

In: R. Sekar & J. McHugh (eds.) Proceedings of the New Security Paradigm Workshop (NSPW'2004), White Point Beach, Nova Scotia, C

fundamental characteristics of agile methods. Section 3 identifies the areas where agile methods seem to conflict with the normal way of dealing with security assurance. Finally, Section 4 offers some avenues to compromise: how to combine the various practices in ways that do not completely remove but rather alleviate the pain points. Section 5 draws conclusions.

2. BACKGROUND AND RELATED WORK

This section provides background material on the subjects of the paper; the methods and techniques of the conventional security assurance and the agile development; and reviews related work.

2.1 Security assurance

Security assurance provides confidence in the security-related properties and functionalities, as well as the operation and administration procedures, of a developed solution. Conventional assurance methods (a detailed description is provided in Appendix 1) can be roughly grouped into the use of best practices in a form of (un)official guidelines; design and architectural principles; use of appropriate tools and technologies; dynamic testing and static analysis; and, most importantly, internal and third-party review, evaluation, and vulnerability testing. While being the keys to achieving confidence that a solution meets its security requirements, a third party's objectivity (and therefore independence) and expertise (and therefore high costs) result in a side-effect, documentation-focused development, which is in conflict with agile development methods. Before going into more detail about the conflicting points, let us provide background on agile development.

2.2 Agile methods

Over the last 8 years or so, new families of software engineering methods have emerged, under different labels: Agile methods, which gelled around the agile manifesto [4]. In these families of methods we find Crystal [5], Adaptive Development [2], Feature-driven Development, Scrum, Lean Software development, and the most famous, because of its provocative name, eXtreme Programming (XP). These processes are quite different from each other, each has its special central features and appeals to different types of projects, but they share a certain number of common characteristics.

They are all fundamentally iterative. They do not replicate the traditional linear sequence of requirements, design, implementation, and test, rather they repeat this sequence again and again, exploiting the fact that software is extremely soft, modifiable, and has no associated manufacturing cost.

The traditional waterfall lifecycle includes some feedback loops, some refinements, as developers cannot get everything right in one pass, but generally rework is considered a "bad thing" that should be minimized at all costs.

The agile methods' iterative nature, which can be traced back to the Spiral Model of Barry Boehm [36], allows them to be more accommodating to changes on several fronts:

- emergence of requirements ("customer on site") and, to match it,
- emergence of the design (refactoring, no big up-front design (BUFD)), which goes hand-in-hand with less focus on "big up-front planning," and

- early and gradual construction of a test suite (test-driven development).

Also, in reaction to previous attempts to make the software development process a rigid, rigorous construction or administrative endeavour, all agile methods exhibit a great aversion for "software bureaucracy," and favour direct communication between participants rather than reliance on written artifacts. This is very visible in practices such as "pair programming", "customer on site," or the daily "scrum" (where all team members rapidly gather for a stand-up 10 minute meeting to assess progress). Direct communication becomes a limiting factor when the size of team increases beyond 12 or 15 people and then some intermediate media must be defined. A somewhat unfortunate result is that agile organizations tend to rely heavily on tacit knowledge, which makes the transfer of software to other organizations more difficult.

Finally, many agile methods put much more emphasis on the person, who is not just a cogwheel in an anonymous software-producing machine.

We will revisit these aspects of agile software development methods: iterative lifecycle, emergence, direct communication and tacit knowledge, in the context of security engineering to examine how they work for or against the current practices in this domain.

2.3 Related work

Other researchers have examined ways to reconcile the approaches. Abrams had looked at fitting security engineering in an evolutionary acquisition process [30].

A working group of the National Cyber Security Summit produced recommendations on processes for developing more secure software, but failed to take into account specificities of iterative or agile processes [31]. More importantly, the report authors ignore the fact that software development companies just cannot afford and have little legal and/or economic incentives to employ the processes and methods the report is advocating. The work reported in this paper looks into the ways of making the engineering of secure software more affordable, which is also recommended by Spafford in [34, 35].

