Editor: John Steven, jsteven@cigital.com
Gunnar Peterson, gunnar@arctecgroup.net

Using Attack Graphs to Design Systems

Suvajit Gupta and Joel Winstead
Cigital

An attack graph is a visual aid used to document the known security risks of a particular architecture; in short, it captures the paths attackers could use to reach their goals. The graph's purpose is to document the risks known at the time the system is designed, which helps architects and analysts understand the system and find good trade-offs that mitigate these risks. Once the risks are identified and understood in this way, the design can be refined iteratively until the risk becomes acceptable.

On a recent project at Cigital, we found informal attack graphs to be helpful in the iterative design of a system used to protect sensitive data at a customer site. In this article, we use a snippet from this project's design to illustrate the value of using attack graphs in a secure software development life cycle.

Where to start

We began constructing an informal attack graph by analyzing the system's purpose and potential attacker goals. In general, you should also consider up-stream artifacts, such as misuse cases[1,2] when eliciting attacker goals. If such artifacts haven't been created, begin by disrupting the application's positive requirements. Consider that the attacker might simply wish to interfere with the designer's goals. However, not all attacker goals necessarily aim to negate the business's mission outright—for instance, when players cheat in online games, they wish no harm to the game's designer.

What potentially interests an attacker defies easy characterization. A business's sensitive information might not necessarily interest everyone; other system information and assets can be just as valuable. An attacker might not want to disrupt the business's systems, for example, but instead use its hosts or network as a launching point for other attacks. Such an intrusion is actually more valuable to the attacker if the system's primary business mission remains operational and the intrusion remains undetected. In addition to goals, we must also consider different classes of attackers, with varying levels of skill, access, and motivation.

In our attack graphs, we represented attack goals as octagons at the bottom. A successful attack draws a path through the graph to a goal at the bottom. System architectural (or threat) diagrams that show annotations for attack surfaces or threat boundaries help facilitate this. Trust boundaries separate those components in the system that aren't equally trusted—that is, those that would have different consequences if compromised. A database containing sensitive data, for example, is more trusted than other parts of the system that don't have access to this data, so a trust boundary separates them. Be sure not to "cheat" when drawing trust boundaries: if databases house sensitive information, but Java objects mirror that information in the application server using a persistence package (such as Hibernate), then trust has "bled": a compromise of the persistence layer would be just as bad as a compromise of the database itself, so both must be trusted. Drawing accurate trust boundaries means tracing sensitive data or access from privileged functionality to its endpoints.

Penetrating the boundaries between high-trust and low-trust parts of the system often becomes an attacker's intermediate goal because it increases leverage. Attack surfaces show the parts of the system (often correlating with trust boundaries) exposed to an attacker that could serve as an avenue for attack. Designs that exhibit defense-in-depth thoroughly compartmentalize high-trust parts of the system, reducing the size of those modules as much as possible. Such designs use multiple trust boundaries to separate the system's high-value assets from the outermost attack surfaces.

Attack subgoals generally involve crossing a trust boundary, and attackers have many options for achieving their subgoals, each of which can have multiple prerequisites. We captured this in the attack graph by representing subgoals as AND and OR nodes, represented by circles and triangles, respectively. Ultimately, this visual representation helps determine our threat model's thoroughness and provides a much better resource for security test planning.

A system's designers will find it difficult, if not impossible, to construct an exhaustively complete at-

tack graph. When system architects create such a graph, they run significant risk of making the same flawed assumptions they did when designing the system (using external reviewers mitigates this risk). We document attack graphs to log known risks and trade-offs (as well as to drive detailed testing) rather than to prove that the system exhibits perfect security. For this purpose, the graph need not comprehensively list attack paths—just those most likely to result in high-impact, likely exploitation.

Having constructed an attack graph, analysts can use it to refine the design and mitigate various scenarios (again, prioritized by impact and likelihood) in an iterative fashion: each design refinement results in an attack graph refinement, which can result in further analysis and design refinements. Let's look more closely at this process with an example system we recently built for a customer.

Case study

Our customer wanted software to protect sensitive data at a distributed set of sites under a variety of extremely constrained circumstances. Constraints on development time, hardware availability, and business process changes limited what solutions were feasible, so we didn't suffer the delusion that resisting the most concerted attack was possible. Instead, we led the customer through ways to manage risks through acceptable trade-offs in system design and operation.

The software we designed to address this problem encrypts sensitive financial information on customer sites using a set of symmetric keys that it distributes globally throughout the enterprise so that all systems with a need to know can read encrypted messages. The system sits at potentially untrusted customer sites, so it should resist physical theft—preventing an attacker from simply walking off with it. Because of network topology and service, the system must also continue to operate if it suffers a lack of connectivity and failure. These requirements create a difficult (and sometimes conflicting) set of constraints and risks to balance.

