Security Patterns 2.0

Towards Security Patterns Based on Security Building Blocks

Alexander van den Berghe
imec-DistriNet
KU Leuven
Leuven, Belgium
alexander.vandenberghe@cs.kuleuven.be

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

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

ABSTRACT

Security patterns are intended to package reusable security solutions and have received considerable research attention in the two decades since their introduction. Practitioners seem less intent to use these security patterns while designing software, though, despite the prevalence of reuse in secure software engineering. We believe this is primarily because current security patterns do not tackle the security problems practitioners actually face.

In this vision paper, we conceive a new way for expressing security patterns, built on a set of reusable and well-known security building blocks. We also advocate the inclusion of these building blocks as first-class citizens in a security-specific modelling language. This not only facilitates unambiguous communication of security solutions such as patterns, but also allows designers to construct a security view of their design which in turn opens new avenues for a broader security by design methodology.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*; *Software security engineering*;

KEYWORDS

Security patterns, Software design, Security view, Modelling language

ACM Reference Format:
Alexander van den Berghe, Koen Yskout, and Wouter Joosen. 2018. Security Patterns 2.0: Towards Security Patterns Based on Security Building Blocks. In *SEAD'18:IEEE/ACM 1st International Workshop on Security Awareness from Design to Deployment , May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194707.3194715

1 INTRODUCTION

Introduced 20 years ago, security patterns [21] are meant to encapsulate well-proven generic solutions to recurring security problems [4, 15], inspired by the now famous design patterns [5]. Furthermore, security patterns are promoted as a tool for non-security experts to incorporate security into their designs [4, 15].

Unfortunately, "until now security patterns have not been used as much as design patterns" [4], as stated in a recent book on security patterns. A survey among Chinese practitioners, for instance, found that just over 20% has used security patterns at least once in their career [3]. Despite practitioners' seemingly limited interest in security patterns, reusing known security solutions is common. For example, authenticating users by username and password, and subsequently letting them perform multiple actions within a session without re-authenticating, is found in most web applications.

Hence, one wonders why the corresponding patterns are not equally popular among software designers. Some authors point to the lack of a good catalogue and methodology as likely reasons for this lack of adoption [4]. Others suggest that the primary culprit is the patterns' sub-optimal documentation quality [2, 6, 10, 22]. We believe these are but symptoms of a more fundamental problem with the state of the art in security patterns, namely that security patterns do not tackle the security issues designers actually face in their daily tasks. More specifically, we have the impression that most security patterns do not deal with *security design flaws* [1, 12] that result from mistakes at the design level. Instead, the existing patterns seem more often concerned with the encapsulation of security features to improve the maintainability of any applied security solutions. While easily maintainable solutions are definitely advantageous, we believe the primary object of security patterns should be the avoidance and resolution of security design flaws.

2 THE PROBLEM WITH SECURITY PATTERNS

To illustrate the problems of current security patterns, we focus on the well-known solution of authentication with sessions mentioned above, which is nowadays implemented in almost every web application using cookie-based sessions. More specifically, we assess how well the Authenticator pattern [4] and Security Session pattern [15] capture these solutions. Note that similar observations can be made for other patterns from these and other sources.

The Authenticator pattern, at its core, says that a subject's identity should always be verified based on some information using some protocol, before assigning a proof of identity to the subject. While this is indeed the desired mechanism, the pattern does not make this much more concrete. The structural view (UML class diagram) relates the most important concepts to each other while abstracting away their security-relevant properties, and the dynamic view (UML sequence diagram) essentially repeats the flow of actions sketched above. Even with this pattern as a resource, a designer thus remains in the dark on how to concretely introduce authentication into a system. How should the proof of identity be associated to the subject? How should authentication information be verified and stored? How and when should the proof of identity be verified, if at all? By following the included references to other patterns (for example, Credential [4] or Single access point [15]), some of these questions can be (partially) answered, at the expense of extra work for the designer by making the correct combinations.

