (2021). DevSecOps Fundamentals Guidebook: DevSecOps Tools & Activities, Version 2.1. https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOps%20Fundamentals%20Guidebook-DevSecOps%20Tools%20and%20Activities_DoD-CIO_20211019.pdf (accessed 02.05.2022)

[2] DJ Schleen. (2019). Interactive DevSecOps Reference Architecture. https://www.alldaydevops.com/blog/interactive-devsecops-reference-architecture (accessed 02.05.2022)

[3] Mojtaba Shanin, Muhammad Ali Babar, Liming Zhu. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access Volume 5, 22.03.2017, pp 3909-3943.

[4] Runfeng Mao, He Zhang, Qiming Dai, Huang Huang, Guoping Rong, Haifeng Shen, Lianping Chen, Kaixiang Lu. (2020). Preliminary Findings about DevSecOps from Grey Literature. 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), Macau, China.

[5] Nor Shahriza Abdul Karim, Arwa Albuolayan, Tanzila Saba, Amjad Rehman. (2016). The practice of secure software development in SDLC: an investigation through existing model and a case study. https://onlinelibrary.wiley.com/doi/full/10.1002/sec.1700 (Accessed 02.05.2022)

[6] Anna Koskinen. (2020). Master Thesis: DEVSECOPS: BUILDING SECURITY INTO THE CORE OF DEVOPS. University of Jyväskylä, 2019. https://jyx.jyu.fi/handle/123456789/67345 (Accessed 02.05.2022)

[7] Mansfield-Devine, S. (2018). DevOps: Finding Room for Security. Network Security, 2018(7), pp. 15−20.

[8] United States of America Department of Defense, Chief Information Officer. (2021). DoD Enterprise DevSecOps Reference Design, Version 1.0, 12.08.2019. https://dodcio.defense.gov/Portals/0/Documents/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20v1.0_Public%20Release.pdf?ver=2019-09-26-115824-583 (accessed 02.05.2022)

[9] Katrina Tsipenyuk, Brian Chess, Gary McGraw. (2005). Seven pernicious kingdoms: a taxonomy of software security errors. IEEE Security & Privacy (Volume: 3, Issue: 6, Nov.-Dec. 2005), pp 81-84

[10] Roseanna W Saaty. "The analytic hierarchy process—what it is and how it is used". In: Mathematical modelling 9.3-5 (1987), pp. 161–176.

[11] Jose Andre Morales, Hasan Yasar, Aaron Volkman. 2018. Implementing DevOps Practices in Highly Regulated Environments. In Proceedings of International Workshop on Secure Software Engineering in DevOps and Agile Development (XP '18 Companion). ACM, New York, NY, USA, Article 4, 9 pages.

[12] Jose Andre Morales, Thomas P. Scanlon, Aaron M. Volkmann, Joseph D. Yankel and Hasan Yasar. 2020. Security Impacts of Sub-Optimal DevSecOps Implementations in a Highly Regulated Environment. In Proceedings of the ACM ARES 2020, August 25–28, 2020, Virtual Event, Ireland.

[13] Thorsten Rangnau, Remco v. Buijtenen, Frank Fransen, Fatih Turkmen. (2020). Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC), Eindhoven, Netherlands.

[14] Vinay M. Igure, Ronald D. Williams. (2008). Taxonomies of attacks and vulnerabilities in computer systems. IEEE Communications Surveys & Tutorials (Volume: 10, Issue: 1, First Quarter 2008), pp 6-19.

