2021 IEEE 29th International Requirements Engineering Conference Workshops (REW) | 978-1-6654-1898-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/REW53955.2021.00063

Security Requirements as Code: Example from VeriDevOps Project

Khaled Ismaeel
Innopolis University
Innopolis, Russia
k.ismaeel@innopolis.university

Alexandr Naumchev
Schaffhausen Institute of Technology
Schaffhausen, Switzerland
an@sit.org

Andrey Sadovykh
Innopolis University and Softeam
Innopolis, Russia and Paris, France
a.sadovykh@innopolis.ru

Dragos Truscan
Åbo Akademi University
Turku, Finland
dragos.truscan@abo.fi

Eduard Paul Enoiu
Mälardalen University
Västeras, Sweden
eduard.paul.enoiu@mdh.se

Cristina Seceleanu
Mälardalen University
Västeras, Sweden
cristina.seceleanu@mdh.se

***Abstract*—This position paper presents and illustrates the concept of *security requirements as code* – a novel approach to security requirements specification. The aspiration to minimize code duplication and maximize its reuse has always been driving the evolution of software development approaches. Object-oriented programming (OOP) takes these approaches to the state in which the resulting code conceptually maps to the problem that the code is supposed to solve. People nowadays start learning to program in the primary school. On the other hand, requirements engineers still heavily rely on natural language based techniques to specify requirements. The key idea of this paper is: artifacts produced by the requirements process should be treated as input to the regular object-oriented analysis. Therefore, the contribution of this paper is the presentation of the major concepts for the security requirements as the code method that is illustrated with a real industry example from the VeriDevOps project.**

I. Introduction

A. Context

Cybersecurity has become one of the most important research topics. The number of discovered vulnerabilities has grown dramatically since 2002 and gone beyond 20000 [1]. In the mean time the average time to patch and fix a vulnerability is still close to 67 days [2]. Software vulnerabilities are one of the most critical and affect many industrial sectors from water supply to transportation [3]. Often the security of industrial applications is treated at infrastructure level after the system has been released and deployed [4]. That represents a serious risk for organizations and individuals, since that may lead to data breaches, economic losses and even threaten life safety [5]. The literature suggests that security has to be at the focus starting from the earliest stages of a project, since it may drastically affect the system architecture and the development process. Yet, dealing with security on requirements level is very difficult for several reasons. First, often the security of a system may be considered as something that the system should have by default and thus often omitted in the requirements documents or expressed in very general terms. Second, requirements are most commonly expressed in natural language which is prone to misinterpretations during the analysis. Third, security requirements are functional in nature and may be spread across functional requirements. In the best case, the requirements specification mentions a security standard or specific guidelines, such as Security Technical Implementation Guide (STIG) [6] that need to be mapped into enforcement and verification means. The current paper proposes a novel approach on applying the object-oriented paradigm [7] to security requirements that intends to alleviate many of the issues mentioned above.

This work is conducted in the context of Horizon 2020 VeriDevOps research project [8] that aims at creating an end-to-end toolchain for protection and prevention of security vulnerabilities at operations and development time. This toolchain intends to deal with security requirements, standards and guidelines in natural language. Those requirements specifications have to be translated into the security requirements by applying formalization patterns. By using this formal specifications, the project intends to bring means for protecting systems at operations as well as preventing vulnerabilities injection at development time. The current paper covers our effort of formalizing requirements expressed in natural language.

This project is supported by end-user companies including Fagor Arrasate (FAGOR). FAGOR produces smart industry equipment that needs to be secured. One of the challenges is that the infrastructure is controlled by Industry PCs (IPCs) to be configured according to the standard guidelines (STIGs). The application of the security policies has to be periodically verified. Thus the guidelines in natural language have to be mapped to specific scripts for configuration and verification.

B. Seamless Object-Oriented Requirements

Seamless Object-Oriented Requirements (SOORs) [9] is a requirements engineering approach that focuses on requirements verifiability and reusability. In SOORs, requirements are written as classes. A SOOR takes the form of a class that inherits from a SOOR template (SOORT), providing system-specific details through the object-oriented (OO) genericity

978-1-6654-1898-0/21/$31.00 ©2021 IEEE
DOI 10.1109/REW53955.2021.00063

and abstraction [7]. The main role of a SOORT is to encapsulate the verification complexity and make the template applicable across multiple systems. What remains to the specifier is to find the SOORT that most closely formalizes the behavioral pattern implied by the target requirement and then inherit from it, providing system-specific details through the OO genericity and abstraction. As a bonus, the resulting SOOR class can automatically produce a structured natural-language representation of the target requirement. As a result, the specifier will compare the initial, less formal, representation with the automatically generated one. Our experiments have demonstrated that this side-by-side comparison may lead either to switching the SOORT formalization pattern or reformulating the initial requirement altogether. Applying the SOORs method to a well-known collection of verification-oriented specification patterns [10] has clearly demonstrated [11] that a general-purpose object-oriented programming language with contracts is powerful enough to make the identified patterns practically reusable.

