An Anti-pattern for Misuse Cases

An Anti-Pattern for Misuse Cases

Mohammad Torabi Dashti[1] and Saša Radomirović[2]

[1] Department of Computer Science
ETH Zurich
mohammad.torabi@inf.ethz.ch
[2] School of Science and Engineering
University of Dundee
s.radomirovic@dundee.ac.uk

Abstract. Misuse case analysis is a method for the elicitation, documentation, and communication of security requirements. It builds upon the well-established use case analysis method and is one of the few existing techniques dedicated to security requirements engineering. We present an anti-pattern for applying misuse cases, dubbed "orphan misuses." Orphan misuse cases by and large ignore the system at hand, thus providing little insight into its security. Common symptoms include implementation-dependent threats and overly general, vacuous mitigations. We illustrate orphan misuse cases through examples, explain their negative consequences in detail, and give guidelines for avoiding them.

1 Introduction

Misuse case analysis is a method for helping requirements engineers with the notorious task of eliciting security requirements. The elicited requirements are documented textually, or as UML-inspired diagrams, to facilitate communication among business analysts, developers, project managers, and other stakeholders. Similarly to use cases, misuse cases can also form a basis for estimating project cost and efforts. Other applications of misuse cases include documenting the provenance of security functionalities and enabling security testing and risk analysis; see, for example, [16, 17].

To carry out misuse case analysis, first, a system's functional requirements are elicited as a set of use cases. This step follows the well-established use case analysis method, extensively studied and applied in software engineering. Then, engineers consider each elicited functional use case, and investigate how an adversary might "misuse" it. What constitutes a misuse is determined by the security objectives that are, implicitly or explicitly, available to the engineers.

Finally, to mitigate the misuse cases obtained in the second step, new functional use cases are elicited. These are called security use cases. Optionally, the analysis loops back to the second step for considering threats against the newly added (security) use cases. Functional use cases, security and otherwise, are refined and implemented, whereas misuse cases are fictitious. We assume that the reader has a rudimentary understanding of use cases and misuse cases. For a detailed introduction to these methods see, for example, [1, 16].

Given a problem, an anti-pattern illustrates a recurring solution that has undesired consequences [7]. We present the *orphan misuses anti-pattern*, an anti-pattern for applying misuse case analysis. Intuitively, orphan misuse cases are "orphans" as they ignore the use cases elicited for the system at hand. Consequently, they provide little insight into the system's security: either the analysis prematurely ends with high-level objectives, or a number of well-known, code-level attacks are selected and listed. In both cases, the system at hand remains by and large unanalyzed. Common symptoms of orphan misuse cases include implementation-dependent threats, such as "buffer overflow," and mitigations, such as "prevent fraud," that pertain to (almost) any system and are hence vacuous.

Contributions. We define orphan misuse cases and explain their negative consequences in detail (§2). Afterward, we present the orphan misuses anti-pattern (§3) and illustrate it through examples (§4). We discuss how to avoid orphan misuse cases and give recommendations for writing effective misuse cases (§5).

2 Orphan Misuse Cases

Any attack can be trivially represented as a misuse case: draw a circle, write the name of the attack inside it, and call it a misuse case. Any preventive or prohibitive mitigation mechanism can similarly be represented as a security use case. This trivial expressiveness comes at a cost: requirements engineers, engaged in misuse case analysis, are left with no guidelines. They are supposed to imagine all possible attacks. Consequently, analysis paralysis may ensue: when imagination is set loose, the outcome is paralyzed because of the undue amount of time and energy that is spent on analysis.

Eliciting security requirements needs a cognitive catalyst: a starting point for thinking about possible attacks. In misuse case analysis, the starting point is a (functional) use case that is already elicited through use case analysis. Focusing on elicited use cases guides the thoughts and limits the search space hence discouraging analysis paralysis. We illustrate this point with a simple example.

Example 1. A "register new clients" use case has been elicited for a web site, associated to a political party, see Figure 1. The use case includes a mechanism for preventing two clients from having the same email address: trying to register using an email address that exists in the system leads to an an error, signaling that the address cannot be reused.

