# A Reuse-Based Approach to Determining Security Requirements

Guttorm Sindre[1], Donald G. Firesmith[2], Andreas L. Opdahl[3]

[1] Dept Computer & Info. Science, Norwegian U. Science & Technology
(on leave at Dept MSIS, Univ. Auckland, New Zealand)
[2] Software Engineering Institute
[3] Dept of Information Science, U. of Bergen, Norway

**Abstract.** The paper proposes a reuse-based approach to determining security requirements. Development *for* reuse involves identifying security threats and associated security requirements during application development and abstracting them into a repository of generic threats — expressed as misue cases — and requirements — expressed as security use cases. Development *with* reuse involves identifying security assets, setting security goals for each asset, identifying threats to each goal, analysing risks and determining security requirements, based on reuse of generic threats and requirements from the repository. Advantages of the proposed approach include building and managing security knowledge through the shared repository, assuring the quality of security work by reuse, avoiding overspecification and premature design decisions by reuse at the generic level and focussing on security early in the requirements stage of development.

**Keywords:** Security requirements, security threats, reuse, requirements engineering, security, use cases, misuse cases

## 1 Introduction

*Use cases* [1-3] have become popular for eliciting requirements [4, 5]. Many groups of stakeholders turn out to be more comfortable with descriptions of operational activity paths than with declarative specifications of software requirements [6]. As use cases specifically address what users can do with the system, they are most relevant for functional requirements. But lately the application of use cases has also been investigated in connection with security and safety requirements, in the form of *misuse cases* [7-13], a.k.a. *abuse cases* [12, 14]. Misuse cases have received growing research interest during the last 3-4 years, as an informal front-end method to eliciting security requirements. Previous work on misuse cases has mostly been application-specific. This paper proposes an additional, more abstract level of generic, application-independent misuse cases, which can be stored in a repository for reuse. In order to avoid premature design decisions, generic misuse cases are written on the requirements level,. Hence, such a repository could be highly reusable across development projects and help requirements workers get a flying start in expressing their security requirements, as well as contributing to the security education of both developers and other stakeholders.

These describe interactions that cause harm to the system or its stakeholders and can be used as an informal front-end to more formal security systems engineering. Like use cases they have the advantage that more stakeholders can be involved. A closely related topic of research is that of security use cases [15]. Like misuse cases these describe interaction sequences where harm is attempted, but unlike misuse cases, the system ends up preventing or at least mitigating the damage.

In spite of the growing research interest in misuse cases, and promising early applications [10, 11, 13], the approach has yet to be put into large-scale industrial use. Many software development organizations tend to put little focus on security requirements, even if these are

increasing in importance [16]. Partly, this may be due to a lacking understanding of security requirements. Indeed, even when they attempt to write security requirements, many developers tend instead to describe design solutions in terms of protection mechanisms, rather than making declarative statements about the degree of protection required [17]. Another reason for neglecting security requirements may be a perceived shortage of time in projects with narrow deadlines. For instance, case studies [18, 19] showed that security requirements were poorly addressed in several e-commerce projects. E-commerce is even a domain where security would be of major importance – but also a domain where short lead-times may be crucial.

To make misuse case analysis more appealing to practitioners *reuse* may be essential – as security requirements could then be specified more rapidly. As pointed out already in the 80's, reuse of requirements could lead to significant savings in development time and cost [20]. Although requirements reuse has attracted much research attention since then, e.g., [21-31], its application in the software industry is limited. Much more reuse has been achieved for patterns and frameworks on the design level.

Requirements may be difficult to reuse because they are (at least partly) different from project to project. But for security requirements (or the preceding description of threats, as captured by misuse cases) reuse might be *easier* to achieve than for functional requirements – which has been the main focus in much earlier research. While functional requirements will vary from domain to domain, and even significantly from application to application within a domain, the general threats that systems face – such as spoofing of identity, tampering with data, information disclosure, denial of service – will tend to be fairly similar from system to system. For instance, while there are huge differences between missile control systems and e-shops, so that there is little hope of reusing functional requirements between these domains, both will face the threat that a misuser may spoof a legitimate user to gain access to system services. And while stronger security would probably be needed for the missile control system than the e-shop, this might just be a matter of *degree* – both kinds of systems would have security requirements in the same categories, e.g., identification, authentication, authorization, integrity. A similar case is made for the closely related area of *legal requirements* in [32]: Whatever the application domain, one has to satisfy the same or similar laws, for instance about privacy. Hence legal requirements may lend themselves better to reuse than functional requirements.