C. Requirements as Code

While SOORs focus on reusable verifiability, the more general concept of requirements as code suggests applying the OO techniques to all dimensions of requirements. The example that we handle in Section II contains the following components:

- Natural language description of a security requirement that a Windows-based server is expected to meet.
- A PowerShell script that checks whether the requirement is met or not on the target machine.
- A PowerShell script that enforces the requirement on the target machine in the event it is not met.

We took the example from a large requirements document consisting of requirements with the same structure. The requirements document was kindly provided to us by FAGOR, our industrial partner. Even a superficial analysis of this document revealed a lot of duplication in the PowerShell code. Our goal is to exhaustively analyze the document with the object-oriented analysis techniques and the following research questions in mind:

- For the subsets of requirements with very similar PowerShell scripts, is it possible to reformulate their natural language components in such a way that they differ proportionally to their PowerShell-wise differences?
- How much reuse can we achieve if we derive object-oriented classes for subsets of similar requirements (patterns) and then express all requirements from the document as descendants of these pattern classes?

The preliminary results (Section II) that we have got after handling the very first requirement in the document give us motivated hope that our experiment makes a lot of sense.

II. The Example

The sample requirements provided to us by FAGOR are a subset of the Windows 10 Security Technical Implementation Guide (STIG), a requirement specification document published by the Defense Information Systems Agency "as a tool to improve the security of Department of Defense (DoD) information systems" [6]. The STIG is divided into several *profiles* according to confidentiality and mission assurance levels, typically totaling 9 profiles (3 confidentiality and 3 assurance levels). Each of these profiles consists of a linear list of *findings*, which are certain forms of natural language requirements with associated checking and enforcing procedures. The STIG under consideration consists of 287 findings. The sample from FAGOR consists of 20 of those findings, each of which has an additional PowerShell script for automating the enforce procedure.

Out of the total 20 samples provided by FAGOR, we will be considering 4 in particular:

- **V-63447:** The system must be configured to audit Account Management - User Account Management failures.
- **V-63449:** The system must be configured to audit Account Management - User Account Management successes.
- **V-63463:** The system must be configured to audit Logon/Logoff - Logon failures.
- **V-63467:** The system must be configured to audit Logon/Logoff - Logon successes.

We can deduce several patterns by observing only the titles of the findings above. It is obvious that all previous findings relate to Windows system audit policies. Also, we can see that the first two findings relate to user account management auditing while the last two findings relate to logon auditing. This grouping is clearly visible throughout the whole findings specification and not only their titles; the first two findings have identical description section in addition to identical check and fix (enforce) texts up to swapping the terms *success* and *failure*. Such similarity is also found between the last two findings. Furthermore, the description section in all findings contains the same opening sentences relating to Windows system audit policies in general. Finally, and indeed most importantly, the enforce scripts provided by FAGOR share the same patterns; identical PowerShell scripts within both groups up to swapping parameters `/success` and `/failure` of `auditpol.exe` Windows utility.

Straightforward object-oriented analysis would suggest minimizing (or eliminating) redundancy by building an abstraction over the linear list of requirements to get a tree of requirement classes instead. Such an abstraction would roughly consist of the following. We will use Java syntax in the listings below, nevertheless the concepts are portable to all object-oriented languages.

```
abstract class Requirement;
```

This class implements general features such as parsing a requirement in HTML and leaves checking and enforcing as abstract methods.

```
abstract class AuditPolicyRequirement
 extends Requirement;
```

Here lies the core logic of checking and enforcing procedures. The logic is implemented as simple querying and pattern

matching in `auditpol.exe`'s output. The exact parameters and strings to query and match are left abstract.

```
abstract class AccountMgmtRequirement
 extends AuditPolicyRequirement;
abstract class UserAccountMgmtRequirement
 extends AccountMgmtRequirement;
```

These are super classes of the first two findings in our sample. Here we implement relevant parameters and strings for querying and matching.

```
abstract class LogonLogoffRequirement
 extends AuditPolicyRequirement;
abstract class LogonRequirement
 extends LogonLogoffRequirement;
```

Finally, these classes serve as super classes of the last two findings in our sample, again implementing relevant parameters and string for querying and matching. Note that the inheritance hierarchy corresponds to Windows audit policies hierarchy noted in the titles of our sample findings.

