Risk-driven architectural decomposition

Thomas Heyman*, Riccardo Scandariato and Wouter Joosen
DistriNet, K.U.Leuven, Belgium
*Email: thomas.heyman@cs.kuleuven.be

Abstract

Architectural modelling and risk management are invaluable to increase the security of a software system. The purpose of this work is to show how both approaches can be integrated in a risk-aware architectural modelling technique. As the modelling is formal, the approach offers stronger guarantees that the resulting architecture achieves its security requirements.

1. Introduction

Making incorrect assumptions during system development is a root cause of insecurity. Therefore, securing a software system implies eliciting the assumptions on which the correct operation of the system is based that may turn out to be invalid, and mitigate them. For this approach to work, the relevant assumptions need to be found and assessed in severity to distinguish acceptable assumptions from fallacious ones.

Model checkers can help in finding assumptions by formally showing what preconditions need to hold for a software system to fulfil its requirements. Additionally, they are able to provide counter-examples when the model contains inconsistencies or fails to uphold the intended requirements. On the other hand, modelling efforts should be focused on security-critical parts of the system. This is where risk comes into play.

Risk management usually elicits unwanted system behavior by means of misuse scenarios or checklists. The risk inherent to these threats is then assessed and the results of the assessment are used to guide the security effort. However, this assumes that attackers behave as modelled in these checklists or misuse scenarios.

This work proposes a stronger integration of risk management and formal modelling to combine the best of both worlds. Formally modelling a system provides strong indications to the software engineer that the system model is correct. Risk assessment focuses modelling effort on critical model parts and assures that no improbable assumptions are made on the context in which the system is being developed and deployed.

This paper is structured as follows. The general approach is described in Section 2 and Section 3. It is illustrated by means of a practical example in Section 4. An overview of related work is given in Section 5. Conclusions are drawn in Section 6.

2. From requirements to assumptions

An overview of architectural modelling is presented in Section 2.1. Different types of assumptions that arise in such a model and guidelines to make them explicit are discussed in Section 2.2.

2.1. Architectural modelling

A software system has to realize certain functionalities, which are usually expressed as functional requirements. Next to functional requirements, non-functional requirements capture qualities (e.g., security, usability and performance) that are important for the system-to-be. These qualities are best embedded at an architectural level [1]. This is also true for security, see for instance [2], [3].

An architecture can be structurally modelled as a set of interconnected components. A component may expose certain operations, grouped in an interface, to be used by other components. Two components can be interconnected by means of a connector, which allows one component to use the interface offered by the other. In this work, Alloy\(^1\) is used as the formal architectural modelling language, as it allows for a natural and concise expression of these architectural concepts.

As a running example, consider an online shop that requires a logger component to store sales transactions.

---

Tampering with the log should be detected, that is, it should be possible to verify that the log is intact. The logger exposes three operations as its interface—one to log messages, one to read them, and one to verify the integrity of the log.

In Alloy, components can be modelled as sets (a sig-declaration in Alloy). An operation is modelled as a predicate on the component offering that operation. An interface is the set of all operations defined on a component—this is not represented explicitly in an Alloy-construct. A connector from component $A$ to $B$ is modelled as an attribute of type $B$ on $A$. Note that Alloy does not have a built-in notion of time. However, time can be modelled as an ordered set, while operations are explicitly quantified with a time instance. Applying the architectural modelling guidelines to the logger example results in the following Alloy model (details will be added later).

```alloy
/* component definitions */
sig Logger {
  contains: set Message -> Time
}
sig Shop {
  /* connector definition */
  log: one Logger
}
/* the log-interface, containing */
/* three operations */
pred Logger.log(m:Msg, t:Time) {
  m in this.contains.t
}
pred Logger.read(m:Message,t:Time) {...}
pred Logger.verify(t:Time) {...}
```

Security requirements, as non-functional requirements, can be translated into constraints placed on component interfaces (so-called component constraints, [4]–[6]). These can be expressed as Alloy assertions, which can be verified by the Alloy analyzer. In order to realize the component constraints, the architecture can be detailed in terms of Alloy statements. These statements, in turn, contain assumptions that are expected to be true in the final system. This situation is depicted in Figure 1.