[15] United States of America Department of Defense. (2021). DoD Enterprise DevSecOps Reference Design: CNCF Kubernetes, Version 2.1. https://dodcio.defense.gov/Portals/0/Documents/Library/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20-%20CNCF%20Kubernetes%20w-DD1910_cleared_20211022.pdf (accessed 02.05.2022)

[16] United States of America Department of Defense. (2021). DoD Enterprise DevSecOps Strategy Guide, Version 2.0. https://dodcio.defense.gov/Portals/0/Documents/Library/DoD%20Enterprise%20DevSecOps%20Strategy%20Guide_DoD-CIO_20211019.pdf (accessed 02.05.2022)

[17] Havard Myrbakken, Ricardo Colomo-Palacios. (2017). DevSecOps: A Multivocal Literature Review. International Conference on Software Process Improvement and Capability Determination. Palma de Mallorca, Spain, October 4–5, 2017.

[18] Laurie Williams, Akond Ashfaque Ur Rahman. (19.04.2016). Security Practices in DevOps. Department of Computer Science, North Carolina State University, Raleigh, NC, USA. HotSos '16: Proceedings of the Symposium and Bootcamp on the Science of Security, pp 109-111.

[19] Yashod.R. (2021). Overview of System Development Life Cycle Models. SSRN Electronic Journal. 11(1), pp 12-22.

[20] Cois, C. A., Yankel, J. & Connell, A. (2014). Modern DevOps: Optimizing Software Development Through Effective System Interactions. 2014 IEEE International Professional Communication Conference, IPCC (1−7). Pittsburgh, PA, USA, 2014.

[21] Nayan B. Ruparelia. (05.2010). Software Development Lifecycle Models. ACM SIGSOFT Software Engineering Notes 35(3). pp 8-13

[22] Denmark Technical University. Waterfall model. http://apppm.man.dtu.dk/index.php/Waterfall_model (Accessed 02.05.2022)

[23] Mahdi H. Miraz, Maaruf Ali. (01.2020). Blockchain Enabled Smart Contract Based Applications: Deficiencies with the Software Development Life Cycle Models. Baltica 33(1). pp 101-116

[24] Ediz Şaykol. (10.2012). An Economic Analysis of Software Development Process based on Cost Models. INTERNATIONAL CONFERENCE ON EURASIAN ECONOMIES 2012.

[25] Asif Irshad Khan, Rizwan Jameel Qurashi and Usman Ali Khan. (07.2011). A Comprehensive Study of Commonly Practiced Heavy & Light Weight Software Methodologies. IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 4, No 1, July 2011.

https://arxiv.org/ftp/arxiv/papers/1202/1202.2514.pdf (Accessed 02.05.2022).

[26] Agile Manifesto. (2001). Agile Manifesto, https://agilemanifesto.org/ (accessed 02.05.2022)

[27] Asma Aziz. (04.2012). Brief comparison of SDLC models. https://www.researchgate.net/publication/271834029_Brief_comparison_of_SDLC_models (Accessed 02.05.2022)

[28] Putu Adi Guna Permana. (09.2015). Scrum Method Implementation in a Software Development Project Management. International Journal of Advanced Computer Science and Applications 6(9).

[29] Ur Rahman, A. A. and Williams, L. (2016). Software Security in DevOps: Synthesizing Practitioners' Perceptions and Practices. 2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery, CSED (70−76). Austin, TX, USA.

[30] Yuri Bobbert. (01.2021). Problems of CI/CD and DevOps on Security Compliance. In book: Strategic Approaches to Digital Platform Security Assurance (pp.256-285)

[31] Thomas Zühlke, Arvato Systems. (20.09.2018). Die Time-To-Market liegt zwischen zwei Tassen Kaffee. https://www.arvato-systems.de/blog/schnelle-time-to-market-mit-devops (Accessed 02.05.2022), publication is in German.

[32] Fabiola Moyón, Rafael Soares, Maria Pinto Albuquerque and Daniel Méndez Fernández. (05.2021). Integration of Security Standards in DevOps Pipelines: An Industry Case Study. https://www.researchgate.net/publication/351929062_Integration_of_Security_Standards_in_DevOps_Pipelines_An_Industry_Case_Study (Accessed 02.05.2022)

[33] Muhammad Asad, Shafique Ahmed. (06.2016). Model Driven Architecture for Secure Software Development Life Cycle. International Journal of Computer Science and Information Security, 14(6).

