COMPUTER RESEARCH AND MODELING
2024 VOL. 16 NO. 7 P. 1687–1702
DOI: 10.20537/2076-7633-2024-16-7-1687-1702

Ки&М

SPECIAL ISSUE

UDC: 004.056

Enhancing DevSecOps with continuous security requirements analysis and testing

A. Sadovykh[1,a], **V. Ivanov**[2,b]

[1]SOFTEAM,
France
[2]Innopolis University,
1 Universitetskaya st., Innopolis, 420500, Russia

E-mail: [a] andrey.sadovykh@softeam.fr, [b] v.ivanov@innopolis.ru

Received 27.10.2024, after completion — 12.11.2024
Accepted for publication 25.11.2024

The fast-paced environment of DevSecOps requires integrating security at every stage of software development to ensure secure, compliant applications. Traditional methods of security testing, often performed late in the development cycle, are insufficient to address the unique challenges of continuous integration and continuous deployment (CI/CD) pipelines, particularly in complex, high-stakes sectors such as industrial automation. In this paper, we propose an approach that automates the analysis and testing of security requirements by embedding requirements verification into the CI/CD pipeline. Our method employs the ARQAN tool to map high-level security requirements to Security Technical Implementation Guides (STIGs) using semantic search, and RQCODE to formalize these requirements as code, providing testable and enforceable security guidelines. We implemented ARQAN and RQCODE within a CI/CD framework, integrating them with GitHub Actions for real-time security checks and automated compliance verification. Our approach supports established security standards like IEC 62443 and automates security assessment starting from the planning phase, enhancing the traceability and consistency of security practices throughout the pipeline. Evaluation of this approach in collaboration with an industrial automation company shows that it effectively covers critical security requirements, achieving automated compliance for 66.15 % of STIG guidelines relevant to the Windows 10 platform. Feedback from industry practitioners further underscores its practicality, as 85 % of security requirements mapped to concrete STIG recommendations, with 62 % of these requirements having matching testable implementations in RQCODE. This evaluation highlights the approach's potential to shift security validation earlier in the development process, contributing to a more resilient and secure DevSecOps lifecycle.

Keywords: cybersecurity, DevSecOps, DevOps, continuous integration, requirements, requirements engineering, tests, natural language processing, machine learning, SBERT, RQCODE, ARQAN, GITHUB

Citation: *Computer Research and Modeling*, 2024, vol. 16, no. 7, pp. 1687–1702.

**КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2024 Т. 16 № 7 С. 1687–1702**
DOI: 10.20537/2076-7633-2024-16-7-1687-1702

СПЕЦИАЛЬНЫЙ ВЫПУСК

УДК: 004.056

Улучшение DevSecOps с помощью непрерывного анализа и тестирования требований безопасности

А. Садовых[1,a], В. Иванов[2,b]

[1]СОФТИМ,
Франция
[2]Университет Иннополис,
Россия, 420500, г. Иннополис, ул. Университетская, д. 1

E-mail: [a] andrey.sadovykh@softeam.fr, [b] v.ivanov@innopolis.ru

Получено 27.10.2024, после доработки — 12.11.2024
Принято к публикации 25.11.2024

DevSecOps требует интеграции безопасности на каждом этапе разработки программного обеспечения для обеспечения безопасных и соответствующих требованиям приложений. Традиционные методы тестирования безопасности, часто выполняемые на поздних этапах разработки, недостаточны для решения задач, связанных с непрерывной интеграцией и непрерывной доставкой (CI/CD), особенно в сложных, критически важных секторах, таких как промышленная автоматизация. В данной статье мы предлагаем подход, который автоматизирует анализ и тестирование требований безопасности путем встраивания проверки требований в конвейер CI/CD. Наш метод использует инструмент ARQAN для сопоставления высокоуровневых требований безопасности с Руководствами по технической реализации безопасности (STIGs) с помощью семантического поиска, а также RQCODE для формализации этих требований в виде кода, предоставляя тестируемые и поддающиеся исполнению руководства по безопасности. Мы внедрили ARQAN и RQCODE в рамках CI/CD, интегрировав их с GitHub Actions для обеспечения проверки безопасности в реальном времени и автоматической проверки соответствия. Наш подход поддерживает стандарты безопасности, такие как IEC 62443, и автоматизирует оценку безопасности, начиная с этапа планирования, улучшая прослеживаемость и согласованность практик безопасности на протяжении всего конвейера. Предварительная оценка этого подхода в сотрудничестве с компанией по промышленной автоматизации показывает, что он эффективно охватывает критические требования безопасности, достигая автоматического соответствия 66,15 % руководств STIG, относящихся к платформе Windows 10. Обратная связь от отраслевых специалистов подчеркивает его практичность: 85 % требований безопасности сопоставлены с конкретными рекомендациями STIG, и 62 % из этих требований имеют соответствующие тестируемые реализации в RQCODE. Эта оценка подчеркивает потенциал подхода для сдвига проверки безопасности на более ранние этапы разработки, способствуя более устойчивому и безопасному жизненному циклу DevSecOps.

Ключевые слова: кибербезопасность, DevSecOps, DevOps, непрерывная интеграция, требования, требования к проектированию, тесты, обработка естественного языка, машинное обучение, SBERT, RQCODE, ARQAN, GITHUB

Introduction

DevSecOps is an important step forward in software development. It aims to include security practices in the fast-paced world of DevOps. DevSecOps builds on the collaborative nature of DevOps, which traditionally connects development and operations teams, by adding security as a shared responsibility from the beginning. This change is necessary because traditional security methods, often applied at the end of the development cycle, cannot keep up with the speed of DevOps [Myrbakken, Colomo-Palacios, 2017].

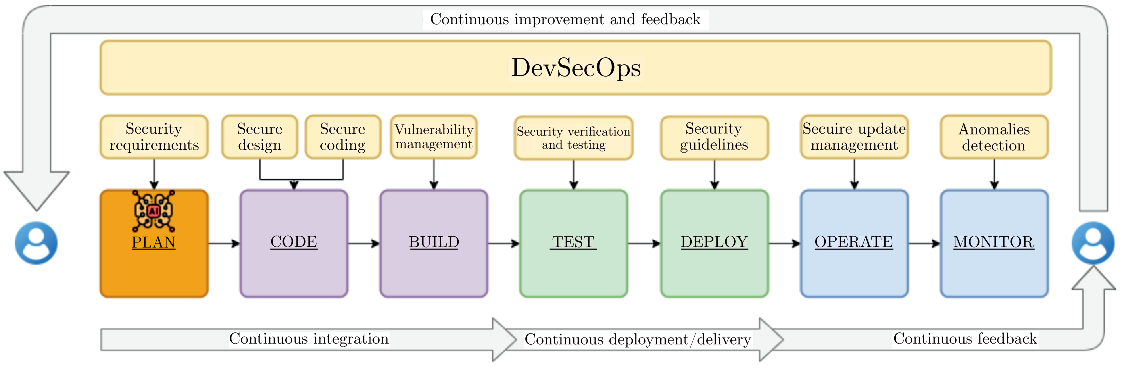


Figure 1. DevSecOps paradigm (adapted from [Moyón et al., 2020])