The Authenticator pattern seems to focus on capturing essential concepts and actions related to authentication. While certainly advantageous for educating someone about authentication, its practical use when designing software is limited; we believe a security pattern for authentication should focus on where and how the various parts of an authentication mechanism should be integrated into a design to prevent certain security flaws. The central idea of this paper is the belief that it is very hard to talk about the integration of security mechanisms such as authentication in a software design, given that your primitives (UML classes in this case) are suitable (only) for reasoning about generic objects and their type, state, and operations. Such a language strongly pushes towards a 'domain model' of the security solution and its building blocks, rather than the required combination of (existing) security primitives.

Let's now look at the Security Session pattern [15], which essentially advocates encapsulating a user's access rights into a session object that is created after successful authentication. This object thus contains sensitive information, and should be adequately protected from leaking information and unauthorised changes, yet the pattern barely touches upon this. Furthermore, the pattern introduces unique session identifiers, but hardly mentions that they should remain confidential and that the integrity must be preserved. Similarly, the pattern merely mentions that session identiiers should be unique and not easily guessable without further pointers on how to ensure this. Again, we believe the use of a generic modelling language leads pattern creators to focus on describing the important concepts that are related to the solution, and neglecting the security flaws that may arise when instantiating them.

A final issue —shared by all security patterns— is that the attacker model is implicit. For example, it is up to the designer to work out that authentication offers little protection against insider threats.

Given these issues, it's not surprising that security patterns aren't picked up very much by practitioners. It appears that they are more likely to consult other (more practical) guidance, for example cheat sheets from OWASP [9, 19], which —although sometimes more focused towards implementation aspects— can also be considered security patterns (without being explicitly named so).

In summary, our position is that the current shortcomings of security patterns are due to (wrongly) approaching them from a general software engineering perspective, instead of a security-specific one, primarily caused by the lack of a suitable language for expressing security solutions and patterns. In the next sections, we outline our vision for such a language, and what opportunities it offers for security patterns and security by design in general.

3 A NEW VIEW ON SECURITY PATTERNS

We claim that security patterns can be improved by building them on top of recurring security building blocks, using a language that directly supports these building blocks. This section first lists the building blocks that we believe are most relevant, and then uses them to revisit the Authenticator and Security Session patterns. The third part of this section discusses the incorporation of the building blocks into a dedicated security design language.

3.1 Security Building Blocks

We have observed that security solutions tend to commonly reuse a restricted set of building blocks, which are also related to common security design flaws [1]. In what follows, we will go over the set of common building blocks that we have encountered.

Data is often a central aspect in security, corroborated by the fact that popular security analysis techniques such as STRIDE [16] are largely data driven. This importance materialises in four data-specific security building blocks.

When dealing with security a number of **data types**, e.g. cryptographic keys, are frequently encountered. It is often important to easily identify this potentially sensitive data and associate extra properties with each instance, which can be encapsulated into built-in data types. For example, the length of cryptographic keys strongly influences the strength of any applied encryption. Furthermore, such data types simplify specifying relations between different types, e.g. pairing a public and private key.

Being able to keep track which **data flows** where within a single system as well as where data enters and exits a system is paramount for security. Sensitive data should be protected appropriately to the applicable attacker model. Data provided by external sources should be properly validated. For example, any user text input should be checked for SQL injections before inserting it into a database.

In security, the **creation of data** sometimes comes with extra requirements, e.g. session identifiers should both be unique and difficult to guess. Explicitly modelling where such data is created in a design allows to expressly incorporate these requirements.

Most software has some sort of **data storage** and, for security, it is important to keep track of places where sensitive data is stored, be it in persistent or volatile storage. Stored passwords, for example, have to remain confidential and be guarded against tampering.

Security measures in a system are only effective if they are enforced, typically via one or more **enforcement points** in a system. For example, authenticating users is of little use when this authentication step can be (easily) bypassed. Modelling which measure is enforced at an enforcement point and for which parts of a system this enforcement applies can significantly simplify such analysis.