With the abstraction hierarchy in place, we can write the final specification of findings as follows.

```
class V_63447
 extends UserAccountMgmtRequirement;
class V_63449
 extends UserAccountMgmtRequirement;
class V_63463
 extends LogonRequirement;
class V_63467
 extends LogonRequirement;
```

These class definitions implement the remaining parameters and strings for checking and enforcing procedures. The object-oriented hierarchy above provides several benefits:

- *Elimination of redundancy in natural language description*: for example, the common prefix paragraph of the *description* section in our sample findings would be specified in the `AuditPolicyRequirement` class, while the paragraphs for checking logon auditing would be parametrically specified in `LogonRequirement` class.
- *Elimination of redundancy in programmatic logic*: all PowerShell scripts are replaced by a single parameterized logic in the `AuditPolicyRequirement` class.
- *Improved maintainability of requirements*: should a specification update be needed, for example, the update can be applied to all related requirement by modifying their super class only.
- *Improved extensibility*: we can for instance create new requirements for auditing logoff events by extending the `LogonLogoffRequirement` class and all the natural language description templates and core checking and enforcing logic would be already implemented.
- *Combination of documentation and programmatic logic in one location*: in our case we can also specify natural language description in JavaDoc comments instead, possibly simplifying requirement parsing for us.

It is worth noting that this object-oriented construction is somewhat different from SOOR in the sense that it does not utilize generics. However, the core goal of catching requirement patterns is still realized in this implementation. Another difference is that formal verification of this construction is not straightforward since it depends on the external `auditpol.exe` utility, for which no formal verification is openly available. In general, STIG's rely heavily on operating system API's, the formal verification of which is an enormous effort in the case of Windows 10, which makes this form of requirements particularly difficult to verify. What these requirements offer instead are *tests*, which in many contexts are more concrete than a formal proof of a property. By *concrete* we mean that the testing process does not need any assumptions about the verified system. Unlike testing, formal verification inevitably requires certain assumptions about the indivisible building blocks of the system. For example, to verify a system software, the verifier will make assumptions about the behavior of operating system calls. To verify an operating system implementation, one will need to make assumptions about possible underlying CPU architectures and other devices that the resulting system is expected to operate. Testing, on the other hand, checks the actual behavior of the verified system, from the high-level implementation down to hardware.

III. Related Work

We have identified the following publications that have to do with security requirement patterns. Konrad et al. [12] formalize security patterns in the style of the "Gang of Four" [13] and then enrich them with LTL constraints in the spirit of specification patterns by Dwyer et al. [10]. A candidate system for verification is specified in class, sequence, and state UML diagrams. The practitioner of the approach then chooses a pattern and instantiates it based on the candidate system's model. The resulting instantiation of the chosen pattern is then submitted to the SPIN model checker for verifying its conformance to temporal properties. The full list of security patterns specified in this way may be found in [14].

Several other works build on top of the results by Konrad [12] and Wasserman [14]. The work by Yoshioka et al. [15] surveys approaches to security patterns. More specifically, it identifies key activities in security patterns extraction and application processes, then assessing different approaches based on how they contribute to the said activities. The survey only mentions two approaches to security patterns that enable precise checking of security properties: the already reviewed one by Konrad et al. [12], and another one by Jürjens et al. [16]. The work in [16] proposes encoding security properties in UMLsec [17], an extension of UML. The resulting UMLsec specification is then submitted to AutoFocus – a CASE tool that can generate test sequences. These test sequences need to be instantiated in the context of a candidate system to test the said system.

Work [18] by Ouchani and Debbabi surveys approaches to specification, verification, and quantification of security in model-based systems. Within the present document, we are mostly interested in pattern-based approaches that support

verification and have tool support. Among the works assessed in [18], the following ones meet our criteria: [12] (discussed earlier), [14] (discussed earlier), [19], [20], [21], [22], [23], [24], and [25]. All these works share common roots, in the sense that they do model checking of UML models in one or another way. The work by Ouchani et al. [25], however, has brought to our attention CAPEC – Common Attack Pattern Enumeration and Classification. The official CAPEC website [1] gives the following introduction:

> The Common Attack Pattern Enumeration and Classification effort provides a publicly available catalog of common attack patterns that helps users understand how adversaries exploit weaknesses in applications and other cyber-enabled capabilities. "Attack Patterns" are descriptions of the common attributes and approaches employed by adversaries to exploit known weaknesses in cyber-enabled capabilities. Attack patterns define the challenges that an adversary may face and how they go about solving it. They derive from the concept of design patterns applied in a destructive rather than constructive context and are generated from in-depth analysis of specific real-world exploit examples. Each attack pattern captures knowledge about how specific parts of an attack are designed and executed, and gives guidance on ways to mitigate the attack's effectiveness. Attack patterns help those developing applications, or administrating cyber-enabled capabilities to better understand the specific elements of an attack and how to stop them from succeeding.

Ouchani et al. model both the target systems and the CAPEC patterns as SysML activity diagrams. They then compute the probabilities of a given system being vulnerable to each CAPEC pattern by submitting the resulting activity diagrams to the PRISM [26] probabilistic model checker.

Many other results build on top of the CAPEC repository. Kotenko and Doynikova [27] present a technique and an accompanying tool for generating random attack sequences and security events based on CAPEC. The technique relies on network configurations as the main input. Kanakogi et al. [28] propose a natural language processing-based method to automatically trace the related CAPEC patterns from CVE entries. *This work is especially relevant to our project because it formalizes natural language too; our project is different in that it will formalize natural language that is even less formal than CVE descriptions.* Kanakogi et al. experimented with TF-IDF [29] and Doc2Vec [30] and concluded that TF-IDF was more accurate for the task of tracing the related CAPEC patterns from CVE entries. Yuan et al. [31] reported an then-ongoing effort of developing a tool that would take on input a STRIDE [32] threat model and automatically propose CAPEC attack patterns sorted by relevance to the input threat model. Kaiya et al. [33] proposed a method using which a requirements analyst can automatically acquire the candidates of attacks against a functional requirement. We found the said method especially interesting because it was the first CAPEC-based method to work with requirements as inputs. Also, based on our personal experience, real work on security requirements starts when functional requirements already exist in some form. Sometimes only functional requirements are specified, with security concerns being postponed till the later stages of the software process.

Williams [34] [35] builds his work on top of the results by Kaiya et al. [33]. He proposes an ontology based collaborative recommender system for security requirements elicitation. The proposed system takes use cases on input and identifies relevant CAPEC patterns. It then connects the identified CAPEC patterns with the system-specific vocabulary to construct abuse cases [36] for the system in question.

We had a discussion in the process of working on the present document, after which we decided to give special attention to security testing of APIs because of their widespread use. Sudhodanan et al. [37] proposed a methodology in which security experts can create attack patterns from known attacks. Then they describe a security testing framework that leverages attack patterns to automatically generate test cases for security testing of multi-party web applications. The approach relies on proxy-based web security scanners to record client-server interactions and automatically detect applicability of attack patterns to the recorded interactions. Sudhodanan et al. implemented their approach on top of OWASP ZAP [2] proxy-based web security scanner and uncovered twenty one previously unknown vulnerabilities in well-known multi-party web applications. Bozic et al. [38] capture attack patterns as UML state diagrams. They use ACTS [39], a combinatorial testing tool for generation of test input strings based on the attack patterns and domain specific parameters and constraints. Bozic and Wotawa [40] encode security testing patterns as UML state diagrams and submit them to a tool that automatically generates test cases from these diagrams. They implemented a prototype tool on top of the WebScarab [3] proxy-based web security scanning framework (one of the most mature tools in the field). The two above works above authored by Bozic originated from a project called DIAMONDS (ITEA2 project on Development and Industrial Application of Multi-Domain Security Testing Technologies). Several more works found in the literature happened to originate from that project.

Smith and Williams [41] developed six black-box security test patterns – for (1) input validation vulnerability tests, (2) force exposure tests, (3) malicious file tests, (4) malicious use of security functions tests, (5) dangerous URL tests, (6) audit tests. They also developed a tool called Security Test Pattern Instantiator (STPI; we could not find the tool online) to help software testers instantiate security test patterns based on functional requirements. Finally, Smith and Williams conducted a user case study in which 21 graduate and 26 undergraduate students used the STPI tool to develop a black

[1] https://capec.mitre.org/about/index.html

[2] https://www.zaproxy.org/

[3] https://github.com/OWASP/OWASP-WebScarab

box security test plan. The study revealed that the novices' decisions were very close to the "golden standard" developed by a committee of experts.