DevSecOps (Fig. 1) is a continuous cycle where security is integrated into every aspect of software development. The process can be understood through the "infinite loop" model, emphasizing its continuous and iterative nature [Moyón et al., 2020]. In the PLAN stage, the project scope is defined, requirements are gathered, and security considerations are established from the outset. This involves threat modeling, risk assessment, and defining security requirements as integral parts of user stories. Collaboration between security experts, developers, and operations teams is crucial to ensure security is embedded in the project's foundation. During the CODE stage, developers write a code while adhering to security best practices and incorporating security controls defined in the planning stage. This includes using secure coding libraries, implementing security features, and following security guidelines. Static code analysis tools are employed to identify potential vulnerabilities early in the development process. The Build stage involves compiling the code and creating a build artifact. Security considerations focus on ensuring the integrity and security of the build process, which includes using secure build environments, verifying the authenticity of dependencies, and scanning the build artifact for known vulnerabilities. In the TEST stage, the software undergoes rigorous testing to ensure functionality and identify security vulnerabilities. This includes static analysis, dynamic analysis, and penetration testing. Automated security testing tools are essential for maintaining the speed of DevSecOps, enabling rapid feedback and remediation of security issues. The DEPLOY stage involves deploying the software to the target environment, which could be production, staging, or testing. Security considerations include securing the deployment process and ensuring the software is deployed in a secure configuration. Automated deployment tools, security settings configuration, and monitoring the deployed application for security issues are key activities. In the OPERATE stage, the software is monitored for performance, stability, and security. Continuous monitoring is emphasized to detect and respond to security incidents promptly. Activities include log analysis, security event monitoring, and incident response. Finally, the MONITOR stage involves continuously collecting and analyzing data to identify security threats and vulnerabilities. Feedback from this stage informs future planning and development cycles, ensuring continuous improvement of the DevSecOps process.

The importance of DevSecOps comes from several factors. First, the quick deployment cycles in DevOps can lead to security issues being missed or poorly addressed. DevSecOps aims to fix this by integrating security practices from the start, allowing organizations to deliver software that is both fast and secure [Myrbakken, Colomo-Palacios, 2017]. By including security considerations from the planning phase, DevSecOps encourages a proactive approach to risk management [Moyón et al., 2020]. This means anticipating potential security threats and vulnerabilities early in the development process, so teams can address risks before they become major problems. Second, DevSecOps also promotes a culture of shared responsibility for security, breaking down the barriers between development, operations, and security teams [Mohan, Othmane, 2016; Myrbakken, Colomo-Palacios, 2017]. This improved collaboration leads to better communication and knowledge sharing, resulting in a more comprehensive understanding of security throughout the organization. Third, DevSecOps promotes automation to streamline security practices, allowing teams to integrate security controls without slowing down development and deployment [Myrbakken, Colomo-Palacios, 2017]. Automation can include tasks like security testing, vulnerability scanning, and compliance monitoring, freeing up security experts to focus on more strategic tasks.

The integration of security requirements analysis into the DevSecOps process is essential for ensuring that security is addressed from the beginning of the development lifecycle and continuously throughout. This approach shifts security considerations to the forefront, embedding them into every stage of the DevSecOps pipeline. Authors indicate the importance of incorporating security requirements from the planning stage, as reflected in the PLAN stage of the Security Standard Compliant DevOps Pipeline for the IEC 62443 Standard [ISA, 2018; Moyón et al., 2020]. This involves treating security requirements as integral components of user stories and involving security experts in requirement elicitation. The emphasis on threat modeling and risk assessment from the outset [Myrbakken, Colomo-Palacios, 2017] underscores the need for proactive identification and mitigation of potential security risks.

The automation capabilities analysis for the IEC 62443 standard reveals that while some security activities can be fully automated (31 %), others require partial or complete manual intervention (69 %) [Moyón et al., 2020]. This challenge is particularly valid for the PLAN stage. The challenge remains in conducting rapid security requirements assessment in a DevSecOps environment due to the speed of continuous deployments. To address this, the concept of continuous security assessment is emphasized, where security is continuously monitored and addressed throughout the lifecycle, rather than only at the end [Rajapakse et al., 2022].

In this paper, we present an approach towards enhancing the PLAN stage with automating the security requirements analysis integrated into the Continuous Integration and Deployment pipeline.

Approach

In our approach, we shift security activities further left by focusing on the early phases of the DevSecOps lifecycle. Specifically, we emphasize automating security requirements analysis during the *PLAN* phase, integrating this process directly into the Continuous Integration/Continuous Deployment (CI/CD) pipeline. This method moves beyond the typical automation of verification tasks by addressing security concerns at the requirements stage. Figure 2 illustrates our overall approach.

To manage security requirements, we utilize the GitHub Issue Tracking system. Security requirements are captured as GitHub Issues, which allows for collaborative tracking and management. These requirements are then semantically mapped to Security Technical Implementation Guides (STIGs) [DoD, 2022] using Natural Language Processing (NLP).

STIGs are detailed guidelines aimed at securing IT systems and products used by the US Department of Defense (DoD) and other governmental agencies. STIGs provide specific instructions for configuring and securing a wide array of systems, including network devices, databases, operating

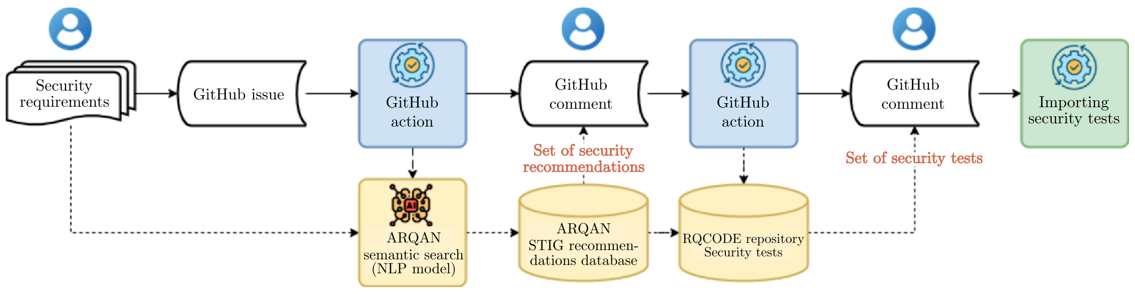


Figure 2. Automating security requirements analysis and testing

systems, and software. The goal of these guidelines is to minimize cybersecurity risks by enforcing secure configurations. For example, they cover areas such as database management systems, firewalls, virtualization, email servers, web servers, and identity and access management systems. The guidance includes specific settings and configurations aimed at mitigating cybersecurity threats.