Wäyrynen et al. started to investigate the issue of adapting XP to support security engineering [28], which is the opposite to the question we consider in this paper, that is, how to adapt security assurance to fit agile software development. Nevertheless, their conclusions could be helpful for making security assurance and agile development meet half way:

1. Include a security engineer in the development team for assessing security risks, proposing security-related user stories, and for performing "real-time" security reviews of the system design and code through pair programming.
2. Document the security engineer's pair programming activities to build an assurance argument.
3. Document the security architecture for the sake of assurance argument.
4. Complement pair programming with static verification and automatic policy enforcement.

3. PAIN POINTS

When examining the normal practices of security assurance in the context of the agile methods we just briefly characterized, we run into a number of difficulties, conflicts, or “pain points”: 1) reliance on third-party reviews, 2) reliance on third-party evaluation, and 3) reliance on third-party testing. By “third party,” we mean specialists coming “after the show,” expecting to inspect, analyze, validate, test, and then certify a more or less finished product: complete requirements, complete design, and finished implementation, almost ready –to ship.

Current security assurance practices clash with agile development on four fronts:

1. **Tacit Knowledge/Documentation:** Direct communication and tacit knowledge: the specialists have not been on site, so

they must rely on extensive documentation,

2. **Lifecycle:** An iterative lifecycle, as the third-party would (in theory) have been involved at each iteration
3. **Refactoring:** Refactoring, and other major architectural changes.
4. **Testing:** The testing philosophy.

3.1 Direct communication and tacit knowledge

A fundamental practice in the assurance business is to keep developers and security evaluators “at arm’s length” from each other so that they do not affect each other’s ideas. Since security assurance must be completely neutral and objective, its practitioners and the developers should not become too closely involved except

Table 1 – Classification of impedance mismatches

Security assurance method or technique		Match (2)	Independent (8)	(semi)-automated (4)	Mis-match (12)
Re-quire-ments	Guidelines		X		
	Specification analysis				X
	Review				X
Design	Application of specific architectural approaches		X		
	Use of secure design principles		X		
	Formal validation				X
	Informal validation				X
	Internal review	X			
	External review				X
Implementation	Informal correspondence analysis				X
	Requirements testing			X	
	Informal validation				X
	Formal validation				X
	Security testing			X	
	Vulnerability and penetration testing			X	
	Test depth analysis				X
	Security static analysis			X	
	High-level programming languages and tools		X		
	Adherence to implementation standards		X		
	Use of version control and change tracking		X		
	Change authorization				X
	Integration procedures		X		
	Use of product generation tools		X		
	Internal review	X			
	External review				X
Security evaluation				X	

during their information gathering sessions. This leads to developers often focusing on the functional development with a “tunnel vision” that becomes quite blind to security flaws.

3.2 Iterative lifecycle

Involving a third party is expensive and adds to the development time, for example, from a few days to a few months, depending on the project size and complexity. Iterations, and especially frequent iterations common to agile processes, significantly increase the cost of involving a third party at each iteration. While the security assurance efforts proceed, development should continue, but you are going at risks, and defeating some of the benefits of the inspection.

3.3 Refactoring

Since refactoring leads to the redesign, from the bottom up—often to eliminate code redundancies—modules may be assigned new functionality, which may not work well with security constraints. Unfortunately, refactoring is one of the cornerstones of agile development, and it is increasingly supported by tools and methods [33].

3.4 Testing philosophy

The focus of testing is very different for agile methods—functionality testing prepared early and performed routinely throughout development—and for security testing, which proceeds on totally different premises—focus on the least exercised parts of the system (as opposed to general functional testing); focus on pathological aspects, boundary values, and least used aspects.

Complete security testing also involves a test depth analysis, to understand how thorough the tests are. To do this, developers need to document all the tests, which forces early documentation of the requirement and the design, and this brings us again further away from the user stories/emergence of requirements, and closer to BUFD.

