Cost-Effective Security

Johan Peeters
Independent Consultant

Paul Dyson
e2x

To be successful, application software needs compelling functionality, availability within the right timeframe, and a reasonable price. But equally critical, teams must get nonfunctional characteristics right—performance, scalability, manageability, maintainability, usability, and, of course, security. An earlier contribution to this department[1] stressed the importance of going beyond functional requirements. The authors introduced misuse or abuse cases as counterparts to use cases and explained that although use cases capture functional requirements, abuse cases describe how users can misuse a system with malicious intent, thereby identifying additional security requirements. Another prior installment[2] discussed how to fit misuse and abuse cases into the development process by defining who should write them, when to do so, and how to proceed.

In this article, we discuss what abuse cases bring to software development in terms of planning. We don't assume a fixed budget is assigned to security measures but that budgetary constraints apply to the project as a whole. We believe it's reasonable, and often necessary, to trade functionality against security, so the question isn't how to prioritize security requirements but how to prioritize the development effort across both functional and security requirements.

Context

All kinds of pathological behaviors incur costs, resulting in lower than expected net value. Whether such behavior is triggered by malicious or incompetent use, a denial-of-service (DoS) attack, or failure to scale, nonfunctional requirements aren't partitioned in discrete universes; rather, they're defined on a continuum in which they reinforce or live in tension with each other.[3] In terms of reinforcement, consider availability and scalability: classical security literature considers the former a security concern, not the latter, yet well-established approaches to architecting systems for availability tend to benefit scalability and vice versa. A familiar example of the tension between nonfunctional requirements is security versus usability.

If requirements are specified by use cases, the system is said to *implement* the use cases. However, implementation hardly seems the appropriate term in the abuse case context. We favor *refutation*, a term John McDermott introduced in 2001 (www.acsac.org/2001/abstracts/thu-1530-b-mcdermott.html). He argued that developers have a responsibility to construct an assurance argument that shows the system will prevent security breaches given its threat environment. Of course, such assurance arguments might only be possible after implementing adequate security provisions, thus implementation work could flow from a refutation obligation. Note that

Short cycles for hostile environments

Do you want to ensure cost-effective investment in making your software more secure? If so, we have some specific advice for you. In a turbulent environment, plans must be fluid, which is why agile development teams pay little attention to long-term planning. Instead, most of their attention goes into iteration plans, which typically have a three-week horizon. A short delivery cycle lets the organization respond to changes in the environment on very short notice. Whether the delivery is effectively shipped is a business decision, not a technical one. Whereas the traditional reason for short cycles is related to generating revenue quickly, it's also very effective in a hostile environment because adversaries continuously field new attacks. Short iterations are typically the best first step a development team can take to become more agile—but be aware that such change tends to have a high impact on the organization and hence can meet considerable resistance.

refutation can't extend an absolute guarantee that no exploitable vulnerability remains. Instead, the aim is to reduce risk to an acceptable level—no useful system is ever 100 percent secure, and any attempt to make it so rapidly falls afoul of the law of diminishing returns.

Agile development

The agile community's measure of a project's progress is the realization of business value. Customers determine how to spend the development budget, in collaboration with the development team, based on the effort/value ratio. Critically, teams expect changes to both of these parameters during the project's life cycle.[4]

Agile projects typically rely on user stories as the planning unit; such stories serve as lightweight counterparts to use cases and thus tend to express the system's functionality. In short, the customer gives each user story a business value, and the development team estimates the effort involved to realize it and the risks involved with that realization. The customer and development team use these two measures to prioritize requirements and determine what each iteration will deliver. By accepting that the stories, each story's value, and the effort estimate can all change, each iteration plan potentially represents a new approach to delivering the maximum value for the development budget.

However, many agile development teams hide the costs associated with implementing nonfunctional requirements in the estimates of user stories. This is problematic. Obviously, it works for some types of story—for example, a story that describes user registration could include the implementation of a secure connection—but defending against a DoS attack or ensuring scalability isn't easily hidden in any single story, so it tends to become a fudge factor cost across the entire project. Unfortunately, the customer often colludes in this approach, expecting the team to take care of things such as making the system "secure, fast, and scalable." In our experience, however, this collusion often leads to unmet expectations and nasty scares for both the customer and the development team—exactly the situation that agile processes are designed to avoid.

Planning refined

Agile's focus on early delivery of business value often performs better than traditional processes because it builds in early opportunities for feedback and brings forward the date when the project begins to generate revenue. User stories don't adequately express nonfunctional requirements, so it seems natural to extend agile planning with abuser stories, which, instead of business value, bring an expected cost defined as the product of a loss due to a successful attack and the probability of such an attack. The planning challenge is then to solve an optimization problem that not only takes into account the value realized from user stories but also the expected costs incurred by dysfunctional behavior from abuser stories. In other words, each iteration plan should optimize net value.

