Poster: Systematic Elicitation of Common Security Design Flaws

Stef Verreydt
imec-DistriNet
KU Leuven
Heverlee, Belgium
stef.verreydt@kuleuven.be

Koen Yskout
imec-DistriNet
KU Leuven
Heverlee, Belgium
koen.yskout@kuleuven.be

Laurens Sion
imec-DistriNet
KU Leuven
Heverlee, Belgium
laurens.sion@kuleuven.be

Wouter Joosen
imec-DistriNet
KU Leuven
Heverlee, Belgium
wouter.joosen@kuleuven.be

***Abstract*—Threat modeling allows potential security threats to be identified and mitigated at design time. Countermeasures in current threat modeling approaches are mostly modeled as a boolean: either they are implemented, or they are not. This does not allow to take into account potential design flaws for the countermeasure itself. A considerable number of security issues is, however, related to the wrong or incomplete application of common security tactics. For example, the effectiveness of audit logs drops if the data written to the logs is not sanitized. In this paper, we describe our novel approach which aims to systematically and automatically identify common security design flaws.**

***Index Terms*—Threat modeling, CWE, Security-by-design**

1. Introduction

Security and privacy by design principles are becoming increasingly important to develop secure software systems. Indeed, insecure design is one of the most critical software risks according to the OWASP Top 10 2021 [1], and adhering to security and privacy by design principles is even obligated by regulations such as the General Data Protection Regulation (GDPR) [2].

Threat modeling provides a systematic approach to analyze the security and privacy of a software design, thereby allowing potential threats to be identified early on in the development lifecycle. The first step of a threat modeling exercise involves creating a model of the system being analyzed, usually as a Data Flow Diagram (DFD). The DFD notation comprises just five elements, namely *processes*, *data stores*, *external entities*, *data flows* and *trust boundaries*. That model can then be analyzed to identify potential security threats. Tool support for automatic threat elicitation based on machine-readable system models is widespread, and new techniques are being developed rapidly [3]. Common threat elicitation methods used by these tools are based on STRIDE (an acronym for *spoofing*, *tampering*, *information disclosure*, *denial of service* and *elevation of privilege*), but more specific types of threats or attacks such as CAPEC, CWE or CVE entries are also identified by some [4].

The next step of a threat modeling exercise is to mitigate the identified threats by introducing countermeasures. This often involves standard tactics, for example using logging to mitigate repudiation threats [5]. Applying such tactics in a design requires careful consideration of their precise requirements and assumptions. For example, data written to audit logs should be sanitized (CWE-117[1]), and should not contain sensitive information such as credentials (CWE-532[2]). Ideally, design flaws which violate such requirements should be flagged automatically.

One of the underlying issues which prevents this type of analyses is that most threat modeling tools [6] only allow to capture the effect of a countermeasure, and not how countermeasures are included in a design or which elements are involved [7]. For example, a repudiation threat could be marked as mitigated in the Microsoft Threat Modeling Tool [6], but there is no support to explicitly capture the countermeasure which mitigates the threat. To allow tracing back why and how threats are mitigated, Sion et al. [7] extended the DFD notation with a first-class representation for countermeasures. For example, a logging countermeasure can be explicitly added to the system model, allowing tool support to automatically mark certain repudiation threats as mitigated. In our novel approach, we leverage this explicit countermeasure information to automatically identify flaws rooted in the design of the countermeasure itself.

Tuma et al. [8] leverages this notation to automatically identify common security design flaws. Their approach, however, only identified five flaws automatically, and their identification method is mostly based on missing countermeasures rather than design flaws in the countermeasures themselves. For example, one of the flaws identified by their approach is *"insufficient auditing"*, which is identified simply based on the lack of a logging countermeasure. Still their empirical evaluation shows that automatically identifying security design flaws is possible with acceptable precision and recall. In this paper, we therefore describe an approach similar to the one by Tuma et al. [8], but with the ability to automatically identify flaws rooted in the design of countermeasures themselves.

In summary, the goal of our proposal is the following:

Goal. *Automatically eliciting potential security design flaws related to applying standard security tactics during threat modeling.*

In what follows, we provide an overview of our proposed approach, and discuss the advantages compared to traditional threat modeling approaches.

1. https://cwe.mitre.org/data/definitions/117.html
2. https://cwe.mitre.org/data/definitions/532.html

2. Proposal Overview

A high-level overview of our proposal is shown in Fig. 1. The proposed approach was implemented as a extension of an existing threat modeling tool [9] to demonstrate its feasibility. We shortly discuss the main concepts of our proposal and describe the main advantages compared to existing threat modeling approaches.