The purpose of this paper is to provide a reuse-based methodology for misuse case analysis and the subsequent specification of security requirements. There are two key processes in reuse-oriented development [33, 34]:

- Development <u>for</u> reuse, where reusable artifacts are developed and made available for future reuse, for instance in a repository / repository that facilitates easy retrieval.
- Development <u>with</u> reuse, where end-user applications are developed, partly by reusing artifacts created by the "for" process.

There are of course interconnections between these two processes (e.g., development with reuse can discover weaknesses of existing components in the reuse repository, or inspire new ideas for reusable components; and development for reuse can steer development with reuse to the extent that you are able to choose projects – i.e., pick the one where you have the greatest possibility of achieving reuse based on your current repository contents). The rest of this paper is structured as follows: Section 2 deals with development <u>for</u> reuse, discussing what kinds of artifacts should be developed, how the reusability of these artifacts should be ensured, and how the artifacts should be packaged in a repository for future reuse. Section 3 then addresses development <u>with</u> reuse, discussing how to identify candidates for reuse and then adapt them to the specific application. Section 4 discusses the suggested approach, comparing it to related work, and section 5 concludes the paper.

# 2 Development *for* Reuse

Reuse of systems development artefacts may improve the quality of development processes and products and may reduce development costs if each artefact is reused at least 3–4 times (because it is more expensive to develop something reusable than something which will be used only once [35].) To ensure repeated reuse of security threats and requirements, we must find good answers to the following questions:

1. Which development artefacts should be stored in the repository for reuse?
2. How should the repository be organised to best support reuse?

## 2.1 The reusable development artifacts

As mentioned in the Introduction, applications are likely to face the same kinds of threats and have similar categories of required security even if they have different functional requirements. The challenge for reusability will be to describe threats and requirements on a sufficiently generic level, so that detailed differences between applications (e.g., in functionality, architecture) do not hamper the possibility for reuse. We suggest two main kinds of reusable artifacts:

- *Generic threats*, represented as misuse cases in this paper.
- *Generic security requirements*, represented as security use cases in this paper. .

For an example of a generic threat, Table 1 shows a generic misuse case that represents the threat of *spoofing*, i.e., a miuser gaining access to the system by pretending to be a regular user. This is a highly reusable miuse use cases, which covers many different spoofing attacks. It does not matter if authentication is done by username+password, card+PIN, fingerprint scan, voice recognition, human to human recognition of individuals or something else. The interaction sequence is inspired by *essential misuse cases* [2], which focus on the users' intentions rather than concrete actions. In [2] the main motivation for this is to simplify the interaction and avoid premature design decisions. This will also increase the reusability of the description (as premature design would limit the reusability of a description only to those projects who were actually happy with those particular design decisions).

**Table 1:** Generic misuse case "Spoof User Access"

| **Generic Misuse Case:** Spoof User Access | |
|---|---|
| **Summary:** The misuser successfully makes the system (physical, human or computerized) believe he is a legitimate user, and thus gains access to a restricted system / service / resource / building. | |
| **Preconditions:**<br>1) The misuser has a legitimate user's valid means to identify and authenticate ***OR***<br>**2)** The misuser has invalid means to identify and authenticate, but so similar to valid means that the system is unable to distinguish. (Even if the system is operating at its normal capabilities) ***OR***<br>**3)** The system is corrupted to accept means of identification and authentication that would normally have been rejected. Here, the misuser may previously have performed a misuse case "Tamper with system", so the system does not operate at its normal capabilities. | |
| **Misuser interactions** | **System interactions** |
| Request access / service | |
| | Request identification and authentication |
| Misidentify and misauthenticate | |
| | Grant access / provide service |
| **Postconditions:**<br>1) The misuser can do anything the legitimate user could have done within one access session ***AND***<br>2) In the system's log (if any), it will appear that the system was accessed by the legitimate user. | |

For an example of a generic requirement, Table 2 shows one path of the security use case "Access Control", more specifically the one requiring the system to reject misusers with valid means of identification but invalid means of authentication. The entire security use case would be comprised of several paths, depending on various preconditions (e.g., misuser has neither valid identity nor authentication, misuser has valid identity but not valid authentication, misuser has both valid identity and valid authentication). Security

requirements can also be described as requirement lists or as mitigation points in misuse cases, although that is not discussed further here.

Notice also that in spite of having several paths, one security use case may not accommodate all possible requirements meant to mitigate a certain threat. The threat represented by the misuse case in Table 1 may in part be mitigated before or after the misuser's attempt to gain access to the system, rather than during the attempt itself. For instance, there may be requirements towards the means of authentication (for instance in terms of low "stealability", preferably quantified), for a cancellation service if means of authentication are feared compromised, or to reduce the authority of legitimate users as much as possible (to reduce the damage if defenses fail and the misuser indeed gains access to the system). Some of these may be described by other security use cases (e.g., "Cancel means of authentication"), while others may best be described by "shall"-requirements. For instance, it would be difficult to formulate a quantified requirement on the stealability of the means of authentication as a use case.