The logger example has one security requirement: tampering with logged transactions should be prevented. This requirement can be decomposed in “modification of logged entries should be prevented” (the IntegrityProtected constraint) and “it should not be possible to delete entries from the log” (the NoDeletion constraint), both of which are component constraints on the logger component. Consider, for instance, NoDeletion. Reworded in terms of the logger interface, this requirement becomes ‘for all messages that are logged at a certain point in time, the administrator should be able to read them at all later times, or be able to detect that the log is no longer intact’. Formally, in linear temporal logic, $(\forall \ell \in \text{Logger}, m \in \text{Message})(l.log(m) \implies \Box(l.read(\text{AdminEntity},m) \lor \neg l.verify))$. Translated to Alloy, this becomes:

```alloy
/* constraint definition */
assert NoDeletion {
  all l:Logger, t:Time, m:Message {
    l.log[m,t] implies {
      all t1:Time {
        t1.gt[t] implies
        l.read[AdminEntity,m,t1] or
        not l.verify[t1]
      }
    }
  }
}
```

The model does not implement the NoDeletion requirement yet, when verified by the Alloy analyzer. It does, however, after the following explicit assumption is added to the model. Whether this is a reasonable assumption to make is discussed in Section 3. Note that $TO/last$ denotes the last time instance, so $\text{Time} - \text{TO/last}$ are ‘all time instances besides the last one’.

```alloy
/* explicit assumption */
fact AssumeNoDeletion {
  all t:Time - TO/last, log:Logger {
    all m:log.contains.t | m in log.contains.{t.next}
  }
}
```

The initial logger model can now be refined by using a pattern-based approach or similar secure design methodology, or by means of designer expertise. Care must be taken to avoid making assumptions that may not hold. This is the topic of Section 2.2. A second question is when to stop refining. This is where risk assessment comes into play, as illustrated in Section 3.

### 2.2. Locating assumptions

While the failure of functional assumptions is usually experienced as a software bug, the failure of security assumptions results in a vulnerability [3]. By making incorrect assumptions, systems are created that, while secure in theory, fail to guarantee their security requirements in reality. The example model from the previous subsection is largely made up of functional
assumptions, such as ‘two types of components exist: shops and loggers’, ‘every shop has a logger’, etc. However, even this simple example already contains a number of hidden non-functional assumptions that can be easily overlooked during the security effort. These hidden assumptions can be subdivided in three broad classes.

Consider the statement log: one Logger in the previous example. This is clearly a functional constraint, as it imposes that all shop components come equipped with a logger component. However, this statement is stronger than that, as it implies that every shop has exactly one logger. In other words, it contains a hidden availability assumption. This implicit assumption on the cardinality of a relation prevents reasoning about situations where the logger becomes unavailable or when two or more loggers are (accidentally or maliciously) introduced in the system. A weaker definition is log: Logger, as this does not impose a cardinality on the resulting model. Of course, the model may only behave correctly when there is exactly one logger component. By adding the fact all s:Store | one s.log to the model, this non-functional assumption is made explicit.

There is another assumption hidden in log: one Logger. It quietly assumes that the same logger instance will be present throughout the lifetime of the store. In other words, the log connector is assumed to be time-invariant. This prevents reasoning about a logger that is (accidentally or maliciously) swapped with another logger instance. A weaker definition is log: one Logger --> Time, as this allows the log connector to vary through time.

A third class of assumptions is related to timing issues. Consider the statement m in this.contains.t from the definition of the log-operation. It assumes that each message is logged immediately. This implies that the log operation is atomic, and prevents reasoning about interleaving of operations. A weaker definition of this statement is: one t1:Time | t1.gte[t] and m in this.contains.t1

Even this weakened definition assumes that there is a time t1 ≥ t where the message is logged. In other words, liveness issues are not considered. The more general lone t1:Time | ... also takes into account the possibility that there is no time t1 on which the message is logged.

This list is not claimed to be complete. However, it serves to show that simply model checking an architecture is not sufficient to identify all potential security flaws—care must be taken to avoid making too strong statements, resulting in hidden assumptions. By relaxing the model, more potential security issues are uncovered during model checking. In the case where a threat is uncovered, extra facts can be added in Alloy to explicitly constrain the modelled architecture. This helps in distinguishing extraneous non-functional constraints, from those that are essential in upholding the security requirements.

3. Risk and architectural modelling

Risk is the criterion to distinguish a sufficiently secure situation from one that poses an unacceptable high possibility of loss or injury. In this work, risk is used as the criterion to distinguish a “too strong” assumption from an acceptable assumption, as shown further on.

The modelling process can be made risk-aware by assessing risk during model refinement. More precisely, risk inherent to making an explicit assumption is assessed after it is uncovered, in order to distinguish threats which should be mitigated from non-issues. The exact method of assessing risk is out of scope for this approach. Irrespective of the actual method used, it should be able to classify the risk inherent to each assumption as either above or below a predetermined threshold, which represents the risk posture. In this work, risk R is defined as the product of the probability P of an event occurring, times the cost C of it occurring (i.e., the damage). So, \( R = P \times C \).