Enumerating attacker goals

Considering only encryption and decryption, in-team brainstorming on our design and external review led to the following attack goals:

- access to a sensitive file on the system, for direct financial gain;
- access to the keys, to gain access to sensitive data on other systems; and
- control of the host itself for use as a launching point for other attacks.

The attack described in the second bullet has a bigger impact (than the first) because it exposes all data protected by a key, not just data in a particular file. The third bullet indicates another generalization because the attacker might be unaware of the system's sensitive data. Moreover, this attack poses great danger if other systems trust the compromised host: further (otherwise blatant) attacks could proceed unnoticed.

Identifying attack surfaces

Attack surfaces are boundaries or interfaces directly exposed to an attacker, and as such, they're potential points of entry to the system. They include the public interfaces used in normal system operation but can also include side channels or other surfaces not intended to be accessed directly or external components upon which the system depends. Inside attackers, or attackers who have already penetrated part of the system, might also have access to internal interfaces between components.

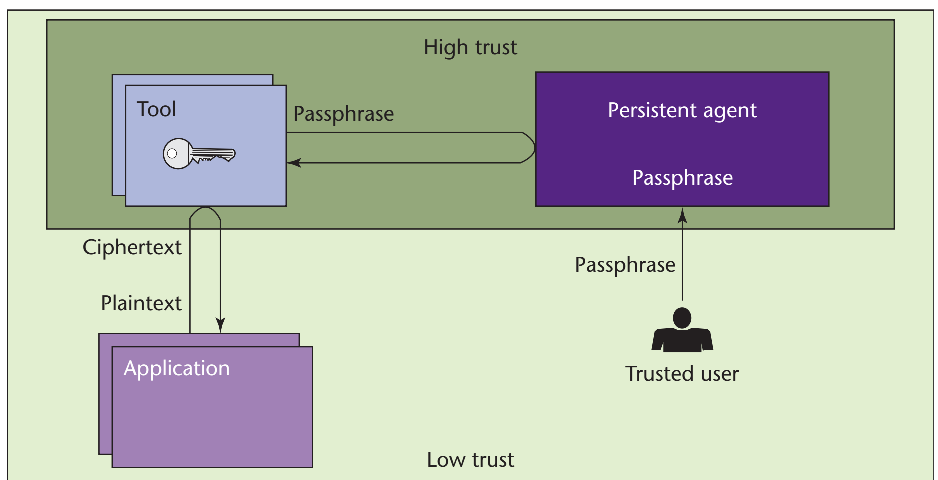
The main attack surfaces we consider in this discussion are the programmatic interfaces to the encryption service and the file system in which sensitive data and encryption keys are stored. An online attacker could attempt to probe a running system through its normal business interface in an attempt to extract keys or data. An attacker could also steal or clone the system's disk and then attempt to extract keys or data from the file system while the system is offline. More specific discussion of attack surfaces would require a technology-specific discussion, which we omit for now.

Initial design and attacks

In the system's initial design, a command-line tool handles encryption and decryption, loads keys, parses data to encrypt, performs the encryption operation, and then exits. The architecture intends the application to call the tool once for each file processed. A passphrase, entered by a trusted operator when the encryption tool starts up, protects the encryption keys.

However, the passphrase requirement presents a usability problem: it must be entered on every tool at start up. Security architects can't reasonably expect a human operator to enter the passphrase at the console every time the tool starts—the application calls this tool on hundreds of files throughout the day, sometimes multiple times. Storing the passphrase on the disk doesn't alleviate an attack, either—that design de-

Umpteen wart hogs grew up, but two mats tickled Paul. One wart hog grew up, however five dwarves auctioned off tickets, even though five extreme 4-line pull quote