A system frequently has to make security-relevant decisions based on some provided information. It is important to clearly know where these **decision points** are located in a system and ensure they have access to all required information to make a decision. For example, authenticating a user requires that the password he or she provided can be compared to that stored by the system. Furthermore, it should be assessed whether decisions can be tampered with by, for instance, influencing its inputs or modifying its result. For example, an attacker could tamper with the public keys retrieved from an external key server when authenticating using public-key cryptography. Different decisions can also depend on each other. For example, authorising users can only be meaningfully conducted after they have authenticated. Explicitly modelling each security decision point in a system helps to ensure they are compatible.

Although cryptography is often essential when designing secure software, designers typically do not, and should not, roll out their own cryptographic algorithms. Instead designers incorporate well-known, proven algorithms into their designs using **cryptographic primitives**, such as encrypt data or calculate hash value. Explicitly representing these primitives allows to append them with required properties and extra requirements. For instance, at places where data is encrypted or decrypted, key management usually occurs as well requiring to protect any used keys. Furthermore, if a design clearly shows any applied cryptography centralising it is simplified.

Finally, security is often subject to the actions of other **active entities** such as attackers, users and (non-security) business functionality. Attackers typically have certain capabilities in how they interact with the system and how they acquire (additional) knowledge. Explicitly modelling these aspects as part of a security design is paramount to assess whether the implemented security measures are sufficient. Similarly considering the user is also a principal aspect for security. For example, a user's device is unlikely to be fully secure, thereby offloading security-relevant functionality is risky.

3.2 The Essence of a Security Pattern

Security patterns can now be expressed in function of the above pre-defined building blocks. This forces the pattern developer to describe the essence of the proposed solution, and it simplifies the verification of pattern instances, at least if these building blocks have well-defined, precise meanings. Furthermore, pattern developers and software designers now share a common vocabulary.

Let's look at the previous session-based authentication example in terms of the above building blocks (shown graphically in Figure 1 using an ad-hoc notation). The essence of this security mechanism consists of users that can login and then receive a session identifier as proof of successful authentication, which is included in subsequent requests of the user. This representation makes it explicit that, for this instance, the security solution in this pattern depends on the presence of two **data storage** places (one for storing sessions, and one for (hashed) passwords) and two **enforcement points** (one for opening a session, and one for checking the session token).

Important **decision points** in this solution concern deciding the user's authentication, which involves the **cryptographic primitive** hashing, (followed by generating a session), and whether the session identifier in subsequent requests is (still) valid.

By explicitly modelling the **data flows** of the pattern, as well as the attacker (as an **active entity**), places where sensitive data needs to be protected become explicit. For example, the flow of the password from the user (another **active entity**) to the system, and the returning flow with the session identifier.

The session identifier returned to the user should be unique and hard to guess. These properties can be explicitly attached to the element where session identifiers are **created**.

Note that this example is incomplete (e.g. password changes and logging out are not covered) and only serves to convey the idea of expressing security patterns in function of security building blocks.

3.3 Language Support for Security

The idea of expressing security patterns in terms of the building blocks does not solve the question of how this can be done in practice. One option would be to express the building blocks in a general-purpose language such as UML. This usually requires heavy extensions to UML, as we experienced ourselves [17], and which can also be seen in UML-based security methodologies such as UMLsec [8] and SecureUML [11]. The main challenge when using UML, or another general-purpose language, is precisely and unambiguously expressing the security aspects within a language that focuses primarily on functionality. For example, to model an attacker, UMLsec has to extend UML with stereotypes and tags, and specify the attacker's capabilities in function of these constructs instead of the attacker model being an integrated part of the design. Furthermore, in practice, UML is less widely used than sometimes assumed [14], thereby diminishing the argument that UML is used and known by the intended audience. A final argument against using UML is that its graphical notation is far from optimal [13].

We believe that using a language specifically founded on the above recurring security building blocks is a superior option. Such a language can be tailored to deal with the intricacies of security, and allows to express an actual security view of the software under design. For instance, the attacker should be a first-class citizen, enabling a precise description of its capabilities.