The clashes we have identified between the agile methods philosophy and security assurance may not be specific to security engineering only. The certification processes for safety-critical systems, particularly those required for medical instrumentation (21 CFR part 11, or ISO 14791), or the aerospace industry (ARINC DO178B) lead to similar concerns for the same reasons: an independent inspection process relying on written documentation, and the cost of doing inspection iteratively (each iteration defeating partially what had been assessed previously).

4. RECONCILING THE OPPOSITE: MATCHING ASSURANCE WITH AGILITY

What can be done to try to reconcile, or accommodate the methods of security assurance with the practices of agile development? The assurance methods, applied naïvely, would create deterring delays between critically short iterations as well as prohibitively inflate the development budgets. They would also turn away the developers, most who are averse to trading development for documentation. Ideally, adopted security assurance methods and techniques would allow evolving the confidence in the system in regards to security in same iterative, incremental, and emerging fashion, and through same direct communication and tacit knowl-

edge practices as agile development does. In a real world, where there are neither magic nor silver bullets, a compromise that would decrease time and budget overheads (due to security assurance employed at every iteration) and yet provide “good enough” assurance, is necessary. In this section we present some strategies toward a compromise.

4.1 Classifying security assurance methods

It is important to note that not all assurance methods and techniques are in conflict with agile development. We found it useful for the purpose of this discussion to distinguish the following groups:

Natural match. Some agile practices fit well with security assurance. For example, pair programming advocated by XP naturally facilitates internal design and code review, and motivates developers to follow coding standards [6, 7], including standards for writing “secure code.” Additionally, developers receive immediate feedback from their peers, which could very well be on the principles and guidelines of secure design (listed in Appendix 1). As Wäyrynen et al. [28] suggest, the practice of pair programming could be further enhanced by involving a security engineer who can use this opportunity for reviewing the design and the code.

Independence of the development methodology. Some security assurance methods, techniques, and tools can (and have to) be applied throughout the lifecycle independently of the development methodology.

Consider version control and change tracking. Thanks to the rapid evolution of tools, version control and change tracking could now be found in the toolbox of any active programmer, and even small one-person projects (see www.sourceforge.net for numerous examples) exercise some form of change control.

Can be (semi-)automated. Some methods and techniques can be (semi-)automated so that they can be applied during each iteration without creating significant budgetary or time overheads for an agile project. Examples are static analysis of the source code with regards to security-related defensive coding standards, system testing for known vulnerabilities, and penetration testing.

Mismatch. Approximately half of the conventional assurance methods and techniques directly clash with the principles and practices of agile development. Most of these techniques create mismatch due to their reliance on extensive documentation served as a subject of analysis, verification, and validation activities. The most salient one is security evaluation, such as Common Criteria [8].

We summarize the classification in Table 1.

4.2 Proposed strategies

Since the first two groups of security assurance methods and techniques, “matching” and “independent,” can evidently be integrated with agile development, this section focuses on the other two groups. Let us first consider the group of methods and techniques that can be (semi-)automated.

4.2.1 For semi-automatable methods

For this group, tool support can and should be boosted. As with unit testing, which became pervasive after the corresponding libraries and tools (JUnit, CppUnit), had matured, automation of security static analysis and dynamic testing, vulnerability and

penetration testing, as well as requirements testing could lead to the wide acceptance of these methods by agile developers. Automation efforts should strive to reduce the overhead of these methods to so little that they could be applied as often as unit tests are. However, there will be a cost for fixing actual problems found. Automation only addresses half of the problem for this group of security assurance methods.

No matter how much automation is achieved, with security dynamic testing, for example, the development of application-specific tests requires security expertise and security-oriented testing philosophy, which cannot be expected from an average developer. On the other hand, the same is true for application domains. For instance, in order for a complex banking application to be well designed and implemented, the developers are expected to have extensive knowledge in the domain of finance.