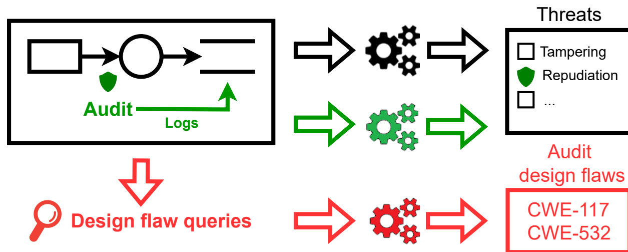

Figure 1. High-level overview of our proposal.
The default threat modeling flow is shown in black. The extension by Sion et al. [7], which adds first-class representations for countermeasures, is highlighted in ***green****. Our proposal further extends this notation by adding queries for common security design flaws, as shown in* ***red****.*

2.1. Extended system model

The following information should be included in the system model to enable the systematic elicitation of common security design issues:

- a DFD of the system, annotated with data type information;
- a structured and generic description of common security tactics; and
- information on how these tactics are applied in the system being analyzed.

Each of these is shortly discussed in what follows.

2.1.1. System description. Figure 2 shows a DFD for a simple client-server application which will be used as a running example. To enable systematically identifying that, for example, data written to audit logs is not sanitized, information on data types is required, which is not included in the default DFD notation. The data flows are therefore annotated with data type information, similar to the proposal by Tuma et al. [10].

2.1.2. Security tactic description. Applying a traditional threat modeling approach such as STRIDE to the DFD

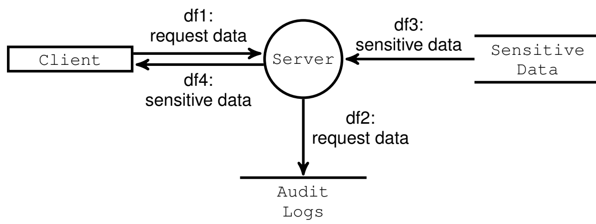

Figure 2. A DFD showing that a *Client* (external entity) can request data from the *Server* (process), which logs the request in one data store and fetches sensitive from another.

shown in Fig. 2 would return several potential threats, for example a tampering threat on the *Audit Logs* or a repudiation threat on the *Server*. The default DFD notation, however, does not allow to systematically capture that the audit tactic is applied to mitigate the repudiation threat on the *Server*. For that, we first need to define what the audit tactic encompasses. For our proposed approach, common security tactics are described generically, similarly to the proposal by van den Berghe et al. [11]. The advantage of generic tactic descriptions is that they can be applied in different countermeasures and across different system models. A tactic is defined by (i) a name, (ii) the threat(s) which it mitigates, and (iii) the roles which make up the tactic. A role is defined by a name and a type (*proces*, *data store*, *external entity*, *data flow* or *data type*). A generic description of the audit tactic is shown in Fig. 3.

```
Name: Audit
Roles:
   - Logged event     : Data Flow
   - Logged data      : Data Type
   - Log database     : Data Store
   - Protected entity: Process
   - Logging process : Process
Mitigates:
   - Repudiation threats on protected entity
```

Figure 3. Structured description of the audit tactic.

2.1.3. Applying tactics. The system model can then be enriched with information on how tactics are applied using the solution-aware DFD notation proposed by Sion et al. [7]. For our proposal, we define a countermeasure as the application of one or more tactics in a specific model. Concretely, a countermeasure is defined by (i) a name, (ii) the applied tactic(s), and (iii) a set of role bindings which, for each of the roles mentioned in the applied tactics' descriptions, specify the DFD element fulfilling that role. Figure 4 describes a countermeasure which applies the audit tactic to the DFD shown in Fig. 2.

```
Name:  Client request logging
Applied tactics: Audit
Role bindings:
   - Logged event     : df1
   - Logged data      : request data
   - Log database     : Audit Logs
   - Protected entity: Server
   - Logging process : Server
```

Figure 4. A countermeasure describing how the the audit tactic (Fig. 3) is applied to Fig. 2.

Based on this information, tool support [9] can automatically mark *Repudiation* threats on the *Server* as mitigated, as the *Server* fulfills the role of *Protected entity*, which, as described in the audit tactic (Fig. 3), is protected against *Repudiation* threats. Our approach build on this by allowing tool support to also identify design flaws in the countermeasure itself, as will be explained next.

2.2. Security design flaw queries

Based on the generic and structured tactic descriptions (Section 2.1.2), queries can be composed for common design flaws related to the tactics. Similar to the tactic descriptions, queries are also described generically so that they can be applied to any system model. A common flaw

for the audit tactic is, for example, that the data written to the logs is not sanitized, as described by CWE-117.[3] Thus, if a countermeasure applies the audit tactic, and the data written to the logs is (or contains) the exact data contained in the logged flow, then CWE-117 should be elicited for that countermeasure. The query for this flaw is shown in Fig. 5.

```
query cwe-117{Solution S} {
  S applies tactic named 'Audit';
  S binds the 'Logged event' role to some data flow DF;
  DF is annotated with a data type X;
  S binds the 'Logged data' role to data type Y;
  Y == X, or Y includes X;
}
```

Figure 5. Pseudo code for the CWE-117 elicitation pattern.