A comprehensive review of security testing techniques by Felderer et al. [42] let us identify another conceptual cluster of pattern-based approaches – risk-based approaches. The risk-based approaches use numerical evaluations of risks' severity to define the required level of test coverage when generating test cases for the associated risks. That is to say, the higher the risk's severity is, the more coverage will be required from the test cases generated for that risk. Großmann et al. [43] described a tool-based iterative approach that combines the CORAS [44] approach to model-driven risk analysis with automated security testing based on patterns such as CAPEC. In every iteration of the approach, the risk analysis results are fed into the process of identifying relevant security test patterns and then instantiating these patterns into actual test cases. The testing results are then fed back into the risk analysis process, and so forth. Botella et al. [45] (originating from the DIAMONDS project) propose an approach that starts with risk analysis that relies on an approach similar to CORAS [44] and concludes with automated security testing of the target system. The test generation process relies on CertifyIt [46], an existing model-based testing (MBT) software. CertifyIt takes on input behavioral models of the system expressed as UML statecharts and risk-based test purposes – formalizations of vulnerability test patterns. The work in [45] relies on an existing catalogue of security test patterns by Vouffo Feudjio [47]. Each of these patterns consists of test procedure template surrounded with other parameters defining when and how to apply the pattern. Other contributions of Feudjio include test automation design patterns for reactive software systems [48] and his PhD thesis [49].

The work of Feudjio was supported by the DIAMONDS project. Other results achieved within this project include a work by Wotawa and Bozic [50] where they propose a planning-based approach to security testing. The authors represent security testing as a planning problem with the goal of breaking the application under test. The approach focuses on security testing with no regard to patterns, which is why we do not detail it any further. In another work resulting from the DIAMONDS project [51], Schieferdecker et al. provide a general overview of the model-based testing field and locates the DIAMONDS project on it. The project is said to focus on risk-based security testing and model-based fuzzing. Schieferdecker et al. refer to the project's website for details[4]. Another work originating from the DIAMONDS project [52] describes the fundamental principles behind model-based security testing. Chronologically and conceptually, this work is a predecessor of the work in [45] already described above. A survey on model-based testing tools for test case generation by Li and Le Gall [53] mentions the results of the DIAMONDS project. Speaking of other pattern-based approaches, the survey only mentions one approach called VERA [54]. The website[5] by the survey, which was said to host a collection of patterns, is not available.

We then decided to find contributions that cite the work of Feudjio [48]. Herzner et al. [55] present an approach for capturing best practices in integrating risk and safety analysis and testing by means of analysis and testing (A&T) patterns. Each A&T pattern encompasses a dedicated workflow starting from analysis down to test steps for achieving certain qualities of the system. The authors mention a repository of 17 ready-to-use patterns[6]. The work by Herzner et al. is cited by a work by Dghaym et al. [56] to originate from the same project. Dghaym et al. present a concept of verification and validation (V&V) patterns that. V&V patterns are a unified format for capturing common verification and validation approaches, such as behavior driven development, model checking, the Event-B method, and others. When applicable, each pattern contains a formalization of the corresponding verification and validation procedure. The language of the formalization depends on the approach assumed by the said pattern. The work is accompanied by a repository of ready-to-use V&V patterns[6] Also, a whole PhD thesis focusing on pattern-driven and model-based vulnerability testing of web applications [57] by Alexandre Vernotte.

IV. Future work

Our ultimate goal is to seamlessly proceed from natural language requirements to their automated processing. Identifying repeating patterns (such as the one in Section II) in realistic requirements documents provided to us by our industrial partners will be our first task. We will then encode the identified patterns as OO classes, like the one we have described in Section II. The result of this effort will be a library of security requirement templates, reusable through the well known OO techniques. The next task would be to train a model that would map natural language descriptions to the templates from the resulting catalogue. The patterns identification step will have already produced the necessary data set as a by-product.

Existing results in the area of security patterns will become our next focus. We have identified several publicly available collections of security verification patterns while analyzing the related work (Section III):

- Temporal patterns [14] in the style of Dwyer et al. [10]; the patterns are enumerated in the article itself.
- CAPEC repository of attack patterns[6].
- Analysis and testing patterns[6] by Herzner et al. [55].
- Validation and verification patterns[6] by Dghaym et al. [56].

Among these catalogues, only the temporal patterns [12] [14] are immediately reusable for encoding with the object-oriented approach – the others rely on natural language descriptions. Our most important task is then to take the

[4]https://www.itea2-diamonds.org/

[5]http://www.spacios.eu/index.php/spacios-tool/

[6]https://vvpatterns.ait.ac.at/the-at-patterns/

[6]https://capec.mitre.org/about/index.html

remaining catalogues and convert them into a form that would be better suited for the automation purposes. The *security requirements as code* approach may provide such a form.