[34] Ngethe Simon Ngug, Tumuti Joshua. (05.2021). INFORMATION SYSTEM SECURITY CONTROLS AND DATA SECURITY IN UNIVERSITIES IN KENYA. A CASE OF KIRIRI WOMEN'S UNIVERSITY OF SCIENCE AND TECHNOLOGY. IARJSET 8(5)

[35] Michailas Ornovskis (author). Security Controls and Processes in private Github repository. https://github.com/mikkyornyx/SDLC/blob/main/README.md (Accessed 02.05.2022).

[36] OWASP project vulnerabilities taxonomy. https://owasp.org/www-community/vulnerabilities/ (Accessed 02.05.2022)

[37] Khairul Anwar Sedek, Osman Norlis, Mohd NIZAM Osman, Kamaruzaman Jusoff. (10.2009). Developing a Secure Web Application Using OWASP Guidelines. Computer and Information Science 2(4).

[38] Mitre CWE taxonomy. https://cwe.mitre.org/data/definitions/699.html (Accessed 02.05.2022).

[39] Fortify Taxonomy: Software Security Errors. https://vulncat.fortify.com/en/weakness (Accessed 02.05.2022).

[40] Kadri Cahani master thesis. (2020). Aligning Information Security Risks with Strategic Goals. Tallinn University of Technology. https://digikogu.taltech.ee/et/Download/b8ab352f-f2b2-4964-882a-0b7b5be986a9 (Accessed 02.05.2022).

[41] Pedro Mota, Ana Rita Campos, and Rui Neves-Silva. "First look at MCDM: Choosing a decision method". In: Advances in Smart Systems Research 3.1 (2012), p. 25.

[42] Evangelos Triantaphyllou. "Multi-criteria decision making methods". In: Multicriteria decision making methods: A comparative study. Springer, 2000, pp. 5–21.

[43] Muhammad Imran Tariq, Shakeel Ahmed, Nisar Ahmed Memon, Engr. Dr. Shahzadi Tayyaba. (02.2020). Prioritization of Information Security Controls through Fuzzy AHP for Cloud Computing Networks and Wireless Sensor Networks. Sensors 20(5):1-39.

[44] Željko Stević, Ilija Tanackov, Ilija Cosic, Slavko Vesković. (09.2015). COMPARISON OF AHP AND FUZZY AHP FOR EVALUATING WEIGHT OF CRITERIA. Conference: V International Symposium New HorizonsAt: Doboj, Bosnia and Herzegovina. (In Croatian). https://www.researchgate.net/publication/310618533_COMPARISON_OF_AHP_AND_FUZZY_AHP_FOR_EVALUATING_WEIGHT_OF_CRITERIA (Accessed 02.05.2022).

[45] Golam Kabir. (01.2011). Fuzzy AHP for Contractor Evaluation in Project Management-A Case Study. Int. Jour. Of Business & Inf. Tech. Vol-1 No. 1 March 2011. pp 85-96.

[46] Eddie WL Cheng and Heng Li. "Information priority-setting for better resource allocation using analytic hierarchy process (AHP)". In: Information Management & Computer Security (2001).

[47] Valentinas Podvezko. (06.2009). Application of AHP technique. Journal of Business Economics and Management 10(2):181-189.

[48] R.W.Saaty. (1987). The Analytic Hierarchy Process – What It Is and How It Is Used. Mathematical Modelling 9(3-5):161-176.

[49] Petar Markovic, Dejan R Stevanovic, Milica Pesic-Georgiadis, Mirjana Banković. (01.2021). Application of MCDA in the determination of optimal block size for open-pit modelling and mine planning. Podzemni Radovi 2021(38):67-85.

