Cloud Shifts the Burden of Security to Development

Arthur Hicken, Parasoft

Abstract. The move to the cloud brings a number of new security challenges, but the application remains your last line of defense. Engineers are extremely well poised to perform tasks critical for securing the application—provided that certain key obstacles are overcome.

Introduction

This paper explores three ways to help development bear the burden of security that the cloud places on them:

- Use penetration testing results to help engineers determine how to effectively "harden" the most vulnerable parts of the application.
- Apply the emerging practice of "service virtualization" to provide engineers the test environment access needed to exercise realistic security scenarios from the development environment.
- Implement policy-driven development to help engineers understand and satisfy management's security expectations.

New Risks, Same Vulnerability

Before the move to the cloud, few organizations lost sleep over application security because they assumed their internally controlled security infrastructure provided ample protection. With the move to cloud, security concerns are thrust into the forefront as organizations consider how much security control they are willing to relinquish to cloud service providers and what level of exposure they are willing to allow [1].

The fact of the matter is that with or without the cloud, failure to secure the application always is—and always has been—a dangerous proposition [2]. Even when the bulk of the network security rested under the organization's direct control, attackers still managed to successfully launch attacks via the application layer. From the 2002 breach at the Australian Taxation office where a hacker accessed tax details on 17,000 businesses [3], to the 2006 incident where Russian hackers stole credit card information from Rhode Island government systems [4], to the recent attack that brought down the NIST vulnerability database [5], it is clear that a deficiency in the application layer can the be one and only entry point an attacker needs.

Public cloud, private cloud, or no cloud at all, the application is your last line of defense and if you do not properly secure the application, you are putting the organization at risk [6]. Nevertheless, the move to the cloud does bring some significant changes to the application security front:

- Applications developed under the assumption of a bullet-proof security infrastructure might need to have their strategies for authorization, encryption, message exchange, and data storage re-envisioned for cloud-based deployment.
- The move to cloud architectures increases the attack surface area, potentially exposing more entry points for hackers. This attack surface area is compounded with more distributed computing technologies, such as mobile, web, and APIs.
- As applications shift from monolithic architectures to composite ones, there is a high degree of interconnectedness with 3rd party services—and a poorly engineered or malfunctioning dependency could raise the security risk of all connected components. For example, a recent attack on Yahoo exploited a vulnerability from a third-party application [7] The composite application is only as secure as its weakest link.
- As organizations push more (and more critical) functionality to the cloud, the potential impact of an attack or breach escalates from embarrassing to potentially devastating—in terms of safety, reputation, and liability.

With the move to the cloud placing more at stake, it is now more critical than ever to make application security a primary concern. The industry has long recognized that development can and should play a significant role in securing the application. This is underscored by the DoD's directive for certifications in the area of software development security (e.g., via CISSP) [8, 9]. Select organizations that have successfully adopted a secure application development initiative have achieved promising results [10]. However, such success stories still remain the exception rather than the rule.

Should Development Be Responsible for Application Security?

Due to software engineers' intimate familiarity with the application's architecture and functionality, they are extremely well-poised to accomplish the various tasks required to safeguard application security. Yet, a number of factors impede engineers' ability to shoulder the burden of security:

- The organization's security objectives are not effectively communicated to the development level.
- For engineers to determine whether a particular module they developed is secure, they need to access and configure dependent resources (e.g., partner services, mainframes, databases) for realistic security scenarios—and such access and configurability is not commonly available within the development environment.
- Management often overlooks security when defining non-functional requirements for engineers and planning development schedules; this oversight, paired with the myopic nature of coding new functionality, commonly reduces security concerns to an afterthought.
- Security tests are frequently started at the testing phase, when it is typically too late to make the necessary critical architectural changes.

In the following sections, we explore how strategies related to penetration testing, service virtualization, and policy-driven development can better prepare engineers to bear the heavy burden of security that accompanies the shift to the cloud.

Moving Beyond Penetration Testing: Divide and Conquer

Penetration testing is routinely used to barrage the application with attack scenarios and determine whether or not the ap-

plication can fend them off. When a simulated attack succeeds, you know for a fact that the application has a vulnerability which makes you susceptible to a particular breed of attacks. It alerts you to real vulnerabilities that can be exploited by known attack patterns—essentially sitting ducks in your applications. When a penetration attack succeeds, there is little need to discuss whether it needs to be repaired. It is not a matter of "if", but rather of "how" and "when."