Any number of attacks are imaginable on this system, ranging from command injection to kidnapping system administrators and coercing them into collaboration. The details are also as varied as the imagination permits.

But, focusing only on the logic of this use case brings to light the fact that an adversary can find out, within a margin of error, whether a certain person has an account on the system. Suppose the adversary tries to register using an email address that belongs to X, a public figure, and receives an error message,

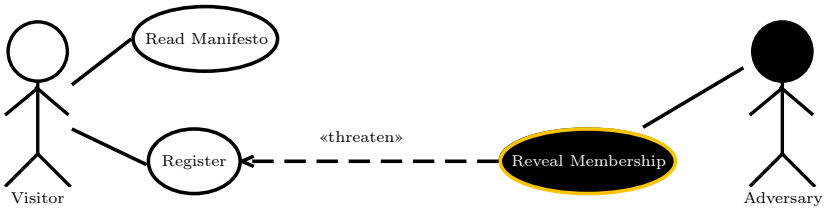

Fig. 1. Analysis for Example 1, drawn with SeaMonster Security Modeling Software

indicating that the address cannot be reused. Then, the adversary can infer that X has an account on the system and is likely sympathetic to the party. This violates the (implicit) security objective that a person's membership in the web site, which betrays a certain political inclination, must not be revealed through the system. This misuse case is denoted "reveal membership" in Figure 1. △

There is a difference between sabotaging a use case and misusing it. For example, command injection attacks hardly constitute a misuse case in the above scenario, because of two reasons. First, a large number of use cases can be attacked through command injection, if their implementation is not based on safe languages. In this sense, command injection has little to do with the specific use case of registering new clients. Therefore, if command injection attacks are to be considered as misuse cases, a myriad of other code-level attacks, including more exotic ones like "row hammering," [6] should also be included, making the elicitation process practically intractable. We return to code-level threats and their mitigation techniques in §5.

The second reason is that command injection is not inherently part of, say, a pen-and-paper procedure. A stored procedure on a SQL database might however suffer from it. While the requirements should not assume a particular implementation, command injection presumes a fixed (sort of) implementation. A similar argument shows that kidnapping system administrators is also hardly a misuse of the "register new clients" use case. Considering this type of attack does not provide insight into a problem that the application's architecture suffers from. In contrast, the "reveal membership" misuse case is inherent to the registration use case with the duplicate detection mechanism described in Example 1. It is a problem that demonstrates how the elicited functionalities can be misused and thus provides insight into a problem that exists at this level of abstraction.

In UML nomenclature, a misuse case amounts to an undesirable use case that "includes" a desired use case, where to include a use case roughly means using it as a subroutine; see [9]. This conforms to the original rationale of Sindre and Opdahl: "many threats to a system can largely be achieved by using that system's normal functionality" [15]. We call a misuse case that does not "include" one or more functional use cases an **orphan** misuse case. Command injection and kidnapping system administrators in the above scenario are examples of orphan misuse cases.

We conclude this section with a side note. Security objectives are not elicited through misuse cases. They must be present, implicitly or explicitly, to determine what a misuse is. Therefore, they are not the (main) outcome of a misuse case analysis. What is elicited through misuse cases is the security functions a system must have; see, e.g., [2]. These functions, i.e. security use cases, in effect constrain the system's other use cases. Referring to these as "security requirements" might be slightly misleading, but it is well-accepted in the literature. For example, Haley, Laney, Moffett, and Nuseibeh define a security requirement as a constraint on system functions [5].

3 Orphan Misuses Anti-Pattern

We argue against writing orphan misuse cases.

Orphan misuse cases are not tied to the functional use cases that have been elicited for the system at hand. Therefore, they tend to be either overly specific, pertaining to a fixed implementation technology, or overly general, bordering on high-level objectives rather than functions. We illustrate these points through examples from the literature.