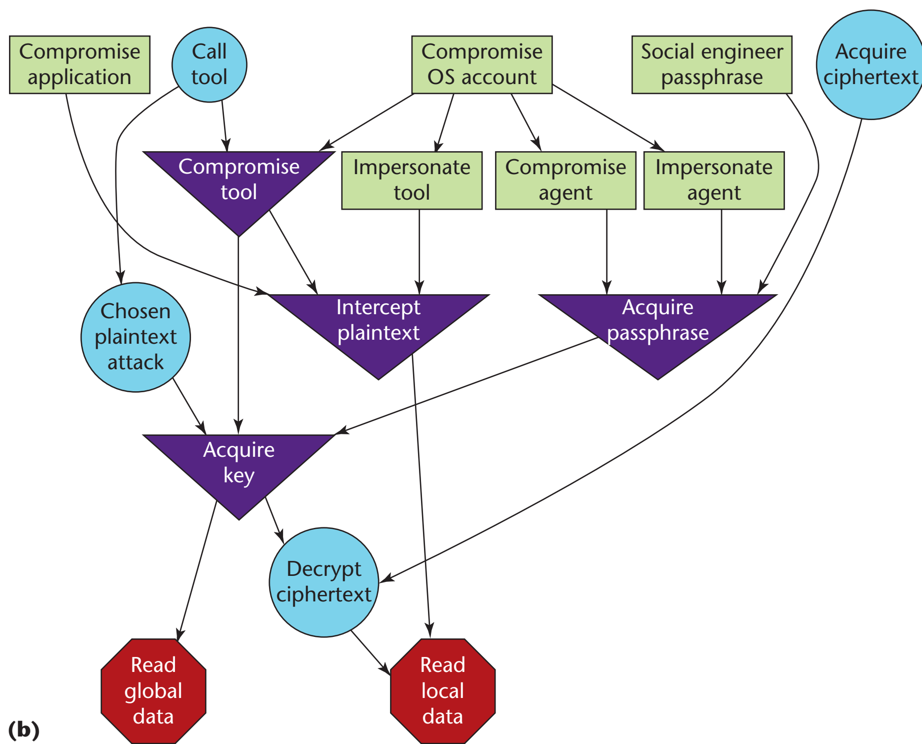


Figure 1. Initial design and trust boundaries. The attack graph below shows potential routes an attacker could take to compromise the system.

cision equates security to storing encryption keys in the clear.

Instead, the initial design solved this problem by storing the passphrase in memory, in an external process dubbed the "passphrase agent." Here, a trusted operator enters the passphrase once, on system boot; when the application invokes the encryption tool and needs the passphrase to unlock its encryption keys, it queries the agent and asks for the passphrase (see Figure 1).

Because it's a high-trust component, the passphrase agent represents a plum target for attack. If revealed to an attacker, the agent could grant the ability to read encryption keys, and thus the ability to decrypt any file offline (satisfying a serious attack goal identified earlier). The encryption tool retrieves the passphrase from the agent via interprocess communication, which also becomes a potential attack surface: impersonating the command-line tool gives an attacker the ability to ask the passphrase agent for the passphrase directly. Although the passphrase agent could attempt to authenticate the command-line tool, or rely on the operating system to prevent the attacker from connecting to the agent process directly, these defenses prove weak.

Revised design and attacks

This realization led to the second design (see Figure 2), which was inspired by SSH and the SSH-agent.[3]

The update moves encryption functionality from the command-line tool to the persistent process. The user interface remains the same: a trusted human operator enters the passphrase into the agent once, at boot time, and the command-line tool encrypts files. However, in this new design, the command-line tool need not be trusted, but the persistent process doesn't leak secrets. Instead, the command-line tool streams the data to be encrypted to the agent, and the agent does all the encryption operations without leaking the key or the passphrase. The keys remain inside a trust boundary with a well-defined interface (which makes assessment easier).

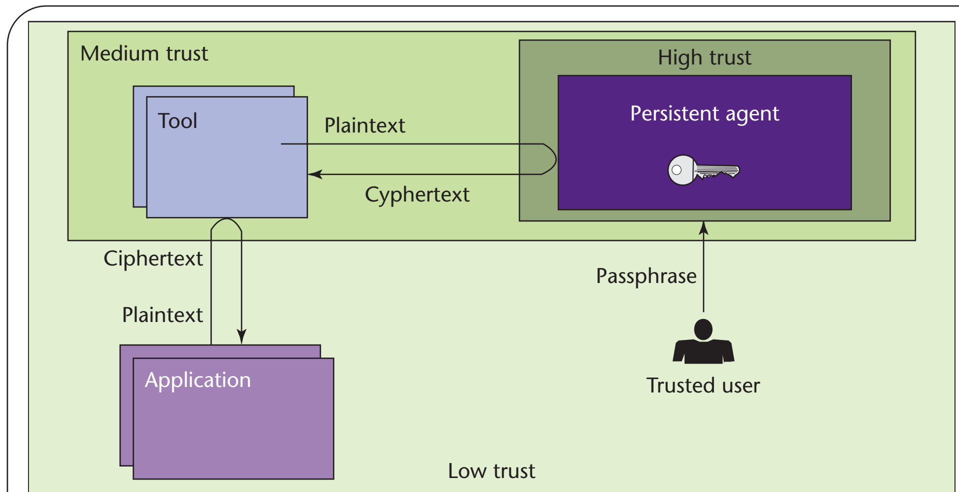
Note that some risks remain: the attacker could still impersonate the command-line tool to contact the agent and decrypt files. However, because we've moved the keys (and the trust boundary), this attack bears less impact than it did in the initial design: although the attacker can decrypt individual files in an online manner, this doesn't give the attacker direct access to the keys or the passphrase that would allow offline attacks.

