Security Pattern Evaluation

Ishbel Duncan
School of Computer Science
University of St Andrews
St Andrews, UK

Jan de Muijnck-Hughes
School of Computer Science
University of St Andrews
St Andrews, UK

***Abstract*—Current Security Pattern evaluation techniques are demonstrated to be incomplete with respect to quantitative measurement and comparison. A proposal for a dynamic testbed system is presented as a potential mechanism for evaluating patterns within a constrained environment**

Keywords: security patterns, evaluation, testing, metrics

I. Introduction

Patterns have been used in the academic community but there has been little take up within industry due to a lack of standardisation or access. Many authors have discussed what patterns are, how they are classified and offered guidelines as to their definition, storage and usage [1-2]. However, there are issues of evaluation of the patterns relying on validation through usage and feedback from users. Halkidis [1] evaluates patterns according to quality criteria and Microsofts STRIDE mnemonic in an attempt to find weakest links and to allow a system to fail securely. Halkidis noted the need to ensure patterns are well analysed and tested. However, this raises the question of how well tested an abstract pattern can be determined and, in parallel, what a well formed pattern actually is. Essentially we are concerned with the assurance that patterns are defined in such a way that erroneous variation is minimised, if not removed, and that any stored, publicly available pattern has been evaluated to a level that enables trust in that pattern. We argue it is not sufficient to accept pattern publication and written, non-metricised, comments stating that testing has occurred. Essentially, an absence of well formed and applied evaluation procedures is not sufficient evidence of a useable pattern. Positive feedback alone can give rise to unwarranted satisfaction and negative feedback is often ignored.

II. Security Life-Cycle

Fernandez [2,3] has noted the need for a Security Development Life-Cycle. By using domain analysis followed by rigorous analysis and design, a pattern can be developed from concept through to implementation and classified at a layer of architectural abstraction. Heyman [4] has indicated the use of metrics to localise weaknesses (e.g. design or requirement led metrics) and by using misuse cases, it can be demonstrated that some types of attacks can be theoretically minimised by use of the stored pattern. However, an issue again arises from pattern theory. Forces (and consequences) are trade-offs necessary to design and build pattern implementations. A user can therefore radically change a pattern or its code depending on what forces are considered more important than others; the pattern may be decomposed in different ways and no want of guidelines and best practice will indicate verification and validation of the result. Essentially changing a pattern, even at a microscopic level, can have multiple ripple effects throughout the forces, the cohesive group of linked patterns and associated tests and consequences.

III. Current Evaluation Methods

Current techniques for evaluating a pattern include shepherding and walkthroughs, the equivalent of a Capability Maturity Model quality approach. Some qualitative and quantitative metrics have been noted by some authors [5-6] but more work is needed on quantitative metrics that take account of deltas, forces and consequences. Work on architecture and model checking may be appropriate at the design stages and many authors have noted the use of UMLSec, OCL etc for examining consequences. Essentially this work can lead to separation of concerns and aspect oriented design rules being applied to pattern design. Ease of use is often measured by either the lines of text accompanying a pattern or by its Flesch-Kincaid value. Both have an impact onto pattern extraction ability or requirements as developers naturally require descriptions to search on unless and until a pattern language is defined which mitigates these issues.

IV. Costs of Complexity

The complexity of a pattern, and its sub-system, is also an important, and often over-looked area of concern. Complexity as a metric is often described as Lines of Code, Depth of Inheritance, FanIn and FanOut (the coupling of modules to other modules). More in-depth complexity measures come from the complexity of the logic in predicates (the number of hidden paths), each of which can give rise to a separate flow, implementation or weakness. Essentially we should measure the amount of noise around a pattern wherein variations of a true, or good pattern, exhibit untoward risks. By enabling a reduction or a constraint in the genericity of a pattern to reduce its noise we can perhaps guide its user towards a cleaner implementation. All of these measures require both static and dynamic measurements of a pattern and its close neighbourhood, a lot of effort should be expended a priori but the benefit will be truly useable patterns where the causes and effects are known to within limitations. To misquote Clint Eastwood as Dirty Harry, ”a good pattern should know its limitations”.

For example, a pattern may connect to several other patterns (its FanOut) and a smaller number may use it (its FanIn). These connections will build up pathways between the code generated by the pattern templates and one could elicit test cases from the logic or the flow of these pathways. FIFO, FanIn

times FanOut gives, allegedly, a measure of the complexity of the code, but in actual fact it is not a clear measure because of the predicates and logical structures (such as loops and decisions) within that code. To state that one has tested code by checking N paths equivalent to FIFO is to not understand that code carries a payload of data and logic. However, however crudely, a large FIFO indicates a larger test space and therefore a larger evaluation space.

Similarly, if the pattern indicates data usage there will be test indicators with respect to data structure size, parameter passing and even types. To a tester, a large data set implies a variety of stress tests forcing the boundaries of the number storage, arithmetic manipulations and the storage structure itself. Hence, to indicate complexity of a pattern, and the code derived thereof, is not a simple count of lines of pattern written or the number of predicates or the number of links. However, there is no doubt a sweet point between little effort to gain a rough guideline of complexity and much effort to get accuracy. What we desire is to be able to indicate that a pattern, or a group of patterns, is more complex than another and therefore indicative of higher testing and evaluation costs.