2.3. Advantages

As described earlier, applying STRIDE to the DFD shown in Fig. 2 would result in several threats, for example a tampering threat on the *Audit Logs* or a repudiation threat on the *Server*. As the system model includes audit logs, the repudiation threat can be marked as mitigated. Traditional threat modeling approaches, however, provide no further guidance on how to systematically evaluate whether the logging solution is designed correctly. Sufficient security knowledge and manual effort is required to identify that (a) a tampering threat on the audit logs reduces the effectiveness of the logging solution, and (b) encryption is not sufficient to mitigate the tampering threat, even though it is a standard tactic for integrity. Indeed, the tampering threat may have been marked as mitigated if an encryption tactic was applied, but this does not prevent more complex issues such as CWE-117 (which can be categorized as a tampering threat).

In comparison, our proposal allows such issues to be identified systematically and automatically. Furthermore, tactic descriptions and flaw queries can be reused across models, thus limiting the security expertise required to apply our approach. Finally, the issues identified by our approach are tailored to the context, which is not the case for traditional approaches. For example, whereas a STRIDE analysis would simply flag an unmitigated tampering threat on the audit logs, our proposal enables automatically generating more detailed issue descriptions such as *"Request data provided by the client is directly written to the audit logs, which may allow attackers to forge log entries. As a result, the client request logging solution may not suffice to prevent repudiation threats. See* cwe-117 *for more information."* This allows to clearly trace the specific cause of the flaw, as well as its impact.

3. Discussion and Future Work

In summary, by extending a system model with information on how tactics are applied in a design, our approach allows common security design flaws to be identified systematically and automatically. Compared to traditional threat modeling approaches, where countermeasures are mostly modeled as booleans (present/absent), this reduces the time and effort needed to find common security flaws. Furthermore, the required security expertise is also reduced, as tactic descriptions and flaw queries are generic and reusable across system models.

To demonstrate the feasibility of our approach, a tool prototype was developed, as well as a number of tactic descriptions and flaw queries for common security design issues. Applying these queries to example models such as the running example did not reveal any issues. In future work, we aim to compose a more extensive catalog of tactics and common flaw queries based on existing knowledge. The Architectural Concepts view of the CWE[4] can serve as a starting point, as it contains a detailed collection of potential design flaws, organized by architectural security tactics to which they apply [12]. Furthermore, an evaluation is needed based on a concrete and realistic case.

References

[1] The OWASP Foundation, "OWASP Top 10 - 2021," https://owasp.org/Top10/, 2021.

[2] European Union, "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016," *Official Journal of the European Union*, vol. 59, no. L 119, pp. 1–88, May 2016.

[3] Z. Shi, K. Graffi, D. Starobinski, and N. Matyunin, "Threat modeling tools: A taxonomy," *IEEE Security & Privacy*, no. 01, pp. 2–13, dec 2021.

[4] B. J. Berger, K. Sohr, and R. Koschke, "Automatically Extracting Threats from Extended Data Flow Diagrams," in *Engineering Secure Software and Systems*, ser. Lecture Notes in Computer Science, J. Caballero, E. Bodden, and E. Athanasopoulos, Eds. Springer International Publishing, 2016, pp. 56–71.

[5] A. Shostack, *Threat Modeling: Designing for Security*, 1st ed., 2014.

[6] Microsoft Corporation. (2020) Microsoft threat modeling tool 7. [Online]. Available: https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool

[7] L. Sion, K. Yskout, D. Van Landuyt, and W. Joosen, "Solution-aware data flow diagrams for security threat modeling," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1425–1432. [Online]. Available: https://doi-org.kuleuven.e-bronnen.be/10.1145/3167132.3167285

[8] K. Tuma, L. Sion, R. Scandariato, and K. Yskout, "Automating the early detection of security design flaws," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 332–342. [Online]. Available: https://doi.org/10.1145/3365438.3410954

[9] L. Sion, D. Van Landuyt, K. Yskout, and W. Joosen, "Sparta: Security & privacy architecture through risk-driven threat assessment," in *International Conference on Software Architecture*. IEEE, 8 2018, pp. 89–92. [Online]. Available: https://lirias.kuleuven.be/1656829

[10] K. Tuma, R. Scandariato, M. Widman, and C. Sandberg, "Towards security threats that matter," in *Computer Security*. Springer International Publishing, 2018, pp. 47–62.

[11] A. van den Berghe, K. Yskout, and W. Joosen, "A reimagined catalogue of software security patterns," in *The 3rd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS'22)*. ACM, 2022.

[12] J. C. S. Santos, K. Tarrit, and M. Mirakhorli, "A catalog of security architecture weaknesses," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 220–223.

3. https://cwe.mitre.org/data/definitions/117.html

4. https://cwe.mitre.org/data/definitions/1008.html