A possible way to close this gap is through codifying in the tools themselves the knowledge necessary for applying assurance methods and techniques from this group. Fault injection [9] and automatic test generation [10, 11] techniques, as an example, could be integrated into the security dynamic testing tools. Tests could also be automatically generated from the code to test for boundary values, and to cover least exercised parts or execution paths in a system.

This reliance on tools however is conditioned by several factors: existence of cost-effective tools; quality of the tools, in particular usability, economic and legal incentives to use them; and the breadth of adoption of these tools by the software industry. Whereas with automatable assurance methods the way out of the tunnel is visible, the fourth group is the most challenging.

4.2.2 For mismatching methods

Security assurance methods in this group rely on either extensive documentation of the system, or the involvement of external security or formal verification experts, or both. As a result the methods are the most difficult to reconcile with agile development due to their daunting budgetary and time overheads, as well as the focus on documentation. For instance, in a recent study reported by Veterling et al. [12] it took 3 months for 18 developers to complete one iteration of developing a relatively small application (20 use cases) conformable to level 2 (i.e., structurally tested) of Common Criteria, which did not even include time for third party evaluation. What can be done to adopt these methods to agile development with its short iterations?

We envision two possibilities for matching assurance methods from the fourth group with agility: The simplest to suggest and the most difficult to implement is the invention of new agile-friendly security assurance methods to replace the ones from this group. Although we cannot offer any insights yet on what exactly such methods could be, they would have to remove the pain points detailed in Section 3 by supporting 1) direct communication and tacit knowledge, 2) short and frequent iterations, 3) emerging design through active refactoring with shared code ownership, and 4) test driven development with tests cases determined by user stories. This direction seems to be a promising area for future research. The other possibility is less spectacular but more practical.

Taking into account the observation that in agile development the biggest questions need to be answered as early as possible, we suggest to apply the assurance methods from this group at least

twice in the development lifecycle: once after first several iterations in a project, and once closer to the end, i.e., several iterations before the system is expected to be shipped. The latter application point is clearly necessary in order to obtain security assurance in the final product. The former enables early confidence in the security properties of the main design and architectural decisions, and reduces the possibility of the “big bang” toward the end of the project. Time and resources permitting, additional applications of the methods from this group in between these two is desirable but can be omitted. The main drawback of this compromise, however, is that this will still lead to too much agile-adverse documentation.

Wäyrynen and her colleagues [28] have also proposed to bring security engineers early in the process, maybe only part-time, to inform and educate the rest of the development team and sensitize them to security issues. And Brian Snow noted that “a single security engineer could support in this manner 5 to 20 development teams, if brought in early and used properly.” A core activity of agile security assurance should be the early identification of the types of design and code changes that are likely to cause security problems, and to use these as guidelines when iterating throughout the lifecycle. Brian also noted that one should prohibit security evaluators from suggesting improvements, so that they do not get too involved and attached to certain elements of the design, losing their objectivity in the way. However this may go against the general spirit of agile methods, which encourage high-bandwidth communication, collective code ownership, etc.

5. CONCLUSIONS

This paper makes an initial step toward integrating security assurance methods and techniques into the agile development practices. It classifies conventional methods and techniques used for security assurance in regards to their acceptability for agile development. It also proposes ways to accommodate the conflicting techniques.

Instead of trying to bend the development process in support for security assurance [12, 13, 28], we look at the problem from the opposite end: can we imagine ways of satisfying the demands of security assurance without making the development documentation focused, and totally integrated in the agile practices and artifacts (user stories, code, testing practices), complemented by security-specific analysis tools that would provide assistance and support for the detection of flaws, and the production of test suites specific to security?

At this point, we can only propose a compromise between the two camps. Is it good enough to alleviate the conflict discussed in Section 3? What needs to be done to get to the point where tools integrated in development environments would incrementally and continuously check, test, and analyze the various artifacts—code, design, requirements—in regards to security, pretty much the way today tools like CruiseControl¹ support continuous configuration management, regression testing, and integration? Is it possible to incorporate seamless generation of the evidence necessary for external review, testing, and evaluation (such as CC [8]) into agile practices? These are our questions for the future work.