The common reaction to a reported penetration failure is to have engineers patch the vulnerability as soon as possible, then move on. In some situations, taking the path of least resistance to eliminating a particular known vulnerability is a necessary evil. However, relying solely on a "whack a mole" strategy for application security leaves a considerable amount of valuable information on the table—information that could be critical for averting the next security crisis.

Switching to a non-software example for a moment, consider what happened when the U.S. Army realized how susceptible Humvees were to roadside bombs in the early 2000s. After initial ad-hoc attempts to improve security with one-off fixes (such as adding sandbags to floorboards and bolting miscellaneous metal to the sides of the vehicles), the Army devised add-on armor kits to address structural vulnerabilities and deployed them across the existing fleet [11]. In parallel with this effort, they also took steps to ensure that additional protection was built into new vehicles that were requisitioned from that point forward.

How does such as strategy play out in terms of software? The first step is recognizing that successful attacks—actual or simulated—are a valuable weapon in determining what parts of your application are the most susceptible to attack. For example, if the penetration tests run this week succeed in an area of the application where penetration tests have failed before—and this is also an area that you have already had to patch twice in response to actual attacks—this module is clearly suffering from some underlying security issues that probably will not be solved by yet another patch.

Divide ...

This is where "divide and conquer" comes into play. If you can zero in on your most vulnerable components, it is much simpler for the engineers tasked with securing the application to devise an effective attack plan.

The first task is to determine which parts of the application are most prone to security attacks. Since penetration testing is essentially an "outside in" testing strategy (e.g., launching attacks from the UI or API level and examining the response for indications of success or failure), this can be challenging—particular if your penetration tests are broad rather than targeted. In other words, penetration testing typically tells you that a certain type of attack succeeded, but fails to inform where or how the success was actually achieved.

One way to better isolate the precise point of failure is to have "runtime error detection" monitor the back-end of the application to report exactly where the attack succeeded. Another way is to use an emerging test environment simulation technique known as Service Virtualization to effectively isolate attacks on a component-by-component basis. This strategy will be discussed in more detail later in this paper (along with other uses of service virtualization for application security purposes).

...and Conquer

Once you have zeroed in on the application components where you want to focus your application security resources, it is time to conquer those areas by addressing security from the inside out. Since the development team is most familiar with the application internals, these tasks can and should fall under their purview.

Many organizations are accustomed to relying on penetration testing performed by performance specialists as their primary security strategy. However, as application security comes to the forefront with the move the cloud, it is no longer conscionable to assume that zero successful penetration test attacks indicates a secure application. It is important to recognize that there are two key drawbacks of penetration testing:

- It is reactive: Penetration testing is a reactive approach, checking whether the application resists a set of known attack patterns. As with anti-virus programs, you are constantly chasing after "flavor of the week" attacks rather than taking a proactive approach of identifying and removing root causes.
- It is incomplete: Penetration tests uncover problems only in the paths that the tests exercise. Considering that even a relatively small 10,000-line program has 100 million possible execution paths, the opportunities for overlooked vulnerabilities are staggering.

Fortunately, such drawbacks can be overcome by having development apply two complementary types of static analysis to the modules that your penetration tests identify as being the most vulnerable to security attacks.

Flow-based static analysis is perfectly suited for quickly exposing security vulnerabilities in large code bases without requiring the definition or execution of a single test case. You can hone in on vulnerabilities like the ones that your penetration tests exposed in this component, or you can scour the component for a broader scope of vulnerabilities that might be of interest.

Although flow-based static analysis undeniably checks a broader swath of the application than penetration testing feasibly could, it is important to realize that it is not a panacea. It too has some significant shortcomings. Most notably:

- A complex application has a virtually infinite number of paths [12], but flow-based static analysis can traverse only a finite number of paths using a finite set of data. As a result, flow-based analysis inevitably finds only a finite number of vulnerabilities.
- Flow-based static analysis identifies symptoms (where the vulnerability manifests itself) rather than root causes (the code that creates or allows the vulnerability).

This is where pattern-based static analysis comes in. Pattern-based static analysis exposes root causes rather than symptoms, and can reliably target every single instance of that root cause. For example, if you are relying on flow-based static analysis alone, you will probably find a few instances of SQL Injection vulnerabilities… but you will not find them all. However, if you enforce an input validation rule through pattern-based static analysis that requires wrapping input methods in validation methods—finding and fixing every instance where inputs are

not properly validated—you can guarantee that SQL Injection vulnerabilities will not occur because you no longer have a near-infinite number of paths to search.