Coming back to AssumeNoDeletion from Section 2.1, it assumes that all logged entries at times t are also present in the log at time t + 1. This is basically a rewording of the NoDeletion requirement, expressed in terms of the logger internals instead of the logger interface. Note that this assumption is not as academic as it may seem. Suppose the log is written to a ‘Write Once Read Many (WORM)’ medium, then the risk involved in accepting this assumption is minimal, as the only way to delete a log entry is by physically destroying the medium.

However, making an unambiguous risk assessment is not always trivial. The realism of a fine-grained (low-level) assumption is generally easier to assess than that of a high-level one. This is where risk assessment can guide the model refinement process, as it clearly indicates where sufficient detail in the model is lacking. Specifically, if the risk inherent to a certain assumption can not be assessed, then the model is at a too high level of abstraction. The converse is also possible—sometimes the risk inherent to an assumption is too high, but the model is not fine-grained enough to allow the risk to be mitigated, e.g., a control is selected but it is not clear where to apply it. This
is the second criterion to determine whether model refinement is complete from the security perspective.

Suppose that the log is not stored on a WORM medium. It is unlikely then that AssumeNoDeletion would hold. However, the precise risk involved is dependent on the actual implementation of the logger (which is not detailed at this point). Therefore, the model must be refined. Suppose that the software engineer follows a pattern-based approach and is inspired by the SECURE LOGGER security pattern from [7]. Using the pattern description, the developer refines the model by adding a sequence number to each logged message. The logger is extended with a counter, nextUID, that contains the next sequence number to assign to log entries. If, at any time, an entry with a sequence number in the range \{1, \ldots, \text{nextUID}\} is missing, then it was deleted. After some experimenting with this new model, NoDeletion is shown to hold when the following three explicit assumptions are added to the model as facts:

1) The logger should be initialized correctly, i.e., with an empty set of logged messages and with a nextUID of 1.
2) All assigned sequence numbers should be unique—if two messages have the same number, then they are the same message.
3) It should be impossible to tamper with the stored message sequence numbers, i.e., the number associated to each message remains constant through time.
4) It should be impossible to tamper with the nextUID-counter—either it remains the same, or it is increased by one and a new message is logged.

Note that these assumptions should follow from the pattern description. As it is, the first three constraints are also mentioned in the description of the SECURE LOGGER. The last one is not.

It is a lot easier to assess the risk inherent to these assumptions. Assumptions 1 and 2 can be realized through proper implementation of the component, with minimal risk (the cost is the failure of NoDeletion, but the probability is low). The risk inherent to assumption 3 is considered too high—when the log is stored on a hard drive, it is not difficult to alter the sequence number by circumventing the logic of the logger and modifying the log entry via the file system. However, the risk can be mitigated by introducing additional security functions, or monitor it by defining metrics to detect violations of the assumption during operation. If mitigation is infeasible due to a too coarse-grained model, refine the model.

c) If no unambiguous risk value can be obtained, refine the part of the model relevant to this assumption.

The level of detail of the resulting architectural model is adapted to the hostility of the environment in which it will be deployed. The logger model is still too coarse grained for its environment. Other refinements are possible to mitigate additional risk, as shown in the next Section.

4. Illustration

As identified in Section 2.1, the logger also has to realize the IntegrityProtected constraint. An Alloy representation of its current model is shown in the left figure in Figure 2. The type ProcessedMessage is introduced to distinguish between messages to be logged, and actual log entries stored on disk. It contains the original message and a sequence number as introduced in the previous Section.

4.1. First iteration

First, the IntegrityProtected constraint is expressed in terms of the interface of the logger. Based on the description of the SECURE LOGGER pattern, the message id should be part of the data that is integrity
assert IntegrityProtected { all l:Logger, t:Time, m:Message { l.log[m,t] implies (some pm:ProcessedMessage, l:UID { l.read[AdminEntity, pm, t] pm.content.t = m pm.id.t = i all t1:Time | t1.gt[t] implies (pm.content.t1 = m and pm.id.t1 = i) or not l.verify[t1] })}}}

The Alloy analyzer shows that this constraint is not valid. In order to correct the situation, the AssumeIntegrityProtected assumption is added (next to AssumeIDsIntegrityProtected). Note that model checking shows that both assumptions are necessary—NoDeletion is no longer valid when AssumeIDsIntegrityProtected is removed, IntegrityProtected is no longer valid when AssumeIntegrityProtected is removed.

fact AssumeIDsIntegrityProtected { all pm:ProcessedMessage, t:Time - TO/last { pm.id.t = pm.id.(t.next) }}

fact AssumeIntegrityProtected { all pm:ProcessedMessage, t:Time - TO/last { pm.content.t = pm.content.(t.next) }}

The risk inherent to AssumeIntegrityProtected is equal to the probability of a message sequence number being altered times the cost of the impact (the failing of IntegrityProtected). The associated cost is high, but the probability of this event happening is unclear, as the current model is too coarse grained to determine how the sequence numbers will be protected. The same reasoning applies to the second assumption, refinement of the model is necessary. Inspired by the SECURE LOGGER pattern, a digital signature is introduced, as presented in the next Subsection.