[50] Edie Ezwan Mohd Safian, Abdul Hadi Nawawi. (01.2011). The Evolution of Analytical Hierarchy Process (AHP) as a Decision Making Tool in Property Sectors. https://www.researchgate.net/publication/254445031_The_Evolution_of_Analytical_Hierarchy_Process_AHP_as_a_Decision_Making_Tool_in_Property_Sectors (Accessed 02.05.2022).

[51] Mubarak Mohammad. (09.2008). TADL - An Architecture Description Language for Trustworthy Component-Based Systems. Conference: Software Architecture, Second European Conference, ECSA 2008, Paphos, Cyprus, September 29 - October 1, 2008, Proceedings. https://www.researchgate.net/publication/220757055_TADL_-_An_Architecture_Description_Language_for_Trustworthy_Component-Based_Systems (Accessed 02.05.2022).

[52] United States Department of Defense DevSecOps Reference Design v1.6 Maturity Review document. https://software.af.mil/wp-content/uploads/2019/12/DoD-Enterprise-DevSecOps-Maturity-Review-v1.6.docx (Accessed 02.05.2022).

[53] OWASP DevSecOps Maturity Model DSOMM. https://owasp.org/www-project-devsecops-maturity-model/ (Accessed 02.05.2022).

[54] Aws Magableh, Anas Alsobeh. (08.2018). Aspect-Oriented Software Security Development Life Cycle (AOSSDLC). Conference: 5th International Conference on Computer Science and Engineering (CSEN-2018) August 25 ~ 26, 2018, Dubai, UAE Volume Editors: Dhinaharan Nagamalai, Jan Zizka ISBN: 978-1-921987-90-8At: Dubai

[55] Michailas Ornovskis (author). Security Controls calculation, All flaws Tab. https://github.com/mikkyornyx/SDLC/blob/main/control_PoC.xlsx (Accessed 02.05.2022).

[56] Michailas Ornovskis (author). Security Controls calculation, All Results Top 20 Tab. https://github.com/mikkyornyx/SDLC/blob/main/control_PoC.xlsx (Accessed 02.05.2022).

[57] Michailas Ornovskis (author). SDLC DevSecOps Architecture with all controls and processes. https://github.com/mikkyornyx/SDLC/blob/main/SDLC_arch.jpg (Accessed 02.05.2022).

[58] Snyk. Secure Software Development Lifecycle (SSDLC). https://snyk.io/learn/secure-sdlc/ (Accessed 02.05.2022).

[59] Microsoft Cloud Adoption Framework DevSecOps controls. (2021). https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/secure/devsecops-controls (Accessed 02.05.2022).

[60] Michailas Ornovskis (author). SDLC DevSecOps Architecture with top 20 controls and processes highlighted. https://github.com/mikkyornyx/SDLC/blob/main/top_20_controls.jpg (Accessed 02.05.2022).

[61] Megan Kaczanowski. (05.04.2021). What is a Buffer Overflow Attack – and How to Stop it. https://www.freecodecamp.org/news/buffer-overflow-attacks/ (Accessed 02.05.2022).

[62] Tenendo. Secure Software Development Life Cycle (Secure SDLC) https://tenendo.com/secure-development/ (Accessed 02.05.2022).

[63] Richard Kissel, Kevin Stine, Matthew Scholl, Hart Rossman, Jim Fahlsing, Jessica Gulick. (October 2008). Security Considerations in the System Development Life Cycle. NIST Special Publication 800-64 Revision 2. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-64r2.pdf (Accessed 02.05.2022).

[64] Pavel Hrabě. Change of TOGAF structure and meta-model. (May 2012). Conference: 18th IBIMA Conference. https://www.researchgate.net/publication/259532982_Change_of_TOGAF_structure_and_metamodel (Accessed 02.05.2022).

[65] Mohamed Sami, (21.01.2020). "Architecture Model, Meta-Model, and Meta-Meta Model," in Mohamed Sami - Personal blog. https://melsatar.blog/2020/01/12/architecture-model-meta-model-and-meta-meta-model/ (Accessed 02.05.2022).