Editor: Jane Cleland-Huang
DePaul University,
jhuang@cs.depaul.edu

How Well Do You Know Your Personae Non Gratae?

Jane Cleland-Huang

IMAGINE THAT YOU'RE building a software system that collects healthcare data and financial information from its users. It might seem obvious that this personal information should be protected from prying eyes through access control mechanisms, audit trails, transaction controls, transmission encryption, and so on—in fact, perhaps so obvious that in many cases people perform only a cursory security analysis and produce rather generic security requirements.

But is this the right way to build secure software? Are security requirements so similar across projects that we simply don't need to invest the time to explore product-level needs or to document requirements at an individual level for each project? Can we instead rely on business-as-usual, organizational security practices implemented at the networking and programming levels?

Finding the Right Fit

In my experience, many requirements specs include a lone security requirement or user story saying something to the effect of, "Only authorized users shall access personal healthcare information." Or, if we're even luckier, "An audit log must be maintained of every access to the patient's healthcare information." Unfortunately, skimping on security analysis can be quite risky.

Perhaps one of the most important questions we should ask is whether specifying security requirements at the individual product level actually leads to more secure systems. We might also ask if the benefits of writing better security requirements outweigh its costs. Drawing an analogy from a related field, it's well known that credit card companies, while excellent at detecting fraud and inactivating stolen cards, almost never attempt to track down the actual offenders. Apparently, the cost of bringing them to justice outweighs the benefits, even though such offenders almost definitely pose ongoing security risks.

A rigorous security analysis for a small application behind a firewall doesn't make much sense.

We clearly need to make informed decisions about how much time and effort to invest in analyzing security needs and specifying product-level security requirements. Performing a rigorous security analysis for a small and unimportant application residing behind an organizational firewall probably doesn't make sense. Similarly, if an organization builds families of Web-based educational tests day in and day out, then it can likely build the next solution by reusing security knowledge. For example, it could simply review existing security requirements, ensure they're suited for the new system, make any necessary customizations, and then reuse them to the fullest extent possible.

If, on the other hand, an organization is branching out into a new domain or building a new kind of system, it simply can't afford to shortchange the security analysis and requirements specification steps. It's important to explore product-level risks and vulnerabilities, design safeguards, and specify them as requirements that are then trackable and testable throughout the entire product life cycle.[1]

Security Requirements

I recently discussed some of the shortcomings of the Healthcare.gov website, one of which was a failure to secure personal data. David Kennedy, CEO of TrustedSec, reported that his passive reconnaissance of the website allowed him to extract 70,000 personal records without even any attempt to hack into the system. This was particularly alarming, given that the data hub connects to—and integrates information from—the Internal Revenue Service (IRS), the Department of Homeland Security (DHS), third-party verification processes, and insurance companies. The system therefore potentially exposes personal information at a scale not previously experienced in the US and requires a robust analysis of security needs. Building any such public-facing system clearly demands a systematic approach for analyzing security needs and documenting mitigating requirements.

There are several techniques that you might consider adopting for such purposes. None of these replace the need for analyzing security at the networking and code levels to prevent buffer overflows, SQL injection attacks, and so on. But they can be used to create a mindset of defensive thinking early in the requirements elicitation process. As a general rule of thumb, thinking defensively means that for every new requirement or feature, we need to think about how it could be either intentionally or unintentionally abused.

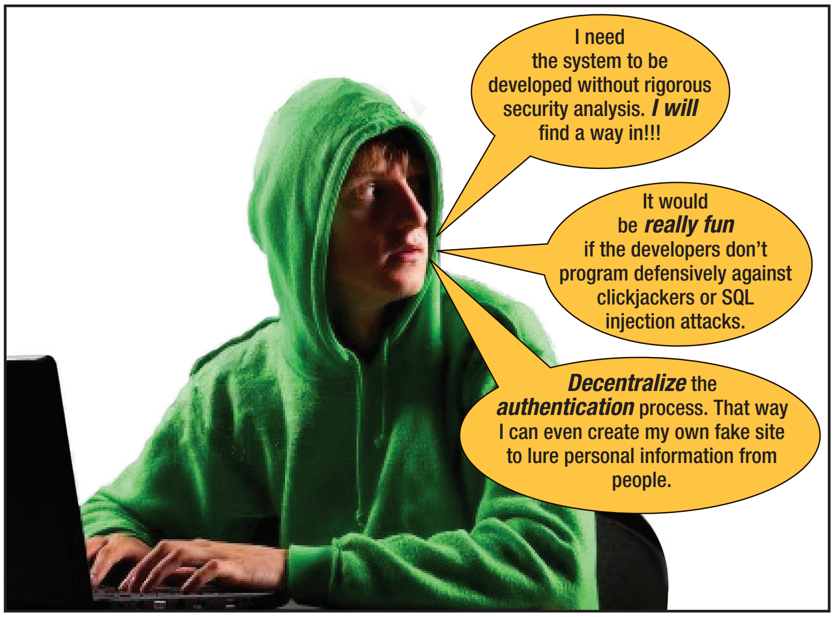

I need the system to be developed without rigorous security analysis. *I will* find a way in!!!
It would be *really fun* if the developers don't program defensively against clickjackers or SQL injection attacks.
Decentralize the *authentication* process. That way I can even create my own fake site to lure personal information from people.