As mentioned above, in the context of the VeriDevOps project we will investigate the integration of template-based approaches within the process of obtaining OO requirements. The former use predefined, yet extensible, boilerplates or specification patterns to describe requirements in a structured manner and facilitate their consistency analysis and formalization. Such approaches help test engineers to formulate security requirements based on reusable descriptions. Prospective candidates for this integration are the following tools: (i) ProPaS [58], [59], [60] that uses specification patterns [10] to aid the formalization of natural language requirements, and (ii) ReSA [61], [62] that uses boilerplates to specify structured natural language requirements and performs on-demand consistency analysis by employing SMT techniques. These tools can be used to specify security requirements from textual descriptions of security threat and vulnerability scenarios. For example, the ProPaS tool enables the automatic transformation of natural language requirements into temporal logical formulas, e.g., Timed Computation Tree Logic (TCTL), based on specification patterns. Consequently, ProPaS can be applied to obtain security requirements expressed in the property language of model checking tools such as UPPAAL [63].

To complement the requirements formalization approach, the patterns implementation in TCTL can be explored. Once we generate a formal specification in TCTL, this can be used as input to verify a certain system design [64] or generate test cases covering these requirements. This assumes the creation of a design model as a network of UPPAAL timed automata.

V. Conclusion

As the discussion in Section II suggests, capturing security requirements as object-oriented code is practical. To demonstrate the practicality, we used a realistic example that comes from an industrial security requirements document. The document contains multi-view requirements, also known as *multirequirements* [65]. The different views are textual description, PowerShell script for checking conformance to the requirement, and PowerShell script for enforcing the requirement on the target system. We have applied object-oriented analysis to a single example from the document (Section II). The results clearly show that the experiment should be extended onto the whole document. We will write a follow-up paper with a more comprehensive report covering the results.

Acknowledgment

This work has received funding from Horizon 2020 programme under grant agreement No. 957212 - VeriDevOps project.

References

[1] T. E. U. A. for Cybersecurity (ENISA), "Is software more vulnerable today?" https://www.enisa.europa.eu/publications/info-notes/is-software-more-vulnerable-today, Mar. 2018, accessed: 2020-10-29.

[2] "EDGESCAN, 2018 VULNERABILITY STATISTICS REPORT," https://www.edgescan.com/wp-content/uploads/2018/05/edgescan-stats-report-2018.pdf, accessed: 2020-10-29.

[3] "Accenture, 2020 Cyber Threatscape Report," https://www.accenture.com/_acnmedia/PDF136/Accenture2020-CyberThreatscapeFullReport.pdf, accessed: 2020-10-29.

[4] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Security Privacy*, vol. 7, no. 2, pp. 87–90, 2009.

[5] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, 2015, pp. 145–152.

[6] "Windows 10 security technical implementation guideline," https://www.stigviewer.com/stig/windows_10/2020-06-15.

[7] B. Meyer, *Object-oriented software construction*. Prentice hall Englewood Cliffs, 1997, vol. 2.

[8] A. Sadovykh, G. Widforss, D. Truscan, E. P. Enoiu, W. Mallouli, R. Iglesias, A. Bagnto, and O. Hendel, "Veridevops: Automated protection and prevention to meet security requirements in devops," in *Design, Automation and Test in Europe Conference DATE 2021, 01 Feb 2021, Online, Sweden*, 2020.

[9] A. Naumchev, "Seamless object-oriented requirements," in *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*. IEEE, 2019, pp. 0743–0748.

[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 411–420.

[11] A. Naumchev, "Exigences orientées objets dans un cycle de vie continu," Ph.D. dissertation, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2019.

[12] S. Konrad, B. H. Cheng, L. A. Campbell, and R. Wassermann, "Using security patterns to model and analyze security requirements," *Requirements Engineering for High Assurance Systems (RHAS'03)*, vol. 11, 2003.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, "Elements of reusable object-oriented software," *Design Patterns. massachusetts: Addison-Wesley Publishing Company*, 1995.

[14] R. Wassermann and B. H. Cheng, "Security patterns," in *Michigan State University, PLoP Conf. Citeseer*. Citeseer, 2003.

[15] N. Yoshioka, H. Washizaki, and K. Maruyama, "A survey on security patterns," *Progress in informatics*, vol. 5, no. 5, pp. 35–47, 2008.