Taking Security to the Next Level with Service Virtualization

One emerging trend that empowers engineers to perform additional security verification—from the early phases of the development process and from their traditional development environment—is Service Virtualization. Service virtualization is a method to emulate the behavior of specific components in heterogeneous component-based applications. It is used to provide development and testing teams access to dependent system components that are needed to exercise an application under test (AUT), but are unavailable or difficult-to-access for testing purposes. With the behavior of the dependent components "virtualized," testing can proceed without having to access the actual live components.

Service virtualization opens a number of opportunities for enabling engineers to perform earlier and more effective security testing.

Begin Security Testing Earlier

As applications move to the cloud, tests that validate the security of an end-to-end transaction must pass through more (and more complex) dependencies. Just testing the security of a single transaction often involves interacting with dependencies ranging from mobile applications, databases, mainframes, 3rd party services, internal services, and more. The common approach to testing in such an interconnected environment is to delay testing until all the necessary resources can be accessed at a single point in time. However, such complete access typically comes late…and sometimes never comes at all. With service virtualization emulating the behavior of unavailable or difficult-to-access components, security testing efforts can start significantly earlier—considerably reducing the cost, effort, and resources required to remediate each vulnerability of weakness detected.

Isolate Where Attacks Succeed

Service virtualization enables testing and validation to be applied at multiple depths and levels. By applying validations and assertions to the messages from the application under test, engineers can isolate and zero in on specific components. They could validate a service's response for any indication that its security controls are not working properly (or that additional ones should be implemented). For example, they can determine how the exact element they are working on handles an attempted attack such as an SQL injection. This direct validation provides significantly more visibility into the behavior of a particular component than could be obtained via the output of the larger integrated system.

Gain Predictable, Complete Environment Access from the Development Desktop

After a vulnerability is exposed during penetration testing (or in the field), the software engineers responsible for remediating the problem typically spend considerable effort trying to reproduce the problematic behavior in their own environment so that they can understand it, diagnose the root cause, then verify whether attempted fixes actually resolved the problem.

When tests are run vs. a simulated test environment, engineers are able to rapidly and accurately replicate the precise conditions that triggered the vulnerability. They can then debug the problem and validate their proposed fixes directly from their desktop. If this debugging and validation involves systems beyond development's scope of control (e.g., partner services, databases that are difficult to access for testing, mainframes, etc.), development can run vs. simulated instances of these systems so access limitations do not delay or compromise their ability to complete security remediation tasks.

Moreover, the anytime/anywhere access that service virtualization provides enables the development team to run continuous regression testing. This ensures that the team is alerted if previously remediated security vulnerabilities ever re-surface as the code base evolves.

Run Functional Tests vs. Realistic and Extensive Security Scenarios

With service virtualization, even team members who are not security experts can take their existing functional tests cases and execute them against a broad set of preconfigured security scenarios. Security specialists can pre-configure the environment's dependencies to emulate different security scenarios that would otherwise be difficult to set up and unfeasible to test against. For instance, with SSL, you could configure acceptable and unacceptable certificates. Or, you could emulate various behaviors related to authorization, authentication, and access controls. You could also configure the dependent system to deliver malicious payloads. This provides very granular control over the security behavior of the dependencies in your environment. Now, as a complement to penetration testing, the team's standard functional test scenarios can then be run against these different environment configurations.

Perform Stateful Security Testing

While standard penetration testing simulates attacks through an API in a non-stateful manner, service virtualization lets you emulate attacks at various levels of the system and at different points within a stateful process (e.g., at different points in a logical use case or workflow). This not only broadens your test coverage, but can also expose security vulnerabilities that are manifested only under a certain set of environment conditions—and that would not be apparent with non-stateful penetration testing.

Closing the Gap: Putting Policy in Place

A third component critical to helping development shoulder the burden of security is "policy-driven development." The goal here is to ensure that security requirements (and other non-functional requirements) are exposed and measured as aggressively as functional requirements. Adopting a policy-driven development process where expectations are enforced in

a standardized way across the development group is a low-hanging fruit for taking the necessary level of control over not only what software is being developed, but also over how that software is being developed.