STIGs provide platform-specific security recommendations, and we ensure that security requirements from, e. g., IEC 62443 are linked to the most relevant STIG guidance based on the deployment environment. To operationalize this, we created a repository of security tests aligned with the STIG guidelines. These tests are formulated using the Requirements as Code (RQCODE) concept. Here, STIG recommendations are formalized as security requirements, represented by Java objects. Each requirement object is linked with corresponding security tests, making the requirement both verifiable and reusable across different contexts. This formalization enhances traceability, ensuring that security requirements can be tested consistently throughout the CI/CD pipeline.

The automation of this process is achieved through GitHub Actions. These actions invoke our system, ARQAN, which leverages NLP to conduct semantic searches across a vectorized database of STIGs. Once relevant STIGs are identified, GitHub Actions are used to query the RQCODE repository for matching test classes. These security tests can then be integrated into the user's repository, enabling automated execution during the build or deployment stages, ensuring continuous validation of security requirements.

In the following subsections, we provide detailed explanations of the ARQAN and RQCODE processes, highlighting their role as the core components of our contribution to automating security requirements analysis.

ARQAN for sematic search in Security Technology Implementation Guides (STIGs)

A common challenge in security requirements engineering is the general nature of standard specifications. For instance, IEC 62443 [ISA, 2018] includes broad directives such as, "Components shall provide the capability to uniquely identify and authenticate all human users". While this flexibility allows developers to choose architectures and recommendations, it introduces potential vulnerabilities, as critical guidelines can be overlooked. To address this, our approach de-generalizes security requirements by mapping them to practical recommendations, tests, and fixes from the Security Technical Implementation Guides (STIGs) [DoD, 2022] database.

We developed a prototype called ARQAN, leveraging the SBERT [Reimers, Gurevych, 2019] architecture for semantic search across the Security Technical Implementation Guides (STIGs) [DoD, 2022]. This prototype enables engineers to map project-specific security requirements to concrete, platform-specific security recommendations found in STIGs. By identifying semantically similar guidelines, ARQAN aids in connecting project requirements with recognized standards, making security practices more traceable and actionable.

In the context of Natural Language Processing (NLP), semantic search identifies conceptual proximity between text elements. This approach is well-suited for linking general security requirements to specific countermeasures or guidelines, addressing the issue of vague requirements in security standards. Unlike traditional keyword-based search techniques that rely on word frequency or simple string matching, semantic search evaluates the underlying meaning of the text, providing deeper insights into the connections between requirements and standards [Wei, Barnaghi, Bargiela, 2008].

Semantic search goes beyond matching terms based on their surface-level similarity, focusing instead on the meaning behind words and phrases. By embedding both queries and documents into a high-dimensional vector space, semantic search finds the closest vectors, representing the most semantically similar content [Reimers, 2022a]. This vector space represents the deeper meanings of words, capturing the relationships between various security requirements and standards.

To measure similarity between these vectors, we use cosine similarity, which calculates the angle between vectors in an inner product space. A smaller angle indicates greater semantic similarity, making cosine similarity an effective metric for comparing text pairs such as security requirements and corresponding guidelines. The cosine similarity score ranges from -1 to 1, with higher values indicating closer alignment between the meanings of the text elements [Han, Pei, Tong, 2022].

In developing ARQAN's search functionality, a key challenge was selecting the most appropriate model for representing security requirements. After evaluating models such as MPNet and DistilRoBERTa, which are optimized for semantic search [Reimers, 2022b], we found that the vector representations produced by these models offered the best results for this task. By embedding both the security requirements from the STIGs database and user queries into the same vector space, ARQAN is able to retrieve the most relevant recommendations based on their semantic similarity.

In practice, the search process begins with transforming the STIGs database into vector representations using the chosen model. Queries are then embedded into the same vector space, and cosine similarity is applied to compare the query vectors with the STIG vectors. The results are sorted based on similarity scores, allowing the system to retrieve the most relevant security guidelines.

RQCODE for security requirements formalization and verification

Requirements as Code (RQCODE) approach [Ismaeel et al., 2021] is a novel method that applies the Seamless Object-Oriented Requirements (SOOR) paradigm [Naumchev, 2019] to implement security requirements in the Java programming language. RQCODE formalizes security requirements by representing them as Java classes. Each class encapsulates various representations of the requirement, including its natural language description and a set of methods to verify compliance, such as acceptance tests. This structure allows for direct traceability between a requirement and its implementation, enabling continuous verification through the execution of the embedded tests.

A key advantage of RQCODE is its alignment with object-oriented principles, which facilitates the reuse and extension of requirements. By leveraging inheritance, a requirement can extend or specialize another, making it easier to create templates for similar types of requirements. For instance, a generic security requirement class can serve as the foundation for more specific requirements by initializing it with different parameters. This modularity also simplifies the organization of related requirements, supporting efficient updates and maintenance in a security-sensitive environment.

In RQCODE, a properly specified requirement must be verifiable. Figure 3 outlines the main concepts in RQCODE. At the core is the abstract Requirement class, which includes a mandatory statement attribute to store the textual representation of the requirement. The class also implements the Checkable interface, which mandates the implementation of the check() method. This method is responsible for verifying the requirement's compliance. The results of this verification process are categorized into three possible outcomes: PASS, FAIL, or INCOMPLETE. A PASS indicates successful verification, while a FAIL signifies that the requirement was not met. An INCOMPLETE result occurs when the verification process cannot be completed, such as when necessary data is unavailable.