¹ <http://cruisecontrol.sourceforge.net/> and <http://www.martinfowler.com/articles/continuousIntegration.html>

As noted by an anonymous reviewer, maybe the impedance mismatch that we face is a blessing in disguise; it could cause us to challenge the “good old heavyweight assurance processes” that have been enshrined in many standards and acquisition policies, and may lead to their replacement by other approaches and processes that are “good enough” and more suitable for rapidly developed and deployed commercial software.

References

- [1] Standish Group, *The Chaos Report*, West Yarmouth, MA: The Standish Group, 1995.
- [2] J. A. Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, New York: Dorset House, 2000.
- [3] K. Beznosov, "Extreme Security Engineering: On Employing XP Practices to Achieve 'Good Enough Security' without Defining It," presented at First ACM Workshop on Business Driven Security Engineering (BizSec), Fairfax, VA, USA, 2003.
- [4] Agile Alliance, "Manifesto for Agile Software Development," 2001.
- [5] A. Cockburn, *Agile Software Development*, Boston: Addison-Wesley, 2002.
- [6] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair-Programming," *IEEE Software*, vol. 17, pp. 19-25, 2000.
- [7] R. W. Jensen, "A Pair Programming Experience," in *CrossTalk* 2003.
- [8] CC, "Common Criteria for Information Technology Security Evaluation," 2.1 ed, 1999.
- [9] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*, New York: John Wiley and Sons, 1997.
- [10] G. Wimmel and J. Jürjens, "Specification-Based Test Generation for Security-Critical Systems Using Mutations," presented at the 4th International Conference on Formal Engineering Methods, 2002.
- [11] G. Fink and M. Bishop, "Property Based Testing: A New Approach to Testing for Assurance," in *ACM SIGSOFT Software Engineering Notes*, vol. 22, 1997.
- [12] M. Vetterling, G. Wimmel, and A. Wisspeintner, "Secure Systems Development Based on the Common Criteria: The PalME Project," presented at the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering, Charleston, South Carolina, USA, 2002.
- [13] R. Breu, K. Burger, M. Hafner, J. Jürjens, G. Popp, G. Wimmel, and V. Lotz, "Key Issues of a Formally Based Process Model for Security Engineering," presented at 16th International Conference on Software & Systems Engineering & their Applications (ICSSEA), 2003.
- [14] C. o. t. E. Communities, "Information Technology Security Evaluation Criteria," 1.2 ed, 1991.
- [15] J. Anderson, "Computer Security Technology Planning Study," Air Force Electronic Systems Division ESD-TR-73-51, Vols. I and II, 1972.
- [16] M. Bishop, *Computer Security: Art and Science*, Boston: Pearson Education, Inc., 2003.
- [17] P. G. Neumann, R. J. Feiertag, K. N. Levitt, and L. Robinson, "Software Development and Proofs of Multi-level Security," presented at International Conference on Software Engineering, 1976.
- [18] S. Software, "RATS: Rough Auditing Tool for Security," 2004.
- [19] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," in *IEEE Software*, vol. 19, 2002, pp. 42-51.
- [20] D. A. Wheeler, "Flawfinder," 2001.
- [21] J. Viega, G. McGraw, T. Mutdosch, and E. W. Felten, "Statically Scanning Java Code: Finding Security Vulnerabilities," in *IEEE Software*, vol. 17, 2000, pp. 68-77.
- [22] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code," presented at Annual Computer Security Applications Conference, New Orleans, Louisiana, USA, 2000.
- [23] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Boston: Addison-Wesley, 2001.
- [24] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*, Boston: Pearson Higher Education, 2004.
- [25] G. McGraw and E. Felten, *Java Security: Hostile Applets, Holes & Antidotes*, New York: John Wiley & Sons, 1996.
- [26] G. McGraw and E. W. Felten, *Securing Java: Getting Down to Business with Mobile Code*, 2nd ed, New York: John Wiley & Sons, 1999.
- [27] K. Beck, "Embracing Change with Extreme Programming," *IEEE Computer*, vol. 32, pp. 70-77, 1999.
- [28] J. Wäyrynen, M. Bodén, and G. Boström, "Security Engineering and eXtreme Programming: an Impossible marriage?," in *Extreme programming and agile methods-XP/Agile Universe 2000*, C. Zannier, H. Erdogmus, and L. Lindstrom, Eds. LNCS3134, Berlin: Springer-Verlag, 2004, pp. 117-128.
- [29] M. Poppendieck and R. Morsicato, "Using XP for Safety-Critical Software," *Cutter IT Journal*, vol. 15, no. 9, 2002, pp. 12-16.
- [30] M. D. Abrams, "Security Engineering in an Evolutionary Acquisition Environment," in *Proceedings of New Security Paradigms Workshop*, Charlottesville, VA, 1998, pp. 11-20.
- [31] S. T. Redwine and N. Davis, ed., *Processes to Produce Secure Software*, Towards more Secure Software, Software Process Subgroup of the Task Force on Security across the Software Development Lifecycle National Cyber Security Summit, 2004.
- [32] P. Amey and R. Chapman, "Static verification and extreme programming," in *Proceedings of 2003 annual international conference on Ada*, San Diego, CA, USA, 2003, ACM Press, pp. 4-9.
- [33] J. Kerievsky, *Refactoring to Patterns*, Boston: Addison-Wesley, 2004.