V. A DYNAMIC TESTBED FOR PATTERN EVALUATION

Modern software development life-cycles indicate rapid development, pair programming and mini cycles encouraging a small team of developers to implement code in short cycle bursts. Security life-cycles are not known for rapid development production cycles (and perhaps never should be) but interest should be taken in the prototyping methods of short burst and evaluation cycles demonstrating that a design or an implementation is both validated and verified within the confines of the tests applied. Further, benefit can be gained from continuous integration testing using previous (regression)and new test cases to an implementation. Testing and evaluating security vulnerabilities has to be a continual and adaptable process of iterative test cycles. The ripple effect of minor changes in design and implementation can be measured and minimised by this technique. A continuous development approach can indicate weaknesses more quickly in the system under test.

Using mini-cycles of pattern design and development coupled with prototyping variants of the pattern implementation will allow a repository of patterns to be built, complete with test cases and a history of both positive and negative feedback, usage and metrics. However, this implies that one person or organisation develops a single pattern in a single development and this is not necessarily the case. It is proposed to link a repository of patterns to a test harness, a virtualised environment allowing an instantiated pattern (and its variants) to be executed with a series of weaknesses known from the CAPEC, CVE or NVD databases, instantiated through metasploit.

Quantitative measurements of the forces, the pattern FiFo coupling metric and the testing results will allow a score to be derived for each pattern. Essentially this will be a dynamic form of stress testing a pattern but will allow a series of quantitative measures of the pattern to be stored alongside the qualitative quality metrics as defined by other authors. Driving the quantitative measurements would be a series of questions derived to assess the pattern and its implementation; a) How complex is the solution? How many components, connections, variables etc are measured to describe the mass of the pattern and program? b) How well-tempered is the pattern? Is it complete, wellformed, has it been through a model checker? c) What is the ease-of-use of the pattern? Is it readable, usable, are example codes given, are limitations described? d) Is there a feedback loop that incorporates changes (deltas), test cases, integration information as well as comments? These questions are meant to find out the goodness of fit of the pattern.

The dynamic pattern testbed requires several years of research and implementation if it is to succeed, but only a quantitative and dynamic approach to pattern development and implementation would allow patterns to leave the confines of the pattern researchers and be acceptable to system developers. However, our current theory has been led by two strands of work, pattern language development [7] and security testing [8]. By developing a logically robust language to describe patterns, generated patterns would have a rigorous description that would elicit code, logic and even data parameters in a consistent way. An integration testing tool, such as are available for code testing, could be built to work either at the design level indicating incoherent flows, or at the code level to detect contentions. Both static abstract patterns and the generated code can be measured via basic code and complexity metrics. The expectation is that trials would indicate warning (amber) metrics wherein the pattern will not generate well formed code (by a variety of programmers) or that it is not integrating robustly with connecting patterns (and code). Essentially this work proposes to merge known code testing and evaluation techniques with the higher level thinking skills involved in pattern writing to indicate pattern reliability and code-worthiness. With measures of the usefulness of patterns and their code genericity soundness, it is more likely that pattern uptake will occur in the security domain.

REFERENCES

[1] T. Halkidis, A. Chatzigeorgiou and G. Stephanides G. *A qualitative analysis of software security patterns*. Information and Computer Security Series, LNCS 3269. 2006;25:37992.

[2] EB. Fernandez and MM. Larrondo-Petrie. *Designing Secure SCADA Systems Using Security Patterns*, International Conference on System Sciences, (HICSS), 43rd Hawaii Int. Conf. 2010;

[3] EB. Fernandez, J. Wu,MM. Larrondo-Petrie and Y. Shao. *On building secure SCADA systems using security patterns* Proc. 5th Annual Workshop Cyber Security and Information Intelligence Research, CSIIRW 09 New York, USA: ACM Press; 2009;1. 14.

[4] T. Heyman, R. Scandariato, C. Huygens and W. Joosen *Using Security Patterns to Combine Security Metrics* Third International Conference on Availability, Reliability and Security (ARES08), 2008;

[5] AK. Alvi and M. Zulkernine *A Natural Classification Scheme for Software Security Patterns* IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC), 2011

[6] SL. Pauwels, C. Hbscher, Ja Bargas-Avila and K. Opwis. *Building an interaction design pattern language: A case study.* Computers in Human Behaviour. Elsevier Ltd; 2010

[7] JF. de Muijnck-Hughes and IMM. Duncan. *Thinking towards a Pattern Language for Predicate Based Encryption Crypto-Systems*, IEEE Sixth International Conference on Software Secruity and Reliability (SERE12), Washington D.C., June 2012.

[8] IMM. Duncan. *Intelligent Biological Security Testing Agents*, IEEE Sixth International Conference on Software Secruity and Reliability (SERE12), Washington D.C., June 2012