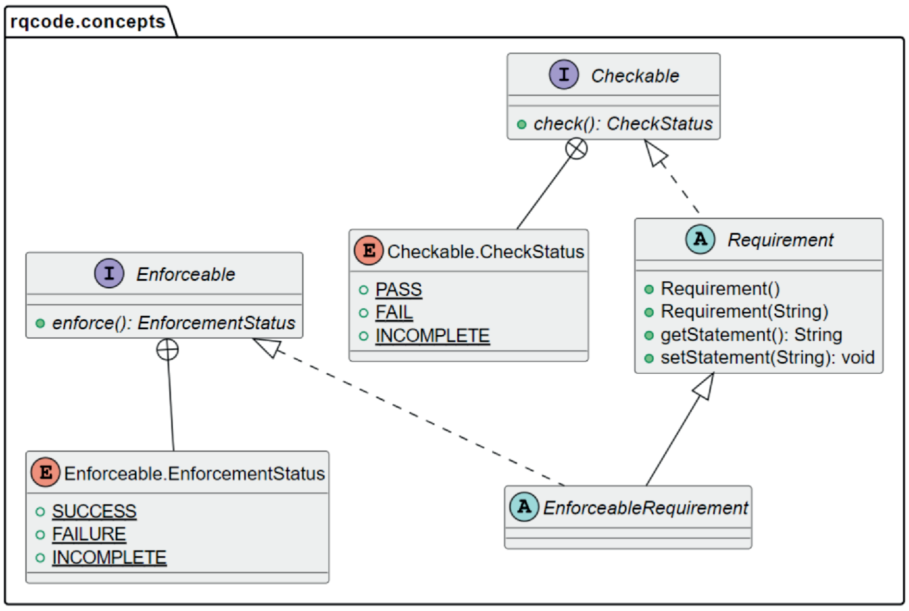


Figure 3. RQCODE concept classes (UML Class Diagram) [Sadovykh et al., 2023a]

In addition to the Checkable interface, RQCODE introduces an Enforceable interface for requirements that can modify the environment to achieve compliance. The enforce() method in this interface attempts to apply necessary changes and returns a result in the form of SUCCESS, FAILURE, or INCOMPLETE, depending on whether the modifications were successfully applied. This enforcement mechanism is particularly useful for security requirements, where automatic remediation (e. g., applying security patches or configuring settings) may be necessary to meet the requirement.

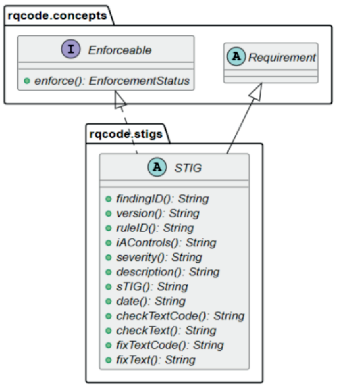
The object-oriented nature of RQCODE allows multiple related requirements to be grouped as class hierarchies, making it easier to manage and reuse them in various security specifications. This structure also enhances traceability, as references to related requirements can be easily identified and navigated.

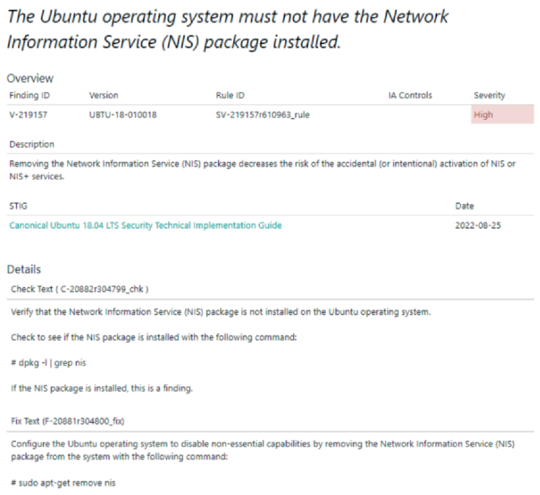
An important feature of RQCODE is that the formalization of security requirements as Java classes allows for several built-in validation mechanisms. First, Java compilers perform semantic checks, ensuring that the code follows the language's syntax and type rules. Additionally, static analysis tools integrated into Java Integrated Development Environments (IDEs) can identify potential issues such as dead code or unreachable branches in the requirements specification. Object-Oriented Programming (OOP) analysis techniques can also be applied to evaluate the clarity and structure of the security requirements. Established OOP metrics, such as those presented in [Chidamber, Kemerer, 1994; Succi et al., 2005], can be used to detect issues like duplicate requirements, circular dependencies, excessive coupling, or deep inheritance hierarchies. These analyses help ensure that the requirements are clear, atomic, non-contradictory, and verifiable.

When it comes to STIG, we focus on the guidelines description in natural language and two main components: Check Text and Fix Text. The Check Text provides detailed steps for verifying that a system meets the required security conditions, such as ensuring correct operating-system configurations. The Fix Text outlines specific actions for remedying security deficiencies, such as applying patches or adjusting system settings. This combination of instructions ensures both the identification and resolution of security issues.

We applied the Seamless Object-Oriented Requirements (SOOR) paradigm to formalize these STIG guidelines using the RQCODE framework. RQCODE translates security requirements into Java classes, with methods designed to check and enforce compliance. In this case, the STIG Check Text maps to the check() method in the RQCODE implementation, which verifies the correct configuration of the system, while the Fix Text is represented in the enforce() method, which applies necessary changes to meet the security requirement. This formalization allows the automatic verification and remediation of STIG guidelines, providing a structured and reusable approach for security compliance.

Figure 4 illustrates this formalization in a UML class diagram. The Requirement abstract class in RQCODE is extended to represent a specific STIG requirement. The check() method ensures that the system adheres to the security configuration specified in the STIG, and the enforce() method enables automatic remediation where applicable. Each STIG requirement is thus directly tied to its associated tests and fixes, facilitating traceability and enabling continuous verification within the CI/CD pipeline.




The Ubuntu operating system must not have the Network Information Service (NIS) package installed.

Overview

Finding ID	Version	Rule ID	IA Controls	Severity
V-219157	UBTU-18-010018	SV-219157r610963_rule		High

Description

Removing the Network Information Service (NIS) package decreases the risk of the accidental (or intentional) activation of NIS or NIS+ services.

STIG	Date
Canonical Ubuntu 18.04 LTS Security Technical Implementation Guide	2022-08-25

Details

Check Text (C-20882r304799_chk)

Verify that the Network Information Service (NIS) package is not installed on the Ubuntu operating system.

Check to see if the NIS package is installed with the following command:

```
# dpkg -l | grep nis
```

If the NIS package is installed, this is a finding.

Fix Text (F-20881r304800_fix)

Configure the Ubuntu operating system to disable non-essential capabilities by removing the Network Information Service (NIS) package from the system with the following command:

```
# sudo apt-get remove nis
```

Figure 4. STIG structure with RQCODE in Java (UML Class Diagram)

The overall structure of the RQCODE framework is depicted in Fig. 5. This framework includes core packages that define baseline concepts and temporal patterns, which serve as the foundation for any RQCODE-specified requirement. The baseline concepts provide general abstractions for defining security requirements, while temporal patterns facilitate the definition of dynamic security conditions, such as precedence, universality or eventuality. These core packages are extensible and can be combined with any specific security requirement to define a comprehensive security policy for an IT system.