- [34] E. H. Spafford, "Cyber Terrorism: The New Asymmetric Threat," USA House Armed Services Committee, Subcommittee on Terrorism, Unconventional Threats and Capabilities, Testimony July 24 2003.
- [35] E. H. Spafford, "Exploring Common Criteria: Can it Ensure that the Federal Government Gets Needed Security in Software?" USA House Government Reform Committee Sub-

committee on Technology, Information Policy, Intergovernmental Relations and the Census, Testimony September 17 2003.

- [36] B.W. Boehm, "A Spiral Model of Software Development and Enhancement," ACM SIGSOFT Software Engineering Notes vol. 11, 1986, pp. 22-42.

Appendix 1: Conventional Security Assurance

In simple words, security assurance is confidence that a system (or, generally speaking, a solution) meets its security requirements. Mostly of interest to the solution's users and owners, this confidence is based on specific evidence collected and evaluated through the application of assurance techniques. The techniques consist of a) guidelines for developing security requirements, doing design, implementation, and operation/administration of the solution in question, b) methods for gathering assurance-related evidence, and c) evaluating the evidence. While some techniques, e.g., internal and external reviews, are employed across the whole development and operational processes, others could roughly be grouped according to the lifecycle stages in which they are applied.

Requirements assurance

Requirements assurance methods are concerned with justifying that the security requirements specification is complete, consistent, and technically sound. This type of assurance is commonly achieved through following requirements development guidelines and (informally) analyzing the specification. For example, ITSEC [14], a security evaluation criteria used by some European countries, defines suitability analysis that aids in justifying that the security functional requirements are sufficient to mitigate the threats to the system.

Design assurance

Particular architectural and design principles combined with guidelines on the content of design specification, as well as informal and formal techniques for justifying that the design meets the requirements, are employed for design security assurance. Specifically, security assurance methodologies call for modularity, layering, and security kernel [15], among others, as architectural approaches that help to analyze and evaluate system design in the context of security. Among recommended design principles are the following (adopted from [16]):

- **least privilege** A subject should be given only those privileges that it needs to complete its task.
- **fail-safe defaults** Unless a subject is given explicit access to an object, it should be denied access to that object. This principle is a foundation of closed-world security policies.
- **economy of mechanism** Security mechanisms should be as simple as possible.
- **complete mediation** All accesses to objects should be checked to ensure that they are allowed.
- **open design** Security of a mechanism should not depend on the secrecy of its design or implementation.