4.2. Second iteration

The model of the logger is extended by adding a signature to each ProcessedMessage, encompassing both the message contents and the sequence number, as shown on the right in Figure 2. A ProcessedMessage is wrapped in a SignedMessage, which contains a link to the original signed contents, and one to the current contents. A signature is valid if both point to the same content. A SignedMessage also contains a link to the entity that created the signature. Note that this model of digital signatures is naive, although it is still able to elicit some additional non-functional assumptions.

The first uncovered assumption, LoggerSignaturesCanNotBeForged, is that ProcessedMessages with a valid signature can not be changed without affecting the entity that produced the signature (the signedBy-entity). The second assumption, LoggerSignsOnce, states that the logger only signs each ProcessedMessage once, preventing altered messages to be revalidated by the logger. In essence, the first assumption expresses that the signing algorithm to be used is correct—it is not possible for an attacker to alter the message and pretend that the logger created the signature. The second assumption is a rewording of ‘trust’ in the logger—it is trusted to sign ProcessedMessages at the time they are logged, not after they might be altered.

fact LoggerSignaturesCanNotBeForged { all s:SignedMessage, t:Time { let ts = {t:Time | t = s.signedAt} | t.gt[s.signedAt] implies (s.signedContent.t.content.t = s.signedContent.ts.content.ts and s.signedContent.t.id.t = s.signedContent.ts.id.ts) or s.signedBy.t != s.signedBy.ts }}

fact LoggerSignsOnce { all pm:ProcessedMessage { let msgs = { s:SignedMessage | some t:Time { s.signedContent.t.content.t = pm pm.content.t = pm } } | one msgs } }}

As in the previous iteration, the cost associated to both assumptions is the failing of the IntegrityProtected constraint, and is considered high. The probability that the second assumption fails to hold is low, as this can be enforced at the code-level. The probability of the first assumption failing is unclear at this level, as the model is still too coarse-grained. After another refinement where signatures are produced by means of cryptographic keys, the assumption would be refined to “only the logger knows the signing key”. This can be
assessed and mitigated much better, by implementing a suitable key management scheme. However, this would take us too far.

5. Related work

The Unified Modeling Language (UML) is a commonly used graphical notation for architectural modelling. However, since it is not formally defined, its adoption in formal reasoning techniques is impeded. One approach that tries to overcome this problem is UML/Promela, proposed by McUmber in [8]. It defines a structure-preserving mapping from the UML meta-model to a formal target language meta-model, allowing model checking. The approach is applied to the specification of security patterns, by defining a security pattern template to which a list of formal constraints and consequences can be added [9]. Another approach that attempts to add more formal semantics to UML is [10].

A non-UML approach to architectural modelling and verification is the software specification and analysis method (SAM), first proposed by Wang et al. in [4], and further elaborated in [5], [6]. A SAM model consists of a structural and a behavioural part. The structural specification is provided in CTL/LTL, which allows explicit representation and reasoning about architectural properties, while the behavioural specification is provided by means of a Time Petri Net/Predicate Transition Nets, making it executable. Dong et al. have defined a similar approach to formalize security patterns in XL [11], enabling model checking with the XMC model checker. With it, the authors formalize the behaviour of a number of security patterns and analyze their composition. Security Check (Ray in [2]) is a formal modelling approach for software architectures based on temporal CCS. It allows individual system components to be verified independently, hiding their internals and composing them into larger systems.

6. Discussion and conclusion

This work introduces an iterative architectural model decomposition method, driven by risk assessment on non-functional assumptions made during architectural modelling. It migrates from an explicit attacker model where the capabilities of the attacker are fixed in advance, to an implicit one where the capabilities of the attacker are evaluated against the context. The approach is both relevant w.r.t. the current security requirements, as only assumptions contributing to the realisation of a requirement are considered, and complete w.r.t. a certain abstraction level, as a model checker can verify whether the model realizes the requirements at all times.

Note that the completeness implies that all threats related to the assumption classes described in Section 2.2 are uncovered by the model checking. It does not imply that all possible threats are uncovered, as the list of assumption classes is not claimed to be complete. Even so, the uncovered threats are far from academic, as demonstrated in Section 3.

Future work includes finding a more complete list of hidden assumption classes, as well as validating the approach, e.g., on an industrial case.

References