Implementation-dependent orphan misuse cases. Sindre and Opdahl give "get privileges" as a misuse case for "register customer" use case [16]. Peterson and Steven give "inject commands" as a misuse case for "review account" in a banking system [12]. Rostad gives "overflow attack" as a misuse for "enter user name" use case, which is mitigated by "input validation" [14].

These examples mix up the abstraction levels: a particular implementation technology is presumed at the requirements elicitation phase. Moreover, "eliciting" code-level orphan misuse cases is a rather futile exercise: they appear to be *recalled* from a list of generic attacks, e.g. OWASP's top ten or CWE's top twenty five, rather than being actually *elicited* for the system at hand. This is not unexpected: in the requirements elicitation phase, often no implementation is available. Therefore, the attacks cannot be about an implementation's peculiarities. This signals the occurrence of analysis paralysis: the engineers, not knowing what to think of, fall back on the well-known attacks, ignoring the elicited use cases.

Note that we do not argue against considering wide-spread, code-level attacks. Rather, the point is that when misuse cases for a given system are elicited, there is hardly any value in thinking about generic well-known attacks. By definition, they are all known, and listed elsewhere. We argue that by shifting the focus towards the use cases at hand, genuine misuse cases can be elicited and specific mitigation mechanisms can be devised and documented as security use cases; see §4. Moreover, these steps are guided by elicited use cases, which help with analysis paralysis.

Overly general orphan misuse cases. Regev, Alexander, and Wegmann give "launder money" as a misuse case for "establish reputation," which is mitigated

Table 1. Orphan Misuses Anti-Pattern

Problem. Elicitation of security requirements through misuse case analysis.
Recurrent Solution. Orphan misuse cases which do not include any use case.
The Solution's Negative Consequences. Orphan misuse cases likely ignore the use cases elicited for the system at hand. They often hide the underlying trade-offs between security and cost, efficiency, usability, and so forth, and provide little insight: either the analysis ends prematurely at the objectives level, or a number of well-known code-level attacks are selected and listed. In both cases, the system at hand is by and large ignored.
Common Symptoms. Implementation-dependent misuse cases, such as "buffer overflow." Misuse cases that categorically threaten (almost) any use case. Mitigations amounting to vacuous objectives, of the form "secure it!"
How to Avoid It. Follow the logic of use cases and think about how they, as given, can be misused. Let security objectives inform the elicitation process.

by "check money laundering," in banking [13]. Pauli and Xu give "impersonate user" as a misuse case for "enter appointment," which is mitigated by "recognize user," in a health-care system [11]. Lehtonen, Michahelles, Fleisch give "theft from internal IT" as a misuse case for "tag authentication," which is mitigated by "secure internal IT system," in the RFID context [8].

In the examples above, the analysis has terminated prematurely. For instance, "check money laundering" is a high-level objective, as opposed to a functional (security) use case. Overly general orphan misuse cases, and their mitigation, by and large ignore the specific use case under study. We of course do not dismiss the value of high-level objectives. The problem lies with premature termination of analysis: these objectives need further refinements through settling a range of issues regarding adversarial capabilities and the value of the protected assets. We return to this point in §4.

We define the **orphan misuses anti-pattern** in Table 1. Next, we illustrate how orphan misuse cases can be avoided.

4 Analysis without Orphan Misuse Cases

Below, we start with an elicited use case and work out how it can be misused. Mitigating the elicited threats tends to be more complex than recalling well-known security solutions. It often raises fundamental questions about the presumed adversarial capabilities, the expected level of security and its cost. These issues are inherent to security engineering, and in practice they cannot be resolved by requirements engineers alone. Communication among engineers and other stake-holders is necessary. Compromises are often inevitable, but informed discussions shed light on what is lost for which gain.

Example 2. A news service has two types of users: reporters and visitors. Visitors may read the news, post comments on the news, and read the comments. Reporters inherit visitors' rights and may also post news items. Obviously, to mitigate fake news, the "post news" use case must include an authentication (security) use case, see Figure 2.