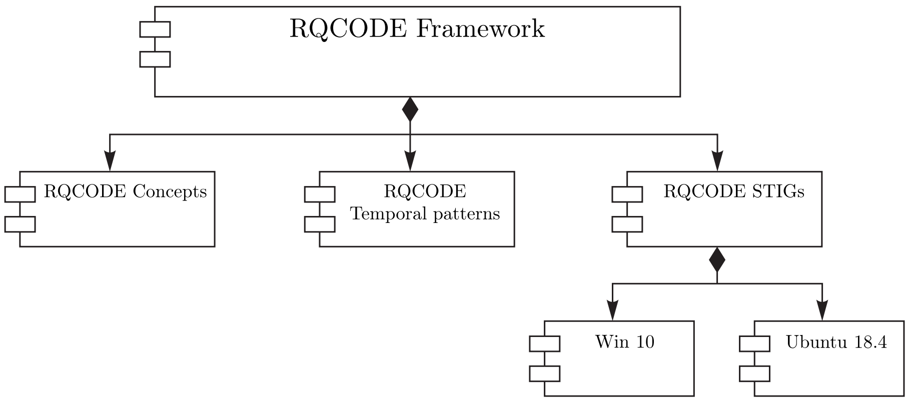


Figure 5. RQCODE Framework Structure (UML Class Diagram) [Sadovykh et al., 2023b]

For experimentation and validation, we prototyped specific implementations of STIGs for Windows 10 and Ubuntu 18.4 platforms. These implementations were organized into platform-specific packages within the RQCODE framework, allowing for modularity and reuse across different system environments. The platform-specific STIG classes contain the tailored logic for verifying and enforcing security guidelines relevant to the respective operating systems. By automating these processes, our approach reduces the likelihood of human error in implementing security configurations and ensures that systems remain compliant with evolving security standards.

The RQCODE framework inherently supports several validation mechanisms due to its basis in Java. Java compilers perform semantic checks on the requirement specifications, while static analysis tools integrated into Java IDEs can identify potential issues like duplicate code or unreachable conditions. Furthermore, Object-Oriented Programming (OOP) analysis techniques can be applied to evaluate the structure of the requirements, ensuring that they are atomic, clear, and non-contradictory. By applying these methods, we can systematically detect problems such as circular dependencies or excessive coupling in the security requirements, making it easier to maintain and evolve the requirements over time.

Formalizing STIG security guidelines as requirements using the RQCODE framework provides a structured and verifiable method for integrating security into the software development lifecycle. By transforming security guidelines into code, this approach enables continuous monitoring, automatic enforcement, and traceability of security requirements. This formalization ensures that security is maintained consistently across different platforms, providing a robust solution for automating security compliance in a DevSecOps context. The following section will explore further details of the implementation and validation of RQCODE using specific STIG examples.

Example

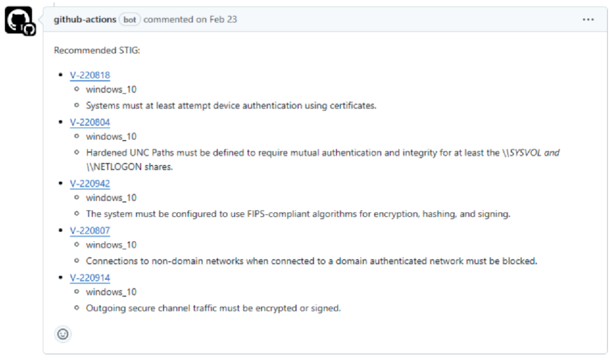
For demonstration purposes, we provide a full pipeline example in [Sadovykh, 2024], where automated security requirements analysis and testing for CI/CD pipelines are illustrated using ARQAN and RQCODE.

To illustrate the process, let us consider a requirement from IEC 62443: “The access via untrusted networks requirements are component-specific and can be located as requirements for each specific component type”. This requirement highlights the need for stringent controls when accessing components over untrusted networks.

In this approach, ARQAN is employed to semantically analyze the requirement and retrieve relevant Security Technical Implementation Guide (STIG) controls that could address the specific security concerns implied by the IEC 62443 standard. ARQAN returns several relevant guidelines:

- V-220818: Systems must at least attempt device authentication using certificates.
- V-220804: Hardened UNC Paths must be defined to require mutual authentication and integrity for at least the `\SYSVOL` and `\NETLOGON` shares.
- V-220942: The system must be configured to use FIPS-compliant algorithms for encryption, hashing, and signing.
- V-220807: Connections to non-domain networks when connected to a domain-authenticated network must be blocked.
- V-220914: Outgoing secure channel traffic must be encrypted or signed.

It is important to note that ARQAN does not rely on keywords or simple lexical matching to locate relevant requirements. For example, the recommendation that “Outgoing secure channel traffic must be encrypted or signed” has no lexical similarities to the original IEC 62443 statement, yet it is highly relevant from a security perspective. This ability to identify semantically similar requirements improves the accuracy and depth of the analysis.


github-actions bot commented on Feb 23

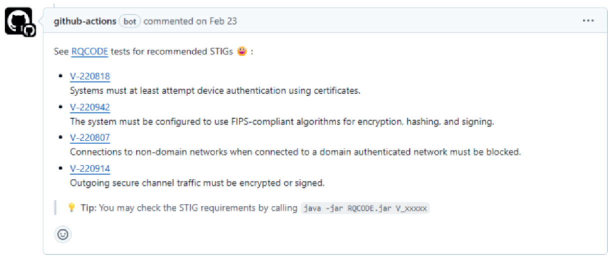
Recommended STIG:

- V-220818
  - windows_10
  - Systems must at least attempt device authentication using certificates.
- V-220804
  - windows_10
  - Hardened UNC Paths must be defined to require mutual authentication and integrity for at least the *SYSVOL and* \\NETLOGON shares.
- V-220942
  - windows_10
  - The system must be configured to use FIPS-compliant algorithms for encryption, hashing, and signing.
- V-220807
  - windows_10
  - Connections to non-domain networks when connected to a domain authenticated network must be blocked.
- V-220914
  - windows_10
  - Outgoing secure channel traffic must be encrypted or signed.

Figure 6. Example: ARQAN's recommendations for an IEC 62443 requirement

Following the identification of relevant guidelines, the GitHub bot searches the RQCODE repository for available tests and fixes associated with each identified STIG control. In this case, RQCODE provides verification and remediation options for the following guidelines: V-220818, V-220942, V-220807, and V-220914. Each of these tests and fixes can then be automatically imported into the CI/CD pipeline.