- **separation of privilege** System should not grant permission based on a single condition.
- **least common mechanism** The mechanisms used to access resources should not be shared to minimize the possibility for attackers to exchange information via the shared mechanisms.
- **psychological acceptability** Security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present.

To aid with assuring a design, corresponding documentation is recommended to specify (informally, semi-formally, or formally) security functions that enforce security in the system; external interfaces through which protected resources and services are accessed; and internal design that defines an implementation of the external interfaces. The primary purpose of the design security specification is to support the validation of the design against the requirements.

Aside from formal techniques (e.g., HDM [17]) based on proof and model checkers and employed almost exclusively in high-assurance efforts, the following informal methods are used for design validation:

- **requirements tracing**: identifying and documenting specific security requirements that are met by (parts of) the design specification,
- **informal correspondence**: showing and documenting that external functional specifications, internal design specifications, and implementation code are consistent with each other, and
- **informal arguments**: helping go beyond tracing design into requirements and to gain confidence in how well the requirements are met by the design.

Implementation assurance

In addition to internal and external reviews, requirements testing and informal correspondence analysis, as well as formal proof techniques, implementation assurance is achieved by the following means:

- a) **Security testing** similar to the testing of other system properties and functionalities, can be functional or structural (a.k.a., black- and white-box testing (respectively), as well as unit and system (a.k.a., end-to-end) testing. Whereas testing of application logic targets common cases and most used functions of a system, successful security testing requires more attention to the least used aspects of a system, pathological cases, and boundary values of the input data.
- b) **Special types of security testing** are focused on finding known vulnerabilities in a solution. Although catching some vulnerabilities can and commonly is automated [18-22], others, such as those caused by errors in the design or improper use of secu-

curity libraries and services [23, 24], require manual efforts of experts. However, even experts can fail to test a security-critical element or execution path of a solution. Confidence in the completeness of security testing is gained through **test depth analysis**, which provides an argument that testing at all levels is sufficient. Such analysis relies upon (and produces more) test documentation, which is also used for internal and external reviews.

c) **High-level programming languages** **tools** designed with security in mind and when used properly, can help gain confidence in an implementation. Run-time environments, JVM and .NET for instance, have been designed to help developers avoid common problems, for systems implemented in vanilla C and C++, such as buffer overflows, string manipulation issues, and memory handling problems. Although systems running in these virtual machines are not automatically free from vulnerabilities [25, 26], they are commonly perceived to help with increasing implementation assurance.

d) The **enforcement of implementation standards** only helps with gaining confidence in the general quality of an implementation but also with security assurance. As a case in point, defensive coding standards that target potential vulnerabilities are starting to appear in the “trenches” [23].

e) Another common approach to increasing security assurance calls for the use of powerful **configuration management tools**

with the capabilities of version control and change tracking, automated integration procedures, product generation, and authorization of changes to a system. The latter capability encourages the discrimination of developers on the basis of which parts of the implementation can be modified by which developer(s). The discrimination is in the conflict with the philosophy of some widely used agile methodologies, such as eXtreme Programming (XP) [27], which “preaches” collective ownership of the code where every developer should feel free to refactor any part of the system to improve or simplify it.

The major source of confidence in the security of high-assurance systems, as well as the main objective of many official assurance efforts, is security evaluation. The dominating security evaluation framework, Common Criteria (CC) [8] describes security related functionality to be included into a system, and assurance requirements on its development. The requirements are organized into evaluation assurance levels (EAL). The highest level, EAL7, requires formal representation of the high-level design and formal proofs of correspondence with the security requirements. CC certification is legally required for military and some government contracts. The CC, however, does not give any guidance on how to fulfill them during the development process [12].