The first step in policy-driven development adoption is to define what policies you want to implement. Policies can be formed around any aspect of the development process. To be effective, they must be definable, enforceable, measureable and auditable. You can define as many policies as necessary to help you achieve your goals, but you should start implementing them on a small scale. Introduce a few policies at a time, and as you become proficient with those policies, you can introduce more in small batches. One approach is to start with policies to ensure that new code is built in a way that prevents security weaknesses and vulnerabilities. Another common approach is to begin by focusing on eliminating the highest severity vulnerabilities in the existing code base.

Next, train software engineers on policies. Beyond documenting the how and the why of your policies, you also want to take steps to ensure that the connection between the two is clear. The absence of training is the Number 1 reason policies fail. If a policy requires code to be structured in a certain way, the engineer may not immediately see the potential for the bug that the structure is intended to prevent. If the engineer does not make the connection during this cycle or even the next few cycles, then the policy looks more like a guideline to him or her, leading typically to an (incorrect) declaration of "false positives." Thus, the code may not be properly structured before the product goes to market, and vulnerabilities may surface in the field. At this point, the implementation of the policy has failed.

Finally, use automation to drive a sustainable process. Automating policy monitoring, as well as the process for routinely notifying engineers of violations, ingrains policies into the day-to-day workflow. Without this level of automation, policies will quickly fade and expected behavior will degrade back into recommended rather than mandated behavior. A centralized infrastructure capable of managing policies will go a long way toward realizing the benefits of policy-driven development. Ideally, a single platform that monitors adherence to multiple types of policies and enables effective implementation will be in place to deliver the traceability required for certification and for audit purposes.

With such a policy-driven process in place, engineers not only know what is expected of them, but receive objective, immediate feedback on whether they are meeting expectations.

ABOUT THE AUTHOR

Arthur Hicken has been involved in automating various practices at Parasoft for almost 20 years. He has worked on projects including database development, the software development lifecycle, web publishing and monitoring, and integration with legacy systems. Arthur has worked with IT departments in companies such as Cisco, Vanguard, and Motorola to help improve their software development practices. He has taught at the College of DuPage in Illinois as well as developing and conducting numerous technical training courses at Parasoft. As an expert in his field, Arthur has been quoted in Business 2.0, Internet Week, and CNET news.com regarding Web site quality issues.

Phone: 626-275-2445
E-mail: ahicken@parasoft.com

REFERENCES

1. States, Lauren. "Tips for Building a Secure Cloud." Lauren States - IBM Cloud Computing. IBM DeveloperWorks, 1 Nov. 2010. Web. 03 May 2013.
2. "Top 10 2010-Main." OWASP. OWASP, n.d. Web. 03 May 2013.
3. Maslowski, Jakub. "10 Reasons Websites Get Hacked." Zone-H.org. Zone-H, 10 Oct. 2007. Web. 03 May 2013.
4. Maslowski, Jakub. "10 Reasons Websites Get Hacked." Zone-H.org. Zone-H, 10 Oct. 2007. Web. 03 May 2013.
5. Gross, Grant. "U.S. NIST's Vulnerability Database Hacked." Computerworld. Computer world, 14 Mar. 2013. Web. 03 May 2013.
6. Schwartz, Matthew. "InformationWeek: The Business Value of Technology." Information Week Security. InformationWeek Security, 7 Apr. 2011. Web. 03 May 2013.
7. Kovacs, Eduard. "Yahoo! Hack Demonstrates the Risks Posed by Third-Party Code in Cloud Computing." Softpedia. Softpedia, 30 Jan. 2013. Web. 03 May 2013.
8. Gilbertson, Daryl. DoD Directive (DoDD) 8570 & GIAC Certification. Publication. SANS, June 2011. Web. 3 May 2013.
9. Grimes, John G. Information Assurance Workforce Improvement Program. Publication. DoD, December 19, 2005, rev January 24, 2012. Web. 9 May 2013.
10. Maiffret, Marc. "Closing the Door on Hackers." New York Times. N.p., 4 Apr. 2013. Web. 3 May 2013.
11. Bowman, Tom. "U.S. Scrambles for Armored Cars as Troops Make Do With Sandbags."Baltimoresun.com. Baltimore Sun, 27 Mar. 2004. Web. 03 May 2013.
12. The underlying "path math" is
    [2n(L1*L2*…*Lx) (V1*V2*…*Vy)]!
    Where:N is the number of decisions (branches).
    Ln is the maximum number of times a loop can loop.
    x is the number of decisions which cause a loop (X≤N).
    Vn is the number of different values each input variable can assume.
    y is the number of input variables.
    The factorial (!) is there because the order of running the test cases counts.