[16] J. Jürjens, G. Popp, and G. Wimmel, "Towards using security patterns in model-based system development," 2002.

[17] J. Jürjens, "Umlsec: Extending uml for secure systems development," in *International Conference on The Unified Modeling Language*. Springer, 2002, pp. 412–425.

[18] S. Ouchani and M. Debbabi, "Specification, verification, and quantification of security in model-based systems," *Computing*, vol. 97, no. 7, pp. 691–711, 2015.

[19] I. Siveroni, A. Zisman, and G. Spanoudakis, "Property specification and static verification of uml models," in *2008 Third International Conference on Availability, Reliability and Security*. IEEE, 2008, pp. 96–103.

[20] ——, "A uml-based static verification framework for security," *Requirements engineering*, vol. 15, no. 1, pp. 95–118, 2010.

[21] A. Zisman, "A static verification framework for secure peer-to-peer applications," in *Second International Conference on Internet and Web Applications and Services (ICIW'07)*. IEEE, 2007, pp. 8–8.

[22] J. Dong, T. Peng, and Y. Zhao, "Automated verification of security pattern compositions," *Information and Software Technology*, vol. 52, no. 3, pp. 274–295, 2010.

[23] S. Ouchani, O. A. Mohamed, and M. Debbabi, "A security risk assessment framework for sysml activity diagrams," in *2013 IEEE 7th International Conference on Software Security and Reliability*. IEEE, 2013, pp. 227–236.

[24] S. Ouchani, O. A. Mohamed, M. Debbabi, and M. Pourzandi, "Verification of the correctness in composed uml behavioural diagrams," in *Software Engineering Research, Management and Applications 2010*. Springer, 2010, pp. 163–177.

[25] S. Ouchani, Y. Jarraya, and O. A. Mohamed, "Model-based systems security quantification," in *2011 Ninth Annual International Conference on Privacy, Security and Trust*. IEEE, 2011, pp. 142–149.

[26] M. Kwiatkowska, G. Norman, and D. Parker, "Prism: Probabilistic symbolic model checker," in *International Conference on Modelling*

Techniques and Tools for Computer Performance Evaluation. Springer, 2002, pp. 200–204.

[27] I. Kotenko and E. Doynikova, "The capec based generator of attack scenarios for network security evaluation," in *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1. IEEE, 2015, pp. 436–441.

[28] K. Kanakogi, H. Washizaki, Y. Fukazawa, S. Ogata, T. Okubo, T. Kato, H. Kanuka, A. Hazeyama, and N. Yoshioka, "Tracing capec attack patterns from cve vulnerability information using natural language processing technique," in *Proceedings of the 54th Hawaii International Conference on System Sciences*, 2021, p. 6996.

[29] D. R. Miller, T. Leek, and R. M. Schwartz, "A hidden markov model information retrieval system," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, 1999, pp. 214–221.

[30] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[31] X. Yuan, E. B. Nuakoh, J. S. Beal, and H. Yu, "Retrieving relevant capec attack patterns for secure software development," in *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, 2014, pp. 33–36.

[32] A. Shostack, "Experiences threat modeling at microsoft." *MODSEC@ MoDELS*, vol. 2008, 2008.

[33] H. Kaiya, S. Kono, S. Ogata, T. Okubo, N. Yoshioka, H. Washizaki, and K. Kaijiri, "Security requirements analysis using knowledge in capec," in *International conference on advanced information systems engineering*. Springer, 2014, pp. 343–348.

[34] I. Williams, "An ontology based collaborative recommender system for security requirements elicitation," in *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 2018, pp. 448–453.

[35] I. Williams and X. Yuan, "Creating abuse cases based on attack patterns: A user study," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 85–86.

[36] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.

[37] A. Sudhodanan, A. Armando, R. Carbone, L. Compagna *et al.*, "Attack patterns for black-box security testing of multi-party web applications." in *NDSS*, 2016.

[38] J. Bozic, D. E. Simos, and F. Wotawa, "Attack pattern-based combinatorial testing," in *Proceedings of the 9th international workshop on automation of software test*, 2014, pp. 1–7.

[39] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Acts: A combinatorial test generation tool," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 370–375.

[40] J. Bozic and F. Wotawa, "Security testing based on attack patterns," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014, pp. 4–11.

[41] B. Smith and L. Williams, "On the effective use of security test patterns," in *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 2012, pp. 108–117.