Rather than thinking about arbitrary attacks that might threaten the system, we focus on the logic of the elicited use cases. We identify "troll" as a misuse of the "post comments" use case. Trolling violates the (implicit security) objective stating that the service's users must be able to share their opinions safely.

Suppose we identify two possible mitigations: (1) users must authenticate to post a comment. (2) Comments are moderated by reporters before becoming public. The first mitigation is prohibitive: a troll can eventually be identified and banned. Of course, it is possible that trolls create new accounts to escape the ban, and also it is not clear how a set of comments is decided to be inappropriate.

The second mitigation, which is preventive, may unduly increase the reporters' work load. Moreover, it enables censorship: rogue reporters can misuse the "moderate comments" use case to silence the commentators who challenge their views. This violates the security objective above. To mitigate the "censor" misuse case, a separation of powers mechanism can prevent reporters from moderating comments on their own posts. A preliminary analysis, omitting the censor misuse case, is shown in Figure 2.

Which mitigation should be adopted? This question cannot be answered by requirements engineers alone. The options, including the cost of enforcing separation of powers, must be communicated to the stake-holders. This facilitates informed discussions for deciding a specific mitigation. △

Misuse case elicitation in Example 2 is focused on the news service, but it is not about well-known, implementation-dependent attacks. These attacks are important issues in practice, but not in the requirements elicitation phase. Moreover, until we get the requirements right, and architect the system accordingly, there is little hope for securing the system even if all code-level attacks are accounted for.

Example 3. In a perimeter control system, the "unlock all doors" use case has been elicited following fire safety regulations: in case of fire all doors must be unlocked. Obviously, this use case should not be publicly accessible. An authentication and authorization (security) use case is due: to activate the use case, one must have a certain role in the organization. This step is rather obvious. The interesting question is what to do about sensitive areas that are left unprotected in case of fire.

Suppose there is a safe in the building. An arsonist adversary might start a fire to ease his/her access to the safe. Suppose we identify two mitigations: (1) a surveillance system, resilient to fire, monitors the safe and the paths that lead to it. (2) In case of fire, aqueous foam is dispensed, which quickly expands and fills the area around the safe, hence delaying the adversary's access [4].

The first mitigation is prohibitive and the second one preventive, but it comes with high maintenance costs. Which one should we choose? Again, the decision