**Table 2:** One path of the generic security use case "Access Control"

| **Generic Security Use Case:** Access Control | | |
|---|---|---|
| **Path name:** Reject invalid authentication | | |
| **Preconditions:**<br>Misuser has valid means of user identification but invalid means of user authentication. | | |
| **Misuser Interactions** | **System Requirements** | |
| | **System Interactions** | **System Actions** |
| | Request user identity and authentication. | |
| Provide valid user identify but invalid user authentication. | | |
| | Reject misuser by cancelling transaction. | Attempt identification, authentication, and authorization. |
| **Postconditions:**<br>1) Misuser has valid means of user identification but invalid means of user authentication ***AND***<br>2) Misuser not authenticated and not granted access ***AND***<br>3) Access control failure registered. | | |

Just like the generic misuse cases, this generic security use case is highly reusable – it makes no design assumptions, and neither does it presuppose any particular application domain. Access control is a feature wanted in a wide range of applications. The above use case could be a representative requirement for accessing an ATM, an internet entertainment service, or a missile control system.

## 2.2 The organisation of the repository

A meta-model showing Threats and Security Requirements and the links between them is given in **Figure 1**. Threats are what misusers try to achieve (i.e., causing harm to the system), and security requirements describe the extent to which the system shall be able to mitigate those threats. Key to understanding the diagram are the two classes on the way from Threat to Security Requirement, namely Threat-Requirement Relationship and Security Requirement Bundle. To start with the latter, this is a set of requirements that pull together in mitigating the same threat. It is often interesting to look at such bundles rather than just individual requirements, because:

- A security requirement bundle is a bigger and more effective unit of reuse. To the extent that one requirement is a good unit of reuse, it is still possible to define a bundle consisting only of that requirement.
- In many cases, single security requirements provide little or no protection unless accompanied by other requirements. For instance, as observed in [17]

identification requirements are seldom of much value alone – in most cases they must be accompanied by authentication requirements.

Indeed, it can be observed that a security use case is in itself a requirement bundle. The example in Table 2 already contains two requirements – a) that the system shall reject access to users without valid means of authentication, and b) that failed access attempts shall be registered. And this is only one path of the bigger use case, so in total it would encompass several requirements.

The Threat-Security link objects will most commonly represent "mitigate" relationships, i.e., that a certain requirement bundle (e.g., the security use case "Access Control") mitigates a threat (e.g., the misuse case "Spoof User Access"). Another possible relationship is "aggravate", i.e., the choice of a requirements bundle may actually increase the risk for a threat. An example is that a bundle of Access Control requirements might increase the risk for Denial of Service (DoS) threats. A classical example is the suspension of console login for a certain user after three failed login attempts – a misuser could then deny access for that user simply by making those three failed login attempts. In more general terms, any requirement that the system should suspend access if sensing an attempted attack might be utilized for DoS purposes.

The Threat-Security Relationship could instead have been modeled as an M:N association (or two M:N associations, one for "mitigate" and one for "aggravate"). However, objectifying these relationships give increased flexibility:

- New relationships (in addition to mitigate and aggravate) can be defined without changing the meta-model
- It is easier to include attributes providing extra information about each relationship instance, for instance some explanations (why or to what extent a certain requirement bundle mitigates or aggravates a certain threat).

Notice that cardinalities into Threat-Requirement Relationship are * on both sides, i.e., there can be several Security Requirement Bundles mitigating the same threat, and one bundle may also contribute to mitigating several threats (for instance, any mitigation of a Spoof User