Once integrated into the pipeline, these verification tests and fixes will be executed at each relevant stage, such as during build, deployment, or post-deployment checks. This integration ensures continuous security validation by automating both the detection and remediation of configuration issues aligned with specific security requirements, enhancing compliance and security readiness throughout the software lifecycle.


github-actions bot commented on Feb 23

See RQCODE tests for recommended STIGs 😛 :

- V-220818
  Systems must at least attempt device authentication using certificates.
- V-220942
  The system must be configured to use FIPS-compliant algorithms for encryption, hashing, and signing.
- V-220807
  Connections to non-domain networks when connected to a domain authenticated network must be blocked.
- V-220914
  Outgoing secure channel traffic must be encrypted or signed.

> 💡 Tip: You may check the STIG requirements by calling `java -jar RQCODE.jar V_xxxxx`

Figure 7. Example: Tests and fixes located in the RQCODE repository

Evaluation

We evaluated our approach to automating security requirements analysis and testing in CI/CD pipelines through a collaboration with a large company in the industrial automation sector. This company, which needs secure setups for its industrial PCs (IPCs), required a way to implement Security Technical Implementation Guides (STIGs) for the Windows 10 platform in their systems.

To cover the relevant security guidelines, we integrated ARQAN with a complete database of STIG recommendations. In RQCODE, we developed 172 test and fix classes specifically for Windows 10 STIGs, covering 66.15 % of the 260 recommendations for that platform. This allowed us to address a substantial part of the Windows 10 security guidelines.

For the evaluation, the company provided a set of 40 standard security requirements specific to their work environment. Using ARQAN, we identified corresponding security recommendations for 85 % of these requirements. Within RQCODE, 62 % of these identified recommendations had available tests and fixes, enabling automated verification and remediation.

Afterwards, we held an interview with the company's representatives to discuss the approach and results. They considered the recommendations and test coverage useful and effective for their context. Overall, they evaluated our method as suitable for automating security requirements analysis and testing in their CI/CD workflows.

Related work

NLP for requirements engineering

In the context of requirements engineering (RE), applying Natural Language Processing (NLP) techniques is a natural fit due to their capability for comprehensive linguistic analysis. NLP, which deals with how computers process human language, has led to the emergence of NLP4RE, a field dedicated to supporting various RE tasks across different phases [Jurafsky, Manning, 2012; Liddy, 2001; Zhao et al., 2020].

The RE process involves analyzing diverse documents such as interview transcripts, standards, and legislation. The methods for automating RE vary significantly depending on the stage of RE. For instance, early stages involve processing raw information, while later stages focus on documents produced during the RE process [Sawyer, Rayson, Cosh, 2005]. NLP methods in RE aim to address tasks like detecting language issues, identifying key domain concepts, and establishing traceability links among requirements.

NLP solutions in RE can be categorized into detection, classification, clustering, pattern extraction, and modeling [Zhao et al., 2020]. Detection focuses on identifying ambiguities in requirements to ensure clarity and correctness. Classification involves predicting categorical classes, such as functional or non-functional requirements, and can include sentiment analysis of user feedback [Hastie, Tibshirani, Friedman, 2009]. Extraction retrieves specific terms from requirement texts for glossaries, aiding in consistency checking and classification. Clustering organizes documents or requirements into cohesive subsets. Modeling uses extracted data to generate models like UML for analysis and design, supporting tasks such as feature synthesis and security testing.

In the security context, NLP techniques are applied to tasks like vulnerability detection and repair, and specification analysis. Vulnerability detection identifies vulnerable code sequences, while vulnerability repair transforms vulnerable code into secure code by learning from examples. Specification analysis assesses security risks in product specifications before code is written, using NLP to process vulnerability descriptions [Kommrusch, 2019].

Transfer learning has emerged as a promising approach to address generalization problems in RE. Hey et al. [Hey et al., 2020] highlight that existing classification methods often fail on unseen projects due to variations in requirements formulation and style. Transfer learning, particularly using models like BERT, helps overcome this by fine-tuning pre-trained models on specific tasks with minimal data. BERT-based models, such as NoRBERT, have shown significant improvements in classification performance, achieving high F1-scores for functional and non-functional requirements [Devlin et al., 2018; Cleland-Huang et al., 2007].

Ajagbe et al. [Ajagbe, Zhao, 2022] introduced BERT4RE, a BERT-based model retrained on requirements texts, which outperforms the base BERT model in tasks like identifying key domain concepts. Ameri et al. [Ameri et al., 2021] and Ranade et al. [Ranade et al., 2021] further fine-tuned BERT for specific domains, achieving notable improvements in classification accuracy and specialized entity recognition. Li et al. [Li et al., 2022] applied a graph attention network (GAT) to enhance BERT's performance in requirements classification, reporting high F1-scores on both seen and unseen projects.

For the requirements similarity analysis, early methods relied on traditional corpus-based techniques to calculate word similarity, as noted by Alnajem et al. [Alnajem, Binkhonain, Shamim Hossain, 2024]. These methods, including word frequency-based techniques, term-based matching, and encoding requirements in XML, were effective but labour-intensive and lacked the efficiency and scalability of modern deep-learning approaches.

Recent advancements have leveraged machine learning and deep learning to improve the efficiency and accuracy of SRS detection. Das et al. [Das et al., 2021] introduced two domain-specific sentence embedding models based on the BERT architecture: PUBER, trained on the PURE dataset of 79 natural language requirements documents, and FiBER, which uses a pre-trained BERT model fine-tuned on the PURE dataset to find cosine similarity between requirement pairs.

Malik et al. [Malik et al., 2024] developed a supervised, two-phase framework called the Supervised Semantic Similarity-based Conflict Detection Algorithm (S3CDA). The first phase identifies conflict candidates through textual similarity using sentence embeddings and cosine similarity, while the second phase refines these candidates by measuring overlapping entities in the requirement texts. Additionally, Malik et al. presented an unsupervised variant, UnSupCDA, which handles unlabeled requirements and identifies conflicts using core elements of S3CDA.

Alnajem et al. [Alnajem, Binkhonain, Shamim Hossain, 2024] also proposed a novel metric-based learning method using Siamese Neural Networks (SNNs). This approach combines a sentence Transformer model and Long Short-Term Memory (LSTM) networks with a backward network layer to measure SRS.

Requirements formalization and verifiability

To make requirements precise and verifiable, researchers have long advocated mathematics-based notations and methods, known as “formal” methods. Various approaches to requirement formalization exist, differing in style, scope, and applicability.