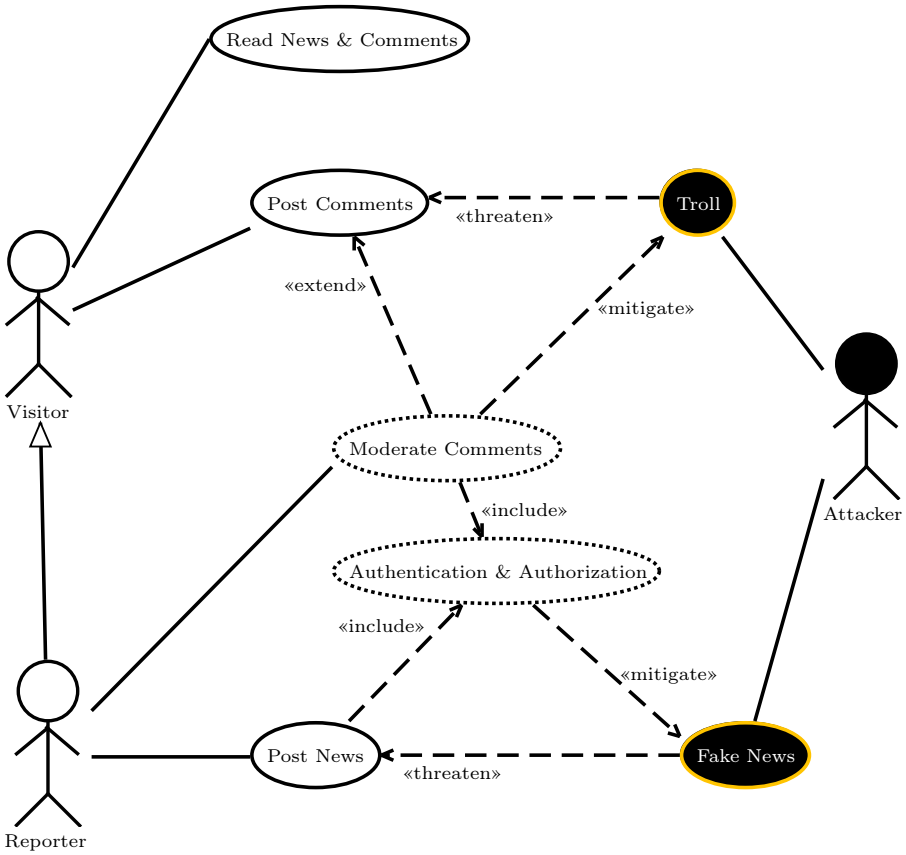

Fig. 2. Analysis for Example 2

is not for requirements engineers to make. The presumed capabilities of the adversary against whom the perimeter is protected, the value of the asset, i.e. the safe, and technological constraints, e.g. the availability and dependability of foam dispensing equipments and their suppliers, are among the questions that must be answered before choosing a concrete mitigation.

Continuing with the misuse case analysis reveals that the first mitigation is privacy-intrusive, e.g. surveillance cameras can be misused by an insider to track a person's movements. These findings help the stake-holders to make informed decisions. △

In Example 3, it is necessary to make risks observable through, e.g., clarifying adversarial capabilities, and evaluating the worth of protected assets. This is because the focus is on a concrete system. In contrast, an orphan misuse case, which is divorced from the system, does not raise these issues naturally. Writing high-level objectives such as "mitigate arson" is trivial, but without addressing

the above issues regarding adversaries, assets, and so forth, this mitigation is vacuous, hence of little use to the developers and other stake-holders.

Example 1's misuse case can be subjected to a similar analysis for finding suitable mitigations. For instance, strengthening the error handling mechanism might appear promising. We do this next.

Example 4 (Continuing Example 1). Consider the register new clients use case. We need a mechanism that prevents two clients from registering the same email address. As discussed, if the mechanism displays an error message on the same channel used to enter email addresses, then an adversary can infer that the owner of an email address has an account on the system. Suppose we change the mechanism to include a confirmation loop, as explained below.

1. When an email address is provided in the registration step, a message is sent to that email address, its contents determined as follows.

2a. If the email address is already registered in the system the recipient is informed that someone tried to register with this address. The recipient is thus reminded that they do have an account.

2b. If the email address is not registered, a link is provided to continue with the registration.

This prevents the "reveal membership" misuse case as the adversary cannot differentiate between the two types of email that are sent to an address that he does not own.

Further extending the analysis can now produce a new misuse case where an attacker is able to misuse the confirmation loop to send unsolicited emails (spam) to arbitrary email addresses. To combat spamming, the number of emails sent to any given email address must be controlled. Clearly, this mitigation comes at the cost of a more complex, stateful error handling mechanism. The resulting diagram is shown in Figure 3. △

We conclude this section by remarking that examples where the elicitation of misuse cases follows the available use cases as a guideline are also found in the literature. The information misuse case given in Example 1 is based on an example of Sindre and Opdahl [16]. OWASP's Testing Guide gives "brute force," i.e. repetitive password guessing, as a misuse of the "authentication" use case, which is mitigated by various checks added to the use case [10]. This too follows the guideline.

5 How To Avoid Orphan Misuse Cases

Writing orphan misuse cases is trivial. High-level misuse cases always amount to the same thing: "sabotage it!" Code-level misuse cases are also not hard to come by. Looking up any of the existing top ten or top twenty five vulnerability lists is a good start. We dismiss such misuse cases as products of an anti-pattern. Since they imitate the elicitation process but do not contribute to it, the system at hand remains unanalyzed.