[42] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Security testing: A survey," in *Advances in Computers*. Elsevier, 2016, vol. 101, pp. 1–51.

[43] J. Großmann, M. Schneider, J. Viehmann, and M.-F. Wendland, "Combining risk analysis and security testing," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2014, pp. 322–336.

[44] M. S. Lund, B. Solhaug, and K. Stølen, *Model-driven risk analysis: the CORAS approach*. Springer Science & Business Media, 2010.

[45] J. Botella, B. Legeard, F. Peureux, and A. Vernotte, "Risk-based vulnerability testing using security test patterns," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2014, pp. 337–352.

[46] F. Bouquet, C. Grandpierre, B. Legeard, and F. Peureux, "A test generation solution to automate software testing," in *Proceedings of the 3rd international workshop on Automation of software test*, 2008, pp. 45–48.

[47] A. Vouffo Feudjio, "Initial security test pattern catalog. public deliverable d3. wp4. t1, diamonds project, berlin, germany (june 2012)," 2014.

[48] A.-G. V. Feudjio, I. Schieferdecker, and A. Vouffo, "Test automation design patterns for reactive software systems," *ceur-ws. org*, vol. 566, p. E6_BlackBoxTesting, 2009.

[49] A.-G. V. Feudjio, "A methodology for pattern-oriented model-driven testing of reactive software systems," Ph.D. dissertation, Universitätsbibliothek der Technischen Universität Berlin, 2011.

[50] F. Wotawa and J. Bozic, "Plan it! automated security testing based on planning," in *IFIP International Conference on Testing Software and Systems*. Springer, 2014, pp. 48–62.

[51] I. Schieferdecker, J. Grossmann, and M. Schneider, "Model-based security testing," *arXiv preprint arXiv:1202.6118*, 2012.

[52] F. Lebeau, B. Legeard, F. Peureux, and A. Vernotte, "Model-based vulnerability testing for web applications," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2013, pp. 445–452.

[53] W. Li, F. Le Gall, and N. Spaseski, "A survey on model-based testing tools for test case generation," in *International Conference on Tools and Methods for Program Analysis*. Springer, 2017, pp. 77–89.

[54] A. Blome, M. Ochoa, K. Li, M. Peroli, and M. T. Dashti, "Vera: A flexible model-based vulnerability testing tool," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 471–478.

[55] W. Herzner, S. Sieverding, O. Kacimi, E. Böde, T. Bauer, and B. Nielsen, "Expressing best practices in (risk) analysis and testing of safety-critical systems using patterns," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 2014, pp. 299–304.

[56] D. Dghaym, T. Fischer, T. S. Hoang, K. Reichl, C. Snook, R. Schlick, and P. Tummeltshammer, "Systematic verification and testing," in *Validation and Verification of Automated Systems*. Springer, 2020, pp. 89–104.

[57] A. Vernotte, "A pattern-driven and model-based vulnerability testing for web applications," Ph.D. dissertation, Université de Franche-Comté, 2015.

[58] P. Filipovikj and C. Seceleanu, "Specifying industrial system requirements using specification patterns: A case study of evaluation with practitioners," in *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. SCITEPRESS, 2019, pp. 92–103.

[59] P. Filipovikj, G. Rodriguez-Navas, M. Nyberg, and C. Seceleanu, "Automated smt-based consistency checking of industrial critical requirements," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 15–28, 2018.

[60] P. Filipovikj, T. Jagerfield, M. Nyberg, G. Rodriguez-Navas, and C. Seceleanu, "Integrating pattern-based formal requirements specification in an industrial tool-chain," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE CS, 2016, pp. 167–173.

[61] N. Mahmud, C. Seceleanu, and O. Ljungkrantz, "Resa: An ontology-based requirement specification language tailored to automotive systems," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2015, pp. 1–10.

[62] ——, "Resa tool: Structured requirements specification and sat-based consistency-checking," in *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2016, pp. 1737–1746.

[63] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International journal on software tools for technology transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[64] D. Slutej, J. Håkansson, J. Suryadevara, C. Seceleanu, and P. Pettersson, "Analyzing a pattern-based model of a real-time turntable system," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 1, pp. 161–178, 2009, proceedings of the Sixth International Workshop on Formal Engineering approches to Software Components and Architectures (FESCA 2009).

[65] B. Meyer, "Multirequirements," in *Modelling and Quality in Requirements Engineering: Essays dedicated to Martin Glinz on the occasion of his 60th birthday*. Verl.-Haus Monsenstein u. Vannerdat, 2013.