Unfortunately, abuser stories make the optimization problem much harder to solve because user and abuser stories aren't independent. Use cases bring their abuse cases in tow, but they have a complex, many-to-many relationship, so adding functionality to the system could enable new attacks. Conversely, several use cases can provide the attack paths for a given abuse case. Net value optimization provides a salient illustration of the *attack surface*, which is a system's exposure to potential attackers:[5] the cost associated with concomitant abuse cases is a good measure of the increased attack surface caused by adding a use case to the system. With this correction, some user stories turn out to have a negative value.

The security community long ago realized the necessity of thinking like an attacker to mount adequate defenses, and writing abuser stories is a manifestation of that principle. Unfortunately, adding abuser stories to system requirements also makes planning significantly harder. Although the user story fragment "…an authenticated user enters the stake and presses the 'game start' button…" and the pair of user and abuser story fragments [user story] "…a user enters the stake and presses the 'game start' 'button…" and [abuser story] "…an attacker impersonates a legitimate user and uses his credit to gamble…" are logically equivalent, they aren't equivalent from a planning viewpoint. Not only is estimating value, cost, and effort significantly more difficult in the latter case, the complexity of the planning optimization problem also increases sharply with the number of abuser stories added. However, this increase in complexity buys us superior planning, tracking, and prioritization.

The first sample story unequivocally states the user must authenticate before gambling, thus the implementation must guarantee it. Even without access to the abuser story, it isn't difficult to see one of the user story's acceptance criteria will be that no unauthenticated user can play. However, a team without explicit abuser stories is more likely to overlook the need for refuting a compromise to the authentication system than a team with an impersonation abuser story. An explicit abuser story also eases tracking over time—beyond its initial refutation. Because systems never become wholly immune to attacks, critical abuser stories should never disappear off the radar.

Evolving systems

A successful attack's likelihood can increase through technological progress, such as someone automating an exploit; it can also increase because the system's assets have become more attractive, making the security breach's impact much higher. Because the situation is fluid, the project plan should track abuser stories throughout the application's life cycle. Even though the development team might have previously mitigated the risks from an abuser story to an acceptable level, the expected cost could rise again.

Tracking abuser stories throughout the application's life cycle might seem at odds with the traditional software architecture maxim of building security into the architecture instead of retrofitting it, but agile approaches model the system under development as a fluid entity in which any aspect can change at any time. In the security realm, this principle channels development resources toward a co-evolution of an application's security posture with the ecosystem's predators. Rather than attempting to build in all the security the application might need throughout its life cycle, development resources are more prudently spent building in just enough to protect it from current threats.

We've seen many successful examples of this approach, including ones in which the architecture evolves in parallel with its functionality. Consider a newly launched business that developed an e-commerce Web site with basic functionality and security, and poor scalability. Once the site was live for several months and exceeded its expected targets for users and revenue, the company developed the system further, with a much greater focus on security and scalability, by using the initial system's success to secure additional development funds.

It might be tempting to dismiss this as a dodgy dotcom business's development approach, but many well-established and well-respected corporations used a similar method of investigating the new possibilities offered via the Internet channel. We've had personal experience with several large-scale, business-critical systems developed and released in an incremental manner, where the nonfunctional characteristics evolved in tandem with functionality.

The specification and prioritization of nonfunctional requirements has so far been a somewhat black art in agile development projects. Agile's fundamental goal is that development projects should deliver systems that are "fit for purpose" rather than ones that deliver grand technical solutions at the expense of functionality or fail to deliver anything at all, but the definition of "fit for purpose" must include a system's nonfunctional requirements. Accordingly, we believe explicit planning and tracking for nightmare scenario refutation goes a long way toward helping the customer and development team place the necessary emphasis on a system's nonfunctional characteristics. □

Some pragmatic advice

As you can probably guess, pessimism is prolific: although it might take some effort to put your developers into an attacker's frame of mind, we find that once they are, they quickly find an abundance of abuser stories. Putting an abuser story cost on the same footing as user story value and keeping track of the dependencies between abuser and user stories soon becomes intractable; hence, our advice is to concentrate the planning effort on the most costly abuser stories.

References

1. P. Hope, G. McGraw, and A.I. Antón, "Misuse and Abuse Cases: Getting Past the Positive," *IEEE Security & Privacy*, vol. 2, no. 3, 2004, pp. 90–92.
2. G. Peterson, and J. Steven, "Defining Misuse within the Development Process," *IEEE Security & Privacy*, vol. 4, no. 6, 2006, pp. 64–67.
3. P. Dyson and A. Longshaw, *Architecting Enterprise Solutions: Patterns for High-Capability Internet-Based Systems*, Wiley, 2004.
4. K. Beck, *Extreme Programming Explained*, Addison-Wesley, 2005.
5. M. Howard and D. LeBlanc, *Writing Secure Code*, Microsoft Press, 2003.

Johan Peeters *is an independent software architect and the founder of secappdev.org, a not-for-profit organization that aims to raise security awareness and grow the skill set of the developer community. Contact him at yo@johanpeeters.com.*

Paul Dyson *is a practicing advocate and early pioneer of agile development methodologies. Paul has provided technical leadership on a wide range of agile projects, especially those that don't fit within the agile "comfort zone" of a small team working closely with a single customer. Paul can be contacted at paul@e2x.co.uk.*