We have previously proposed such a language [20], including a formal semantics, where the building blocks are directly instantiated as one or more language elements. Our language consists of *data* which is operated on by *processes*, each encapsulating a single well-defined function. Processes can be linked to each other to form *networks*, and each of these elements can be further refined using *assumptions*. For example, one provided type of process is the Attacker, accompanied by a (formal) definition of how it obtains (new) knowledge which can be tailored by the designer to match the envisioned attacker model.

Furthermore, processes such as the Authenticator and Enforcer can respectively be used for making authentication decisions and enforcing security measures. Likewise there are processes for cryptographic primitives such as encrypting data or constructing digital signatures. Several built-in data types support common security-relevant data elements such as credentials and cryptographic keys.

4 BEYOND SECURITY PATTERNS

The availability of a security-oriented modelling language provides possibilities beyond crafting better security patterns. It allows designers to create an actual security view [7] of their software, which can be used to thoroughly reason about the software's security properties. Furthermore, such a security view can be leveraged to provide a broader security by design methodology.

Given that the security language comes with precise, unambiguous semantics, as is the case for our previously proposed language, these semantics can be leveraged to (partially) automate formal verification of the design. For instance, our proposal already allows (albeit requiring considerable expertise) to formally prove whether a security property is satisfied. One can also construct such proofs for isolated security patterns. Every designer applying such a pattern automatically receives strong guarantees that the proven property

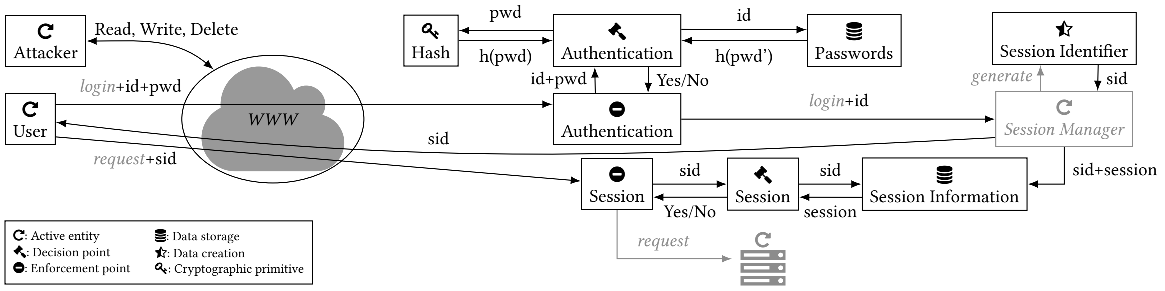

Figure 1: Expressing session-based authentication in function of pre-defined security building blocks forces the pattern developer to describe the essence of the proposed solution.

is satisfied, given that the pattern is correctly instantiated and all necessary conditions are met. For example, it can be proven that the Secure Pipe pattern [18] preserves confidentiality of all data transmitted to it under certain conditions, e.g. correct use of strong encryption. This way, experts can create a catalogue of pre-proven security patterns readily usable by designers.

A security view in which all applied security measures are explicitly represented also facilitates strong traceability throughout the software life cycle. Each element or groups of elements, such as those instantiating a pattern, can be linked to the security requirement that lead to their inclusion. Moreover, these elements can be associated to later artefacts, such as their actual implementation in the developed software, as well as their deployed counterparts. This allows tracing security requirements via their corresponding countermeasures all the way to the deployed software and vice versa. Any later changes to the deployed software can be traced back to the security design thereby allowing to assess how these changes impact already applied countermeasures and thus whether all security requirements are still satisfied.

Furthermore, the semantics underlying the elements in the software's security view also define what behaviour other artefacts such as the implemented software must exhibit. Leveraging the above traceability possibilities the actual implementation of security patterns can be verified against their instances in the design.

5 CONCLUSION

Security patterns have not penetrated the software development practice as far as hoped, despite their popularity among researchers and the prevalent reuse of security solutions. We believe that this is mainly because most patterns do not tackle security design flaws, but rather focus on maintainability aspects of security solutions.