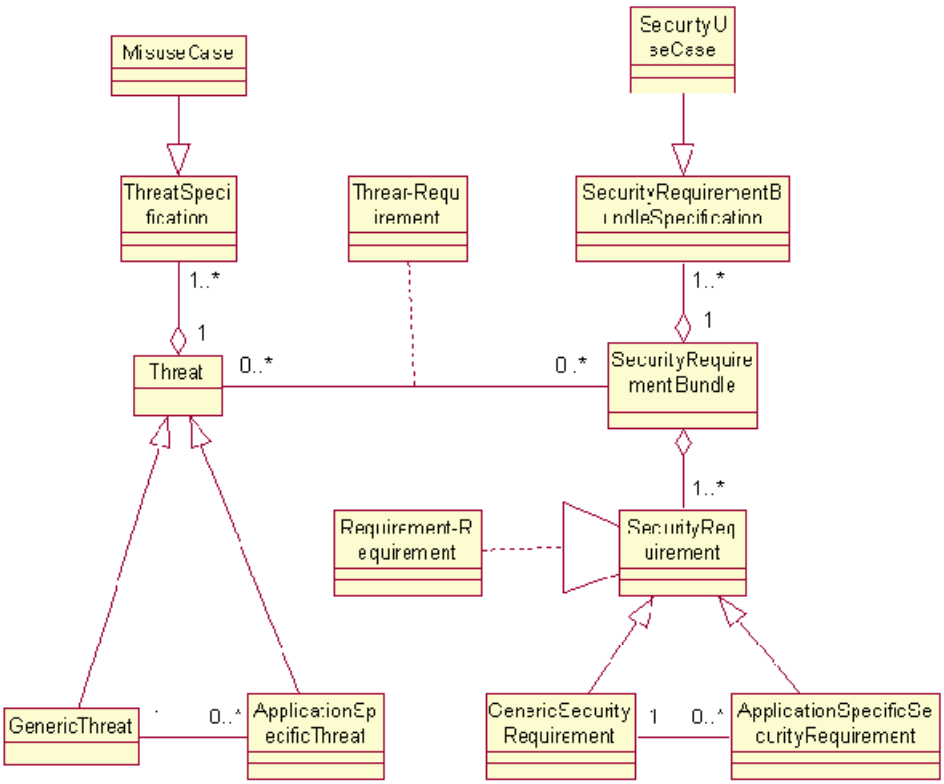


**Figure 1:** Metamodel for threats and security requirements

threat will also serve to mitigate a Repudiation threat).

The Requirement-Requirement Relationship is used to register relationships between requirements, e.g., that they may be overlapping or in conflict. Again, objectification gives somewhat increased flexibility wrt. the possibility for extending the repository with new kinds of relationships, as well as including extra information such as rationale / explanations.

The aggregation from Threat to Threat Representation enables one threat to have several parallel representations in terms of format or language. For instance, the same misuse case could be written both in English, French and Norwegian, or in the same language but with different templates, or there could also be other representations than misuse cases, for instance in more formal languages (not investigated in this paper). Hence, there could be other subclasses to Threat Representation in addition to Misuse Case. The upper right part of the diagram shows an analogous modeling of the requirements side.

The lower left part shows that a Threat can either be a Generic Threat or an Application-Specific Threat, and one Generic Threat may have many Application-Specific instantiations, e.g., the "Spoof User Access" threat of Table 1 may be instantiated to cover illegitimate access to an ATM, a building, an internet entertainment service etc. The lower right part of the figure shows that the requirements side is structured accordingly.

Basing a reuse repository on the above meta-model, the following two advantages are achieved:

- Security requirements may be searchable *via threats* that they are meant to mitigate, rather than having to search for requirements "directly". The "direct" alternative is less useful here – to know what to look for, the developer must then have a pretty clear picture of the requirement already, which reduces the gain from reuse.
- Security requirements can be packaged in bundles that give meaningful protection against commonly seen threats. In most cases this should be more effective than reusing requirements one by one and then assembling them in meaningful bundles on a project-to-project basis.

Having observed these advantages we now turn to development with reuse.

# 3 Development *with* reuse

Figure 2 shoes a UML Activity Diagram that outlines our suggested approach to development with reuse. The steps are as follows:

1. ***Identify critical and/or vulnerable assets:*** Here one must identify all the critical and/or *vulnerable assets* in the enterprise. A vulnerable asset is either information or materials that the enterprise possesses, locations that the enterprise controls or activites that the enterprise performs.[1] The "and/or" should be noticed specifically. It is interesting to look at assets that are critical but not vulnerable because a) further scrutiny may reveal that they were only believed not to be vulnerable, and b) their vulnerability might increase in the future. It is also interesting to look at assets that are vulnerable but not critical, at least if they are of the kind that misusers may use as stepping stones to launch attacks on more critical resources. For example, a server that holds no critical information and runs no critical services might still be used as a as a zombie in an attack against other computing resources, perhaps also in other companies, causing badwill or even liability to the organization. Starting the security analysis with a focus on assets ensures that the final security requirements are anchored in the protection of materials, information, locations and activities that are of value to the enterprise.

[1] The most important assets of enterprises, the knowledge and skills of its workers, is not directly important in an ICT security context, as they are only vulnerable indirectly, through misuse of the other, more tangible assets.