Bruel et al. [Bruel et al., 2021] categorize these approaches into five types: Natural language, Semi-formal, Automata/Graphs, Mathematical, and Seamless (programming-language-based). Natural language approaches express requirements in human languages like English. This method is crucial for understanding system domain knowledge, often through documents, text analysis, or stakeholder interviews. Validation is typically done through discussions and interpretations with stakeholders [Rolland, Proix, 2006]. Semi-formal approaches use partially formalized notations, such as SysML, to represent requirements as artifacts and demonstrate semantic relations like dependency or refinement [Bruel et al., 2021]. Analysis remains mostly manual. Automata/Graphs methods are based on automata or graph theory, formalizing requirements as finite automata, which are idealized machines used to recognize patterns within input from a character set [Nelson, 2022]. Mathematical methods use mathematical and algebraic formalisms, such as Event-B [Abrial, 2010], Alloy [Jackson, 2011], Form-L [Bouskela et al., 2022], VDM [Bjørner, 1979], and Tabular Relations [Parnas, 2011]. Seamless methods, based on programming languages, apply constraint logic and programming by contract [Naumchev, Meyer, 2017].

Verifiability is crucial for requirement quality, impacting understandability and traceability. A correctly specified requirement must be verifiable, implying the presence of a specification for the verification method — static and dynamic. Static approaches work at the design or implementation level without running the system, while dynamic approaches generate and run security tests. Static approaches include model-checking, which uses a formal model of the system to check desired properties such as formalized security requirements. This approach requires an architectural or behavioural model, while code analysis works with candidate program implementations. Dynamic approaches include model-based testing, which generates tests based on architectural and behavioural models, facilitating test-driven development, and vulnerability testing, which attacks running applications, either directly or based on identified security risks and requirements.

Discussion and conclusions

The proposed automated approach enables the dynamic adaptation of security checks within CI/CD pipelines, ensuring that security requirements evolve alongside software development changes while remaining aligned with recognized standards like IEC 62443. By integrating ARQAN and RQCODE, we embed security requirements analysis and verification directly into the CI/CD process. This structure promotes a proactive approach to security compliance, addressing security issues early in development and reducing the likelihood of vulnerabilities in production environments. This evaluation indicates broad applicability of the approach, especially in fields like industrial automation, where secure configurations and automated compliance checks are essential. In collaboration with our industrial partner, we demonstrated the feasibility of integrating ARQAN and RQCODE into CI/CD workflows for effective platform-specific security.

Advantages and practicality

One of the primary benefits of our approach lies in ARQAN's ability to generate specific, actionable recommendations tailored to high-level security requirements. ARQAN matches high-level standards with specific platform security guidelines, ensuring that even generic requirements are made concrete and actionable. Through RQCODE, a large number of pre-built tests and fixes are available, making implementation straightforward within CI/CD processes. The demonstration indicates the method's suitability for automating security assessments as early as the planning stage in the DevSecOps lifecycle, supporting a proactive security.

Limitations

While promising, there are limitations in the current implementation. ARQAN's recommendations rely heavily on the STIG database, so the system is limited by the scope of guidelines available for specific platforms, which may not fully align with all security needs. The STIG database is updated periodically, and ARQAN thus requires consistent updates to remain effective.

RQCODE, implemented in Java, requires a Java Runtime Environment (JRE) on the target system, which may be resource-intensive for some environments. Additionally, certain tests and fixes within RQCODE require administrative privileges, adding a layer of complexity for environments with limited access permissions. Presently, RQCODE primarily supports Windows 10, with only partial coverage for Ubuntu 18. Expanding support for additional platforms would enhance the approach's applicability.

Future work

Future development of ARQAN could include incorporating broader security recommendations, such as those from the Open Web Application Security Project (OWASP), as well as experimenting with advanced models, including those based on large language models (LLMs). ARQAN's capabilities could be expanded to include security threat analysis and recommendations based on NIST standards to create a more comprehensive security tool.

For RQCODE, we plan to explore implementing the Seamless Object-Oriented Requirements (SOOR) paradigm in more lightweight object-oriented programming (OOP) languages, which may make it more adaptable to environments with limited resources. Extending RQCODE's library to support Linux platforms beyond Ubuntu 18 would also be beneficial. Additional explorations include adapting RQCODE to support containerized deployments, such as in Docker environments, to support secure development practices across a broader range of modern deployment architectures.

Overall, the proposed approach offers a practical step forward in shifting security verification leftward in the DevSecOps lifecycle, helping to identify and address security issues earlier in the software development process.

References

Abrial J.-R. Modeling in Event-B: system and software engineering. — Cambridge University Press, 2010.

Ajagbe M., Zhao L. Retraining a BERT model for transfer learning in requirements engineering: a preliminary study // 2022 IEEE 30th International Requirements Engineering Conference (RE). — 2022. — P. 309–315.

Alnajem N. A., Binkhonain M., Shamim Hossain M. Siamese neural networks method for semantic requirements similarity detection // IEEE Access. — 2024. — Vol. 12. — P. 140932–140947. — https://ieeexplore.ieee.org/abstract/document/10697170

Ameri K., Hempel M., Sharif H., Lopez J. Jr., Perumalla K. CyBERT: cybersecurity claim classification by fine-tuning the BERT language model // Journal of Cybersecurity and Privacy. — 2021. — Vol. 1, No. 4. — P. 615–637. — https://www.mdpi.com/2624-800X/1/4/31

Bjørner D. The vienna development method (VDM) // Mathematical studies of information processing / Eds.: E. K. Blum, M. Paul, S. Takasu. — Berlin, Heidelberg: Springer, 1979. — P. 326–359. — DOI: 10.1007/3-540-09541-1_33

Bouskela D., Falcone A., Garro A., Jardin A., Otter M., Thuy N., Tundis A. Formal requirements modeling for cyber-physical systems engineering: an integrated solution based on FORM-L and Modelica // Requirements Eng. — 2022. — Vol. 27, No. 1. — P. 1–30. — https://doi.org/10.1007/s00766-021-00359-z

Bruel J.-M., Ebersold S., Galinier F., Mazzara M., Naumchev A., Meyer B. The role of formalism in system requirements // ACM Comput. Surv. — 2021. — Vol. 54, No. 5. — P. 93:1–93:36. — https://doi.org/10.1145/3448975

Chidamber S. R., Kemerer C. F. A metrics suite for object oriented design // IEEE Transactions on Software Engineering. — 1994. — Vol. 20, No. 6. — P. 476–493. — DOI: 10.1109/32.295895

Cleland-Huang J., Mazrouee S., Liguo H., Port D. nfr. — [Electronic resource]. — Zenodo, 2007. — https://doi.org/10.5281/zenodo.268542

Das S., Deb N., Cortesi A., Chaki N. Sentence embedding models for similarity detection of software requirements // SN Computer Science. — 2021. — Vol. 2, No. 2. — P. 69. — https://doi.org/10.1007/s42979-020-00427-1

Devlin J., Chang M.-W., Lee K., Toutanova K. Bert: Pre-training of deep bidirectional transformers for language understanding // arXiv preprint. — 2018. — arXiv:1810.04805

DoD. Security technical implementation guides (STIGs) complete list // 2022. — https://www.stigviewer.com/stigs

Han J., Pei J., Tong H. Data mining: concepts and techniques. — Morgan kaufmann, 2022.