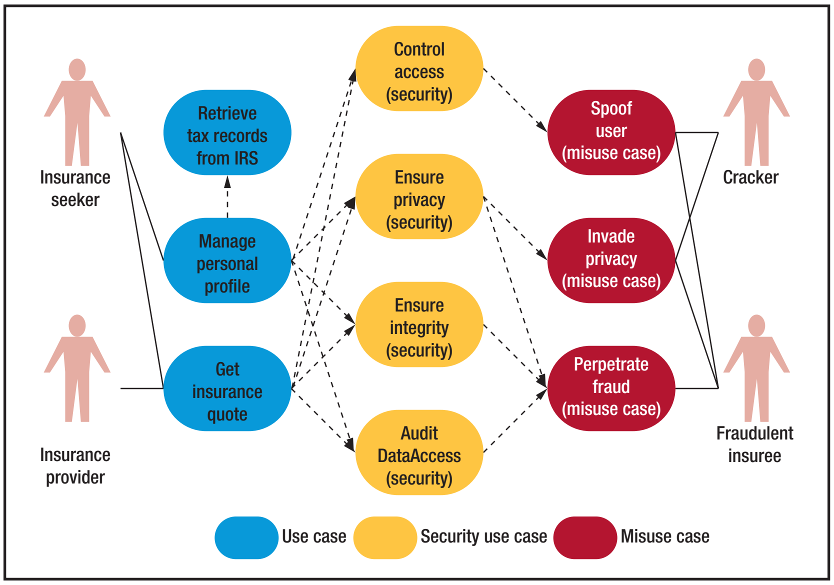
FIGURE 1. An unwanted and malicious intruder is depicted as a persona non grata.

The most obvious technique leverages checklists and regulations. For example, in the US, all systems that involve the use of personal healthcare data are governed by HIPAA's (the Healthcare Information Portability and Accountability Act's) technical safeguards, which stipulate several security-related requirements including access control, automated logout, encryption of stored and transmitted data, and audit logs. Similarly, checklists can be very useful for ensuring that common security issues have been considered. On the other hand, neither of these approaches helps us to think about security risks that might be unique to our application, which means that on their own, they're insufficient.

Three techniques are particularly useful for reasoning about, and defending against, our adversary, the malicious user.

Personae non gratae build on the notion of designing for archetypical users and help us think strategically about the kinds of mischief a malicious user might attempt.[2] Figure 1 shows, Hugh, a young man who intends to attack our system. I've kept Hugh's goals quite generic, but we could create a persona non grata with more specific attack strategies to expose vulnerability points of the product. Creating such personas helps us take a more systematic approach to addressing security concerns throughout the project.

A related technique involves misuse cases, which can be used to determine in advance how the software product should respond to unintended, perhaps malicious, use.[3] In Figure 2's simple misuse case, we see two intended actors: an insurance seeker and an insurance provider, along with their related use cases for managing personal data and seeking/providing insurance quotes.[1] We also see security-related use cases that extend the primary


Insurance seeker
Insurance provider
Retrieve tax records from IRS
Manage personal profile
Get insurance quote
Control access (security)
Ensure privacy (security)
Ensure integrity (security)
Audit DataAccess (security)
Spoof user (misuse case)
Invade privacy (misuse case)
Perpetrate fraud (misuse case)
Cracker
Fraudulent insuree
Use case
Security use case
Misuse case

use cases to achieve access control, privacy, integrity, and auditing goals. On the right side, we see two different types of malicious users—the cracker, who attempts to obtain private information for purposes such as identification theft, and the fraudulent insuree. This second actor isn't necessarily technically savvy but intends to trick the system into

FIGURE 2. Misuse cases showing the interaction of malicious users with a system for attaining healthcare insurance quotes.