2. ***Determine security goals for each asset:*** For each critical and/or vulnerable asset identified in step 1., select the appropriate security goals for the asset. A security goal is specified in terms of (1) *who* are the potential misusers, (2) the *type* of security breaches the asset is vulnerable to and (3) the security *level* necessary for that type of breach. For example, the potential misusers may be Internet script kiddies, business competitors or disgruntled employees. Examples of security types are violations of, e.g., secrecy or integrity, and several of the security threat classifications in the literature can be used in this step. A possible taxonomy of security breaches is proposed in [17]. The security level to be achieved is specified as a probability that the assets will be kept safe from the particular type of breach from the particular type of misuser. Establishing security goals for all the critical and/or vulnerable assets ensures that the eventual security requirements are derived based on thoroughly identified types of misuers and of security breaches. Also, well-defined security goals are a prerequisite for identifying threats. (If you have no goals, there are no threats either. For example, "being killed" is only a threat if you have "staying alive" as a goal and "life" as an asset.)
3. ***Identify threats to each asset:*** For each security goal identified in step 2, find all the threats that can prevent the goal from being achieved or maintained. This is where the repository is used for the first time. First, find misuse cases in the repository that involve the right types of misusers as specified by the goal and, then, select those misuse cases that threaten the right type of security breach. Finally, assess whether the misuse case poses a threat that is relevant given the security level specified by the goal. For example, misuse cases that involve the breaking of cryptographic codes may be a relevant threat to the security and integrity of banks military installations with extremely high security levels, but not to the security and integrity of student information in a university information system. In addition to using the repository, it is of course necessary to look for threats that are not directly implied by the determined security goals, because some security goals may indeed have been forgotten.
4. ***Analyze risk for each threat:*** In its most detailed form, the specification of threats must include the risk of the various threats, i.e., the estimated likelihood of occurrence and cost of the damage if the threat occurs. Whereas the description of threats is highly reusable, risks must normally be determined from application to application. For example, although both an Internet entertainment service and a missile control system face the threat of spoofing, the associated risks may be quite different.
5. ***Determine requirements:*** For each identified threat, and taking its risk into account, determine requirements to mitigate the threat. The repository is used again here. For each threat retrieved from the repository, one or more associated bundles of security requirements may be found. For threats not retrieved from the repository, appropriate security requirements must be determined and specified by other means. Even when threats are retrieved from the repository, additional bundles of security requirements that migitate the threat may be found by other means. Different levels of mitigation will be needed for different threats, and requirements workers must select requirements bundles that together produce the necessary levels of mitigation for all threats and that do not overlap.

When the process is completed, there should be satisfactory requirements specified for all threats, and threats should have been investigated for the security goals of all assets.


**Figure 2:** Development for reuse, outlined process

In this paper we do not discuss the first two steps any further (although one might envision some reuse even in those steps, for instance by means of asset checklists), neither do we discuss step 4. It is however necessary to show these steps to illustrate the context in which the reuse of step 3 and 5 takes place. The activity diagram of Figure 3 shows the decomposition of the threat specification (step 3). Three possible ways are suggested to identify threats:

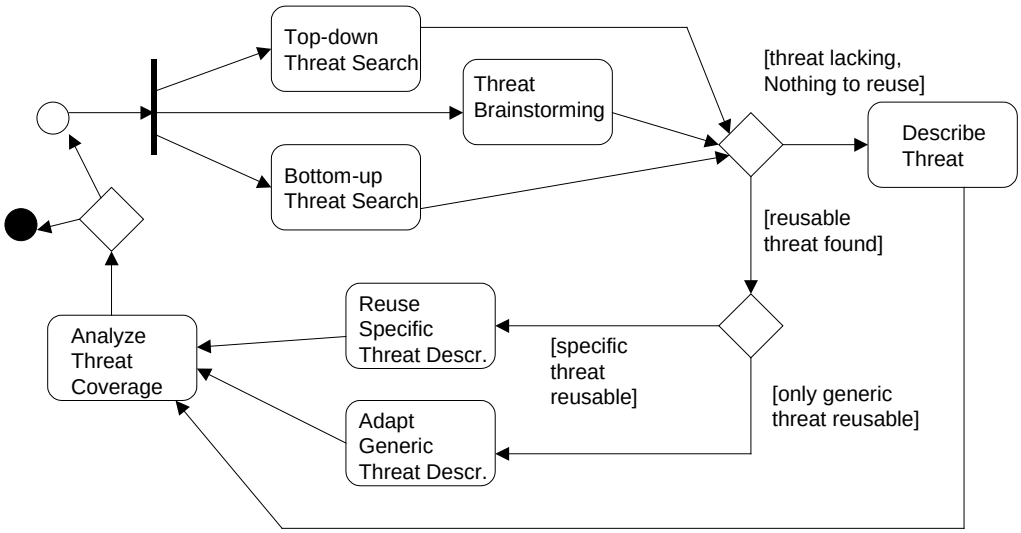


**Figure 3:** Decomposition of "Specify Threats"

- Top down Threat search means that you start from the identified assets and security goals and then try to search the repository for threats relevant to such assets / security goals. This would be best supported if there were attributes pointing to relevant types of assets or security goals in the Threat class, or alternatively there could be separate classes for asset types and goal types which the threats were then associated with. (??? Should we extend the meta-model further?)
- Bottom-up threat search, i.e., starting by looking at what you have in repository (without regard for the determined security goals) and then considering whether different threats described there are relevant to your application. This might seem a less systematic approach than the top-down alternative, and if the repository is big it might cause a lot of wasted time looking at irrelevant threats. However, it might be a valuable corrective to a strict top-down development in that it gives an extra check that no threats have been overlooked. As security goals may have been overlooked in the previous stage (or assets before that), a strict top-down approach gives no guarantee that all threats will be discovered.
- Threat brainstorming. This is the option for threats which cannot be found in the repository (whether mandated from determined security goals or not). But of course, whenever a threat has been suggested by brainstorming, one should check to make sure it is indeed not covered by the repository.

Whatever method a threat has been identified by, one of two situations may occur:

- The repository contains no description that can be reused for this threat. In rare cases this could happen even for threats discovered through the bottom-up approach, i.e., browsing through the repository the developers come upon a threat that is indeed relevant to their application, but the description in the repository is not reusable enough.
- The repository contains a description that can be reused for this threat (can happen even with brainstorming, as once coincidentially brainstorms a threat that was

already covered). In this case there are two new alternatives: Either there is only a generic threat description that can be reused, this must then be adapted to an application specific instantiation. Or there is already a fitting application-specific variety in the repository, then this can possibly be reused as-is, saving even more work for the developers.

As an example, imagine that the repository contained the threat "Spoof User Access" of Table 1, and that this was retrieved and found relevant in the project at hand – to develop a new ATM system. Then, the generic misuse case could be adapted to the application specific misuse case shown in Table 3 – the only phrases that would have to be rewritten would be the underlined ones. However, if there had also been a previous development project for an ATM system by means of the repository, it might well be that there already was such an application-specific misuse case. Then this could be reused directly.

**Table 3:** Application-specific misuse case "Spoof Customer at ATM"

| **Misuse Case Name:** Spoof Customer at ATM | |
|---|---|
| **Summary:** The misuser successfully makes the ATM believe he is a legitimate user. The misuser is thus granted access to the ATM's customer services. | |
| **Preconditions:**<br>1) The misuser has a legitimate user's valid means of identification and authentication ***OR***<br>2) The misuser has invalid means of identification and authentication, but so similar to valid means that the ATM is unable to distinguish ***OR***<br>3) The ATM system is corrupted, accepting means of identification and authentication that would normally have been rejected. | |
| **Misuser interactions** | System interactions |
| Request access | |
| | Request identification and authentication |
| Misidentify and misauthenticate | |
| | Grant access |
| **Postconditions:**<br>1) The misuser can use all the customer services available to the spoofed legitimate user ***AND***<br>2) In the system's log (if any), it will appear that the ATM was accessed by the legitimate user. | |

Moving on to step 5, the decomposed activity diagram for this can be found in Figure 4. When it comes to requirements, the chance for reuse should be considerable if a threat was reused – then one can follow the repository's links to one or more requirement bundles for that threat. If the threat had to be specified from scratch, there are no directly corresponding requirements in the repository, so the chance for reuse is much smaller. Yet, it could pay off at least to browse briefly for requirements related to similar threats, if any.

Either way, it may happen that no requirement bundles are found satisfactory for reuse (because there are none to be found, or because those that were found were not reusable enough, or did not give a sufficient level of mitigation). Or there may be potential for reuse. Here the situation is quite similar to the reuse of threats: It might be that reuse is only possible with adaptation from the generic level, but one might also be lucky enough to be able to reuse something from the specific level, as is. As an example, consider the security use case "ATM Access Control" (or, to be precise, one particular path of that use case). Apart from the name, there was nothing that really needed to be changed in this description – which indeed shows a very high reusability of the generic description. In other cases, more changes might be needed from the generic to the specific level to make descriptions comprehensible to stakeholders.