A motivated attacker might still attempt a cryptographic attack on the keys by using a chosen plaintext

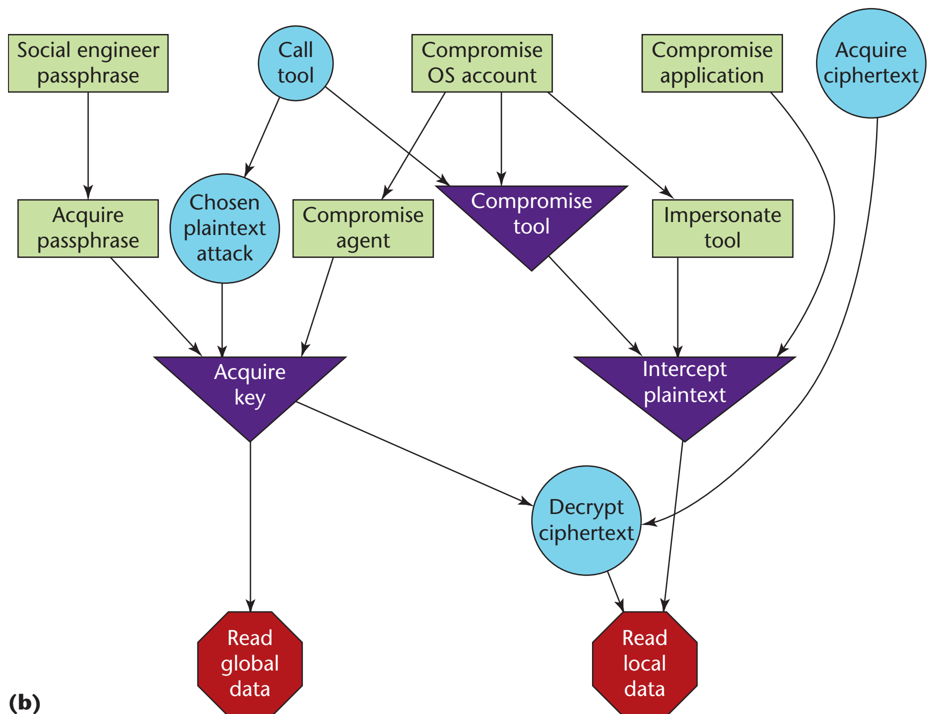
or chosen ciphertext attack, but such an attack requires more skill than the key-revealing attacks in the first design.

The design (and the associated attack graphs) went through several more iterations beyond this article's scope. Through each iteration, the attack graphs provided a high-level representation that helped us reason about the security trade-offs we made that weren't apparent at the code level. Code-analysis tools can help find coding errors, but they do nothing to identify a flaw in the initial design. We aim to manage risk, so we must use an analysis that understands the threats to the system and facilitates trade-offs. A code-scanning tool that assumes the same threats and attack surfaces for all applications would fail to do this, and complete reliance on it could lead to bad trade-offs.

We can't feasibly produce an exhaustive list of all possible attacks, nor does the effort of documenting attack graphs seek to do so. Shared assumptions will always pose problems. Additionally, creative side channels almost always exist, even in an example as simple as the one we just discussed. Regardless, our approach provides value to system implementers because it shows, explicitly, what attacks system designers considered and what trade-offs were made on their behalf. This, in turn, helps maintainers, reviewers, and testers understand the system's security posture, and make good trade-offs that manage the project's overall risk. ▢



(a)



(b)

Figure 2. Revised design. By moving the encryption keys into a separate process, we can tighten the trust boundaries, presenting a higher barrier to the attacker.

References

1. G. Peterson and J. Steven, "Defining Misuse within the Development Process," *IEEE Security & Privacy*, vol. 4, no. 6, 2006, pp. 81–84.
2. J. Peeters and P. Dyson, "Cost-Effective Security," *IEEE Security & Privacy*, vol. 5, no. 3, 2007, pp. 85–87.
3. T. Ylönen, "SSH: Secure Login Connections Over the Internet," *Proc. 6th Usenix Security Symp.*, Usenix Assoc., 1996, pp. 37–42.

Suvajit Gupta *is a managing consultant at Cigital. His research interests include distributed object-oriented architectures, Agile development methodologies, and management of high-performance technical teams. Gupta has an MS in computer engineering from Rensselaer. Contact him at sgupta@cigital.com.*

Joel Winstead *is a senior consultant at Cigital. His research interests include static and dynamic analysis, and architecture for security. Winstead has an MS in computer science from the University of Virginia. Contact him at jwinstea@cigital.com.*