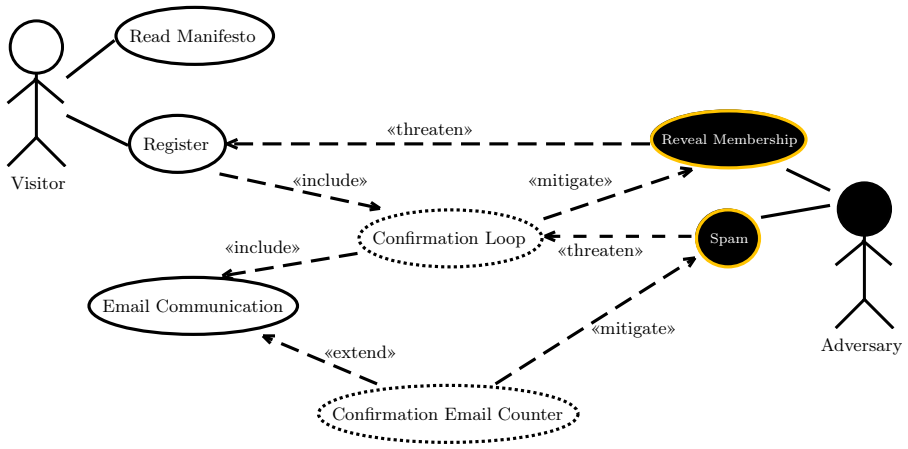

Fig. 3. Analysis for Example 4

We advocate writing misuse cases that are specific to the system under study. This is hard: it demands new thinking and fresh perspectives for each case study. The result is however of higher quality as it is coupled with the system at hand, and moreover the process encourages thinking about fundamental issues, such as the presumed adversarial capabilities and cost-benefit analysis.

Clearly avoiding orphan misuse cases means that not all attacks can be accounted for in misuse case analyses. This is a limitation, but, we argue, a desirable one:

The point of misuse cases is to elicit the security functions that a system must have. Therefore, a successful analysis, no matter how unattainable, is one that covers all necessary security functions for a given system, in light of the adversarial capabilities, cost-benefit analysis, and other forms of compromise inherent to usable, practical security. These security functions, similarly to other functional use cases, are likely to be flawed when implemented. But, there is little we can do about it at the requirements elicitation phase: it is unreasonable to expect misuse cases to account for all security issues. Combating code-level vulnerabilities that could lead, for instance, to malicious code-injection has its own dedicated techniques, such as testing and code inspection, which fall outside the domain of requirements engineering per se.

Pushing every concern to the requirements elicitation phase is impractical for at least two reasons. First, analysis paralysis is real: the list of attacks is virtually inexhaustible. A comparison to chess is instructive here. Once in a while a grand master comes up with a brilliant new attack (or defense) strategy that has eluded thousands of people who played and studied chess for centuries. Then, it should not be surprising that we cannot foresee all attacks against industrial-scale computer systems: these systems are substantially more complex and more opaque than chess.

Second, and more importantly for our argument, genuine security issues, with architectural implications, can be discovered and mitigated at the requirements level, only if we shift the focus towards them. We claim that such issues can be discovered more effectively, and with less cluttering, when orphan misuse cases are avoided. We briefly illustrate this on the application programming interfaces (APIs) that a mobile operating system exposes to application developers. Each API corresponds to a functionality. For example, the API for microphone access enables an application to record sound. This functionality can be misused to undermine privacy objectives and this misuse case can be mitigated by a functionality for the user to disable an application's access to microphone. The sheer number of APIs that an operating system exposes makes the security analysis at this level alone very complex. Security issues due to API misuse, uncovered for example in [3], support this claim.

We now summarize our discussions with five simple, rule-of-thumb guidelines for avoiding orphan misuse cases.

Think of include A misuse case "threatens" a use case, in the same way one use case "includes" another one. Revise the misuse cases that do not include any use cases and those that categorically "threaten" (almost) any use case.

Go bottom-up Start with a use case and work out its misuses. Do not start with an adversary with the intention of brainstorming the ways it can attack the system. There are too many ways, hence leading to analysis paralysis.