Hastie T., Tibshirani R., Friedman J. The elements of statistical learning: data mining, inference, and prediction. — Springer Science & Business Media, 2009.

Hey T., Keim J., Koziolek A., Tichy W. F. NoRBERT: Transfer learning for requirements classification // 2020 IEEE 28th International Requirements Engineering Conference (RE). — 2020. — P. 169–179.

ISA. IEC 62443: Security for industrial automation and control systems — Part 4-1: Secure product development lifecycle requirements. — 2018.

Ismaeel K., Naumchev A., Sadovykh A., Truscan D., Enoiu E. P., Seceleanu C. Security requirements as code: example from VeriDevOps project // 2021 IEEE 29th International Requirements Engineering Conference Workshops (REW). — 2021. — P. 357–363. — DOI: 10.1109/REW53955.2021.00063

Jackson D. Software abstractions, revised edition: logic, language, and analysis. — MIT Press, 2011.

Jurafsky D., Manning C. Natural language processing // Instructor. — 2012. — Vol. 212, No. 998. — P. 3482.

Kommrusch S. Artificial intelligence techniques for security vulnerability prevention // arXiv preprint. — 2019. — arXiv:1912.06796

Li G., Zheng C., Li M., Wang H. Automatic requirements classification based on graph attention network // IEEE Access. — 2022. — Vol. 10. — P. 30080–30090.

Liddy E. D. Natural language processing. — 2001.

Malik G., Cevik M., Parikh D., Basar A. Supervised semantic similarity-based conflict detection algorithm: S3CDA // arXiv. — 2024. — http://arxiv.org/abs/2206.13690

Mohan V., Othmane L. B. SecDevOps: Is it a marketing buzzword? — Mapping research on security in DevOps // 2016 11th International Conference on Availability, Reliability and Security (ARES). — 2016. — P. 542–547. — https://ieeexplore.ieee.org/abstract/document/7784617

Moyón F., Soares R., Pinto-Albuquerque M., Mendez D., Beckers K. Integration of security standards in DevOps pipelines: an industry case study // Product-focused software process improvement / Eds: M. Morisio, M. Torchiano, A. Jedlitschka. — Cham: Springer International Publishing, 2020. — P. 434–452. — DOI: 10.1007/978-3-030-64148-1_27

Myrbakken H., Colomo-Palacios R. DevSecOps: A multivocal literature review // Software process improvement and capability determination / Eds.: A. Mas, A. Mesquida, R. V. O'Connor, T. Rout, A. Dorling. — Cham: Springer International Publishing, 2017. — P. 17–29. — DOI: 10.1007/978-3-319-67383-7_2

Naumchev A. Seamless object-oriented requirements // 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON). — 2019. — P. 0743–0748. — DOI: 10.1109/SIBIRCON48586.2019.8958211

Naumchev A., Meyer B. Seamless requirements // Computer Languages, Systems & Structures. — 2017. — Vol. 49. — P. 119–132. — https://www.sciencedirect.com/science/article/pii/S1477842416301981

Nelson. Finite Automata. — [Electronic resource]. — 2022. — https://www.cs.rochester.edu/u/nelson/courses/csc_173/fa/fa.html

Parnas D. L. Precise documentation: the key to better software // The future of software engineering / Ed.: S. Nanz. — Berlin, Heidelberg: Springer, 2011. — P. 125–148. — https://doi.org/10.1007/978-3-642-15187-3_8

Rajapakse R. N., Zahedi M., Babar M. A., Shen H. Challenges and solutions when adopting DevSecOps: a systematic review // Information and Software Technology. — 2022. — Vol. 141. — P. 106700. — https://www.sciencedirect.com/science/article/pii/S0950584921001543

Ranade P., Piplai A., Joshi A., Finin T. CyBERT: contextualized embeddings for the cybersecurity domain // 2021 IEEE International Conference on Big Data (Big Data). — 2021. — P. 3334–3342.

Reimers N. Pretrained models — sentence-transformers documentation. — [Electronic resource]. — 2022a. — https://sbert.net/docs/pretrained_models.html

Reimers N. Semantic search — sentence-transformers documentation. — [Electronic resource]. — 2022b. — https://sbert.net/examples/applications/semantic-search/README.html

Reimers N., Gurevych I. Sentence-BERT: sentence embeddings using siamese BERT-networks // arXiv. — 2019. — http://arxiv.org/abs/1908.10084

Rolland C., Proix C. A natural language approach for Requirements Engineering // Advanced information systems engineering. Lecture Notes in Computer Science. — Berlin, Heidelberg: Springer, 2006. — Vol. 593. — P. 257–277. — DOI: 10.1007/BFb0035136

Sadovykh A. Demo VeriDevOps/ARQAN.action. — [Electronic resource]. — 2024. — https://github.com/VeriDevOps/ARQAN.action/issues

Sadovykh A., Messe N., Nigmatullin I., Ebersold S., Naumcheva M., Bruel J.-M. Security requirements formalization with RQCODE // CyberSecurity in a DevOps environment: from requirements to monitoring. — Springer, 2023a. — P. 65–92.

Sadovykh A., Yakovlev K., Naumchev A., Ivanov V. Natural language processing with machine learning for security requirements analysis: practical approaches // CyberSecurity in a DevOps environment: from requirements to monitoring. — Springer, 2023b. — P. 35–63.

Sawyer P., Rayson P., Cosh K. Shallow knowledge as an aid to deep understanding in early phase requirements engineering // Software Engineering, IEEE Transactions on. — 2005. — Vol. 31. — P. 969–981. — DOI: 10.1109/TSE.2005.129

Succi G., Pedrycz W., Djokic S., Zuliani P., Russo B. An empirical exploration of the distributions of the Chidamber and Kemerer object-oriented metrics suite // Empirical Software Engineering. — 2005. — Vol. 10, No. 1. — P. 81–104. — https://doi.org/10.1023/B:EMSE.0000048324.12188.a2

Wei W., Barnaghi P. M., Bargiela A. Search with meanings: an overview of semantic search systems // International journal of Communications of SIWN. — 2008. — Vol. 3. — P. 76–82.

Zhao L., Alhoshan W., Ferrari A., Letsholo K., Ajagbe M., Chioasca E.-V., Batista-Navarro R. Natural language processing for requirements engineering: a systematic mapping study. — 2020.