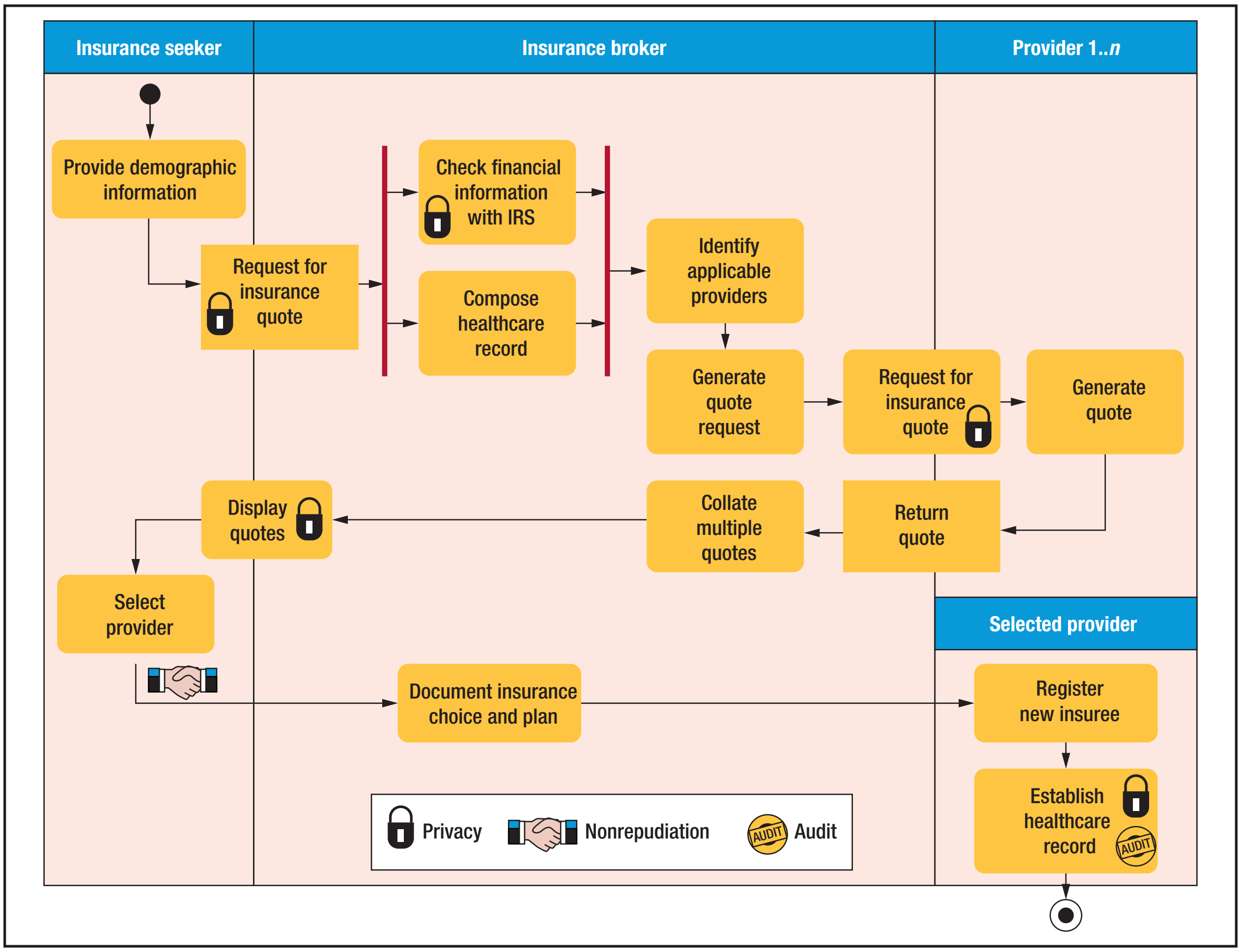

Insurance seeker
Insurance broker
Provider 1..n
Provide demographic information
Request for insurance quote
Check financial information with IRS
Compose healthcare record
Identify applicable providers
Generate quote request
Request for insurance quote
Generate quote
Display quotes
Collate multiple quotes
Return quote
Select provider
Selected provider
Document insurance choice and plan
Register new insuree
Establish healthcare record
AUDIT
Privacy
Nonrepudiation
Audit

FIGURE 3. Activity diagrams model both computational and organizational processes. They can be annotated to depict security concerns.

providing insurance coverage that he's not actually eligible for. While there isn't space to explore these ideas in greater depth, this example illustrates how misuse cases can focus attention on different kinds of abuses, help us explore each of these in depth, determine a suitable set of mitigating requirements, and ultimately design cost-effective countermeasures into the system.

Finally, we could annotate activity diagrams with security concerns.[4] Figure 3 depicts the healthcare insurance seeker's privacy needs, audit capabilities for documenting all official use of the healthcare record, and nonrepudiation requirements related to selecting and signing up for an insurance plan.

These three simple modeling techniques can help you think more defensively during the requirements elicitation and analysis phases of a project. Instead of focusing solely on the system's intended users, you can also take into consideration their counterparts, who either intentionally or accidentally might undermine your system's integrity. Now is the time to think about how well you know your personae non gratae!

References

1. D. Firesmith, "Specifying Reusable Security Requirements," *J. Object Technology*, vol. 3, no. 1, 2004, pp. 61–75.
2. J. Cleland-Huang, "Meet Elaine: A Persona-Driven Approach to Exploring Architecturally Significant Requirements," *IEEE Software*, vol. 30, no. 4, 2013, pp. 18–21.
3. I. Alexander, "Misuse Cases: Use Cases with Hostile Intent," *IEEE Software*, vol. 20, no. 1, 2003, pp. 58–66
4. E. Fernandez-Buglioni, *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*, Wiley, 2013

JANE CLELAND-HUANG is a professor of software engineering at DePaul University. Her research interests include software traceability, architecture, and requirements engineering. Cleland-Huang received a PhD in computer science from the University of Illinois at Chicago. Contact her at jhuang@cs.depaul.edu.

See www.computer.org/software-multimedia for multimedia content related to this article.