We believe that the definition of security patterns should be based on a selected set of commonly reused security building blocks. Furthermore, we advocate that security patterns should be documented using a modelling language specifically tailored to security and in which these building blocks are first-class citizens.

This security-specific language can also be used by designers to create a security view of their software and serve as a foundation for a broader security by design methodology. Such a methodology opens, with sufficient tool support, the door to verification and traceability of security requirements and solutions throughout a software's lifecycle.

ACKNOWLEDGMENTS

This research is partially funded by the Research Fund KU Leuven.

REFERENCES

[1] I. Arce, N. Daswani, J. Delgrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. Mcgraw, B. Schoenfield, M. Seltzer, D. Spinellis, I. Tarandach, and J. West. 2014. Avoiding the Top 10 Software Security Design Flaws. (2014).
[2] M. Bunke. 2015. Software-security patterns: Degree of Maturity. In *Proceedings of the 20th European Conference on Pattern Languages of Programs - EuroPLoP.*
[3] G. Elahi, E. Yu, T. Li, and L. Liu. 2011. Security Requirements Engineering in the Wild: A Survey of Common Practices. In *IEEE 35th Annual Computer Software and Applications Conference.*
[4] E. B. Fernandez. 2013. *Security Patterns in Practice - Designing Secure Architectures Using Software Patterns.* John Wiley & Sons.
[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissidis. 1994. *Design Patterns - Elements of Reusable Object-Oriented Software.*
[6] T. Heyman, K. Yskout, R. Scandariato, and W. Joosen. 2007. An Analysis of the Security Patterns Landscape. In *Software Engineering for Secure Systems, International Workshop on.*
[7] IEC ISO. 2011. *IEEE: ISO/IEC/IEEE 42010: 2011-Systems and Software Engineering–Architecture Description.* Technical Report.
[8] J. Jürjens. 2004. *Secure Systems Development with UML.* Springer-Verlag.
[9] E. Keary, J. Manico, T. Goosen, P. Krawczyk, S. Neuhaus, and M. Aude Morales. [n. d.]. Authentication Cheat Sheet. https://www.owasp.org/index.php/Authentication_Cheat_Sheet. ([n. d.]).
[10] M-a. Laverdiere, A. Mourad, A Hanna, and M. Debbabi. 2006. Security Design Patterns: Survey and Evaluation. In *Canadian Conference on Electrical and Computer Engineering.*
[11] T. Lodderstedt, D. Basin, and J. Doser. 2002. SecureUML : A UML-Based Modeling Language for Model-Driven Security. In *The Unified Modeling Language, Model Engineering, Concepts, and Tools, 5th International Conference.*
[12] G. McGraw. 2006. *Software Security: Building Security In.* Addison-Wesley.
[13] D. L. Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* 35, 6 (2009).
[14] M. Petre. 2013. UML in Practice. In *Proceedings of the 2013 International Conference on Software Engineering.*
[15] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. 2006. *Security Patterns - Integrating Security and Systems Engineering.*
[16] A. Shostack. 2014. *Threat Modeling - Designing for Security.* John Wiley & Sons.
[17] L. Sion, K. Yskout, A. van den Berghe, R. Scandariato, and W. Joosen. 2015. MASC: Modelling Architectural Security Concerns. In *Proceedings - 7th International Workshop on Modeling in Software Engineering, MiSE.*
[18] C. Steel, R. Nagappan, and R. Lai. 2006. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management.* Pearson Education.
[19] J. Steven and J. Manico. [n. d.]. Password Storage Cheat Sheet. https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet. ([n. d.]).
[20] A. van den Berghe, K. Yskout, R. Scandariato, and W. Joosen. 2017. A Model for Provably Secure Software Design. In *IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering, FormaliSE.*
[21] J. Yoder and J. Barcalow. 1998. Architectural Patterns for Enabling Application Security. In *Pattern Languages of Programs Conference (PLoP).*
[22] N. Yoshioka, H. Washizaki, and K. Maruyama. 2008. A survey on security patterns. *Progress in Informatics* 5 (2008).