**Table 4:** One path of the application-specific security use case "ATM Access Control"

| **Security Use Case:** ATM Access Control |
|---|
| **Path name:** Reject invalid authentication |
| **Preconditions:**<br>Misuser has valid means of user identification but invalid means of user authentication. |

| Misuser Interactions | System Requirements | |
|---|---|---|
| | **System Interactions** | **System Actions** |
| | Request user identity and authentication. | |
| Provide valid user identify but invalid user authentication. | | |
| | Reject misuser by cancelling transaction. | Attempt identification, authentication, and authorization. |
| **Postconditions:**<br>1) Misuser has valid means of user identification but invalid means of user authentication ***AND***<br>2) Misuser not authenticated and not granted access ***AND***<br>3) Access control failure registered. | | |

When threats have been analyzed, determining the level of security needed towards various threats, follow the repository links from the threats side to the requirements side, to look at alternative requirements to mitigate the relevant threats – and choose those most appropriate to the needed security levels.

The chosen generic security requirements should then be adapted to application specific ones. In some cases hardly any rewriting is needed (as in the above example with "ATM Access Control"), in other examples it may be necessary to change some terms to application specific ones, and to quantify requirements where the generic ones only indicate the possibility to quantify, e.g., changing <time limit> with an actual time limit or X% with a number.

# 4 Conclusion

The paper has proposed a reuse-based approach to determining security requirements. Development *for* reuse involved identifying security threats and associated security requirements during application development and abstracting them into a repository of generic threats — expressed as misue cases — and requirements — expressed as security use cases. Development *with* reuse involved identifying security assets, setting security goals for each asset, identifying threats to each goal, analysing risks and determining security requirements, based on reuse of generic threats and requirements from the repository. Advantages of the proposed approach include building and managing security knowledge through the shared repository, assuring the quality of security work by reuse, avoiding overspecification and premature design decisions by reuse at the generic level and focussing on security early in the requirements stage of development. The proposed approach may also save time in the early development phases and produce more complete requirements, as the repository may prevent developers from forgetting important threats or requirements. The generic security requirements show the developers what level their description should be at whereas, otherwise, it would be tempting to jump directly from threats to design mechanisms (or even to mechanisms directly, without completely understanding the threats).

Work on reuse-based determination of security requirements is still in its early stages, and industrial case studies are called for. To better support development *for* reuse, further work is needed on how to link misuse cases in the repository to relevant security goals, to better prepare for development *with* reuse. For example, misuse cases in the repository could be described, just like security goals, in terms of the potential misuser, the type(s) of security threat and the typical likelihood of success of the misuse. The repository should be implemented in a tool The repository should be integrated with case tools use for systems development and should support maintaining the repository. For example, the tool should support abstraction of application specific threats and security requirements into generic ones. The tool should also enforce a common taxonomy and terminology, e.g., for misusers and for types of security breaches, in order to increase search efficiency.

To better support development *with* reuse, further work is needed on how to methodically specify security goals, in particular on how to best classify security threats. Heuristics for setting security levels would also be helpful. Of course, the tool should support searching for threats according to misuser and type of security breach, both exactly and approximately.

Comparing the present proposal to goal- and agent-oriented approaches to security requirements work is another path for further work.

## References


1. Jacobson, I., et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*. 1992, Boston: Addison-Wesley.
2. Constantine, L.L. and L.A.D. Lockwood, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. 1999: ACM Press.
3. Cockburn, A., *Writing Effective Use Cases*. 2001, Boston: Addison-Wesley.
4. Rumbaugh, J., *Getting Started: Using use cases to capture requirements.* Journal of Object-Oriented Programming, 1994: p. 8-23.
5. Kulak, D. and E. Guiney, *Use Cases: Requirements in Context*. 2000: ACM Press.
6. Weidenhaupt, K., et al., *Scenario Usage in System Development: A Report on Current Practice.* IEEE Software, 1998. **15**(2): p. 34-45.
7. Sindre, G. and A.L. Opdahl. *Eliciting Security Requirements by Misuse Cases*. in *TOOLS Pacific 2000*. 2000. Sydney: IEEE CS Press.
8. Sindre, G. and A.L. Opdahl. *Templates for Misuse Case Description*. in *7th International Workshop on Requirements Engineering: Foundation of Software Quality (REFSQ'01)*. 2001. Interlaken: Essener Informatik Beiträge.
9. Sindre, G., A.L. Opdahl, and G.F. Breivik. *Generalization/Specialization as a Structuring Mechanism for Misuse Cases*. in *2nd Symposium on Requirements Engineering for Information Security*. 2002. Raleigh, NC: CERIAS / Univ. Purdue.
10. Alexander, I.F. *Initial Industrial Experience of Misuse Cases in Trade-Off Analysis*. in *RE'02*. 2002. Essen: IEEE CS Press.
11. Alexander, I.F., *Misuse Cases, Use Cases with Hostile Intent.* IEEE Software, 2003. **20**(1): p. 58-66.
12. McDermott, J. *Abuse-Case-Based Assurance Arguments*. in *17th Annual Computer Security Applications Conference (ACSAC'01)*. 2001: IEEE CS Press.
13. Alexander, I.F. *Modelling the Interplay of Conflicting Goals with Use and Misuse Cases*. in *8th International Workshop on Requirements Engineering: Foundation for Software Quality*. 2002. Essen, Germany: IEEE CS Press.
14. McDermott, J. and C. Fox. *Using Abuse Case Models for Security Requirements Analysis*. in *15th Annual Computer Security Applications Conference (ACSAC'99)*. 1999: IEEE CS Press.
15. Firesmith, D., *(forthcoming).* Journal of Object Technology, 2003. **2**(3).
16. Crook, R., et al. *Security Requirements Engineering: When Anti-Requirements Hit the Fan*. in *IEEE International Requirements Engineering Conference (RE'02)*. 2002. Essen, Germany.
17. Firesmith, D., *Engineering Security Requirements.* Journal of Object Technology, 2003. **2**(1): p. 53-68.
18. Anton, A.I., et al., *Deriving Goals from a Use Case Based Requirements Specification.* Requirements Engineering Journal, 2001. **6**: p. 63-73.