Respect the abstraction level If functional use cases are at the requirements level, misuse cases should not belong to the objectives level, nor should they encroach on implementation details.

Make risks observable If issues such as adversarial capabilities, value of the protected assets, and cost-benefit compromises do not cross your mind, you are likely ignoring the system at hand. Similarly, decisions that can be made without consulting stake-holders are likely the trivial ones. Make risks observable by explicating underlying assumptions and trade-offs.

Dismiss trivial expressiveness Anything can be written as a misuse case. However, requirements representation is not the same as requirements elicitation. As a representation tool, misuse cases can document any requirement. But, as an elicitation tool, misuse case analysis must be guided by use cases.

We conclude the paper with a note on empirical validation. To empirically study the value of the rule-of-thumb guidelines and the orphan misuses anti-pattern, one must raise one's sights to look beyond symptoms and target root causes as well. Namely, for each flaw found in a system, one must look beyond its immediate source, and identify the reason, i.e. deficiencies in the requirements engineering phase (and other system development phases) that have led to the flaw. The value of this form of root-cause analysis is well-known; see, for example, Van Vleck's *three questions* [18].

An industrial-scale root-cause analysis, for the purpose of evaluating our requirements elicitation approach and the misuse case analysis method in general, demands substantial efforts and is left for future work. An observation we have made in our lectures is nonetheless illustrative: students stop throwing arbitrary

attack names, after we present them with the constraint that a misuse case must include a functional use case as given. Reciting the latest hacker news item becomes irrelevant. Creative thinking, guided by use cases, takes its place.

References

1. Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2001.
2. Donald Firesmith. Security use cases. *Journal of Object Technology*, 2(3):53–64, May-June 2003.
3. Yanick Fratantonio, Chenxiong Qian, Simon P. Chung, and Wenke Lee. Cloak and dagger: From two permissions to complete control of the UI feedback loop. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 1041–1057. IEEE Computer Society, 2017.
4. Mary Lynn Garcia. *The Design and Evaluation of Physical Protection Systems.* Elsevier Science, 2001.
5. C. Haley, R. Laney, J. Moffett, and B. Nuseibeh. Security requirements engineering: A framework for representation and analysis. *IEEE Transactions on Software Engineering*, 34(1):133–153, Jan 2008.
6. Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA*, pages 361–372. IEEE Computer Society, 2014.
7. Andrew Koenig. Patterns and antipatterns. *JOOP*, 8(1):46–48, 1995.
8. M. O. Lehtonen, F. Michahelles, and E. Fleisch. Trust and security in RFID-based product authentication systems. *IEEE Systems Journal*, 1(2):129–144, Dec 2007.
9. Object Management Group. Unified modeling language (OMG UML), version 2.5, 2015.
10. OWASP. Testing guide v. 4, Accessed April 2016. https://www.owasp.org.
11. Joshua J. Pauli and Dianxiang Xu. Misuse case-based design and analysis of secure software architecture. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, ITCC '05, pages 398–403. IEEE Computer Society, 2005.
12. Gunnar Peterson and John Steven. Defining misuse within the development process. *IEEE Security and Privacy*, 4(6):81–84, November 2006.
13. Gil Regev, Ian F. Alexander, and Alain Wegmann. Modelling the regulative role of business processes with use and misuse cases. *Business Process Management Journal*, 11(6):695–708, 2005.
14. L. Rostad. An extended misuse case notation: Including vulnerabilities and the insider threat. In *Working Conf. Requirements Eng.: Foundation for Software Quality (RREFSQ)*, pages 33–34. Essener Informatik Beitrage, 2006.
15. G. Sindre and A. L. Opdahl. Eliciting security requirements by misuse cases. In *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-Pacific 2000*, pages 120–131, 2000.
16. Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.
17. K. R. van Wyk and G. McGraw. Bridging the gap between software development and information security. *IEEE Security Privacy*, 3(5):75–79, Sept 2005.
18. Tom Van Vleck. Three questions about each bug you find. *ACM SIGSOFT Software Engineering Notes*, 14(5):62–63, July 1989.