19. Anton, A.I. and J.B. Earp. *Strategies for Developing Policies and Requirements for Secure Electronic Commerce Systems*. in *1st ACM Workshop on Security and Privacy in E-Commerce*. 2000.
20. Biggerstaff, T. and C. Richter, *Reusability Framework, Assessment and Directions.* IEEE Software, 1987. **4**(2): p. 41-49.
21. Maiden, N. and A. Sutcliffe, *Exploiting Reusable Specification through Analogy.* Communications of the ACM, 1993. **35**(4): p. 55-64.
22. Reubenstein, H. and R. Waters, *The Requirements Apprentice: Automated assistance for requirements acquisition.* IEEE Transactions on Software Engineering, 1991. **17**: p. 226-240.
23. Massonet, P. and A. van Lamsweerde. *Analogical Reuse of Requirements Frameworks*. in *3rd International Conference on Requirements Engineering*. 1997. Washington DC: IEEE CS Press.
24. Lam, W., J.A. McDermid, and A. Vickers, *Ten Steps Towards Systematic Requirements Reuse.* Requirements Engineering Journal, 1997. **2**(2): p. 102-113.
25. Spanoudakis, G. and P. Constantopoulos, *Analogical Reuse of Requirements Specifications: A Computational Model.* Applied Artificial Intelligence, 1996. **10**(4): p. 281-306.
26. Bellinzona, R., M.G. Fugini, and B. Pernici, *Reusing Specifications in OO Applications.* IEEE Software, 1995. **12**(2): p. 65-75.
27. Wirsing, M., R. Hennicker, and R. Stabl. *Menu - an example for the systematic reuse of specifications*. in *2nd European Software Engineering Conference*. 1989. Coventry, England: Springer Verlag.
28. Shehata, M., A. Eberlein, and J. Hoover. *Requirements Reuse and Feature Interaction Management*. in *15th International Conference on Software & Systems Engineering and their Applications (ICSSEA'02)*. 2002. Paris, France.
29. Mannion, M., et al. *Reusing Single Requirements from Application Family Requirements*. in *21st International Conference on Software Engineering (ICSE'99)*. 1999. Los Angeles, CA: IEEE.
30. Baum, L., et al. *Mapping requirements to reusable components using design spaces*. in *4th International Conference on Requirements Engineering (ICRE'00)*. 2000. Schaumburg, IL: IEEE.
31. Lopez, O., M.A. Laguna, and F.J. Garcia. *Reuse-based Analysis and Clustering of Requirements Diagrams*. in *8th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'02)*. 2002. Essen, Germany.
32. Toval, A., A. Olmos, and M. Piattini. *Legal Requirements Reuse: A Critical Success Factor for Requirements Quality and Personal Data Protection*. in *IEEE Joint International Conference on Requirements Engineering (RE'02)*. 2002. Essen, Germany: IEEE.
33. Karlsson, E.-A., ed. *Software Reuse: A Holistic Approach*. Wiley Series in Software Based Systems. 1995, John Wiley & Sons.
34. Sindre, G., R. Conradi, and E.-A. Karlsson, *The REBOOT Approach to Software Reuse.* Journal of Systems and Software, 1995. **30**(3): p. 201-212.
35. Tracz, W., *Software Reuse Myths.* ACM SIGSOFT Software Engineering Notes, 1988. **13**(1): p. 17-21.