How to Instrument for Advanced Web Application Penetration Testing

NETWORKS®

Table of Contents

1 Foreword

It's critical your web applications are as secure as possible while also staying on schedule and within budget. Often organizations turn to penetration testing and application code scanning to identify security vulnerabilities. While neither approach is perfect, they both do find lots of areas that your developers need to tighten up (and just as importantly, they keep the compliance folks happy). Still, there's no question that there are important vulnerabilities that are being missed. The challenge is how to find the most risky ones without blowing the budget or schedule.

This technical white paper describes a new approach to identifying your most critical web application vulnerabilities faster and at lower cost. The concept is to conduct your penetration testing, vulnerability assessment, or dynamic application testing with deep visibility instrumentation at the database tier. After all, what the attackers want is your data – they either want to steal it or they want to modify it. And much of your most sensitive data of course is in your databases.

Instrumenting with DB Networks' DBN-6300 during a dynamic web application test provides valuable insights into which exploits are actually penetrating your web and application tiers and attacking your critical database assets. Without DB Networks' DBN-6300, a penetration test that reports "no vulnerabilities found" might have been frighteningly close to owning you, but neither you nor the penetration tester would know how close. With in-depth visibility into the database tier you'll be able to analyze these "near misses" during your penetration test post-mortem. And knowing which vulnerabilities allowed unauthorized access to your database tier – your inner sanctum – enables you prioritize what your developers need to fix first.

2 Problem

In the ongoing battle between IT security and those who would do harm, the bad guys have two distinct advantages: time and opportunity. Time, because they can probe a site for many months using automated tools and with inexpensive labor to find a way in. Opportunity, because they can probe and attack a variety of different IT assets and attempt endless approaches until they are ultimately successful in breaching your defenses.

As a defender, on the other hand, you're required to protect all your IT assets all the time, while keeping within an often severely constrained budget. Web applications are a major point of vulnerability in organizations today. Web application vulnerabilities have resulted in the theft of hundreds of millions of credit card numbers, the breach of millions of confidential records, and major financial and reputational damage to a wide variety of high-profile organizations. Application security testing aims to identify weaknesses and vulnerabilities that must be addressed prior to being exploited; and while a great deal of effort has gone into creating various different types of application security tests, the effectiveness and efficiency of application security testing, whether dynamic or static, are still far from optimal. Chief among the issues is a lack of real-time visibility. Tests that appear to pass, finding no vulnerability, may discover important vulnerabilities when greater visibility is available.

3 Background

To explore the strengths and weaknesses of application security testing, the next sections explore three common categories of application testing: dynamic, static, and interactive.

3.1 Dynamic Application Security Testing (DAST)

Dynamic testing, including penetration testing, treats your web application as a "black box" – just as an actual attacker would. Dynamic testers typically begin by looking for common vulnerabilities with automated tools. Skilled DAST practitioners also take an exploratory approach, using clues learned from interacting with your web application to guide them as to where issues are most likely to be found. They may also use information gleaned from a vulnerability assessment, when available.

Dynamic testing clearly has its benefits. It can certainly identify the "low-hanging fruit" – vulnerabilities that are easiest for attackers to exploit, and

therefore most urgent to address. It also identifies a fraction of the more subtle issues and some that may be unique to your environment. The number and severity of vulnerabilities identified during penetration testing typically depend heavily on the skill of the tester and the amount of time allocated for the test.

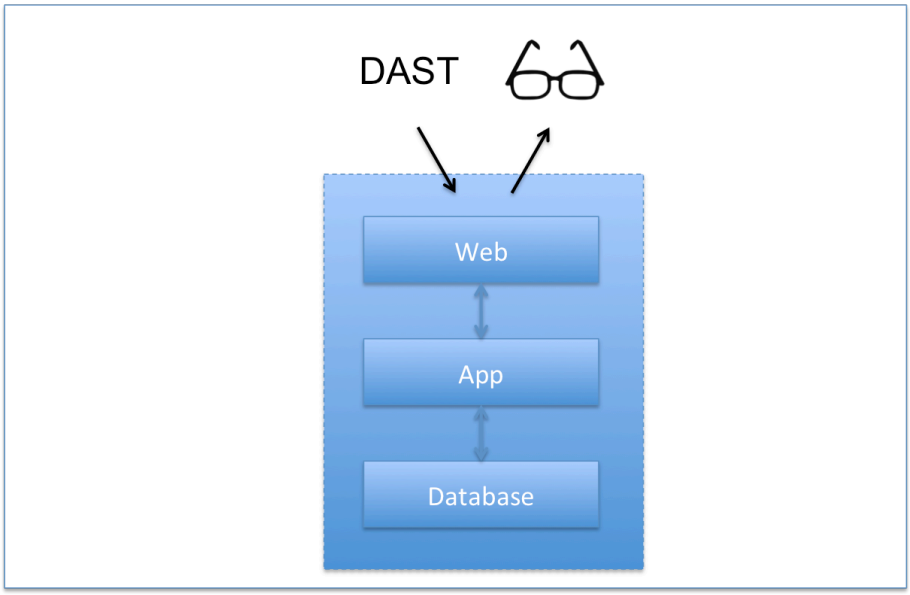

Figure 1. Dynamic testing

Dynamic testing suffers from several well-known limitations:

Limited observability. Dynamic tests and penetration tests probe your system from the outside in. They typically send sequences of requests and then watch for associated responses. As a result they often can't identify when a test actually succeeds in penetrating an application if that penetration does not generate a direct response or an easily visible change. There are two important cases here: a) penetrations that have subtle or hard-to-detect effects, b) near-misses, where an exploit gets through the outer layers of a defense-in-depth architecture but has not quite found the key to the innermost layer. (Some DAST vendors offer instrumentation agents that help provide visibility – these are discussed below in the section on Interactive Application Security Testing.)

Limited time. Closely related to the observability problem is the issue of how much time, and how much money, you can afford to allocate to dynamic testing. The longer a dynamic test runs, the more vulnerabilities it will find; at some point, though, time's up and the test must be concluded. Issues are

inevitably missed that would likely have been found in a longer, more thorough test.

Limited application knowledge. One of the great benefits of dynamic testing – the fact that it can find vulnerabilities in any kind of application using any kind of architecture or technology – can also make it inefficient. Application architecture makes some vulnerabilities less likely and others more likely. Not knowing this, a typical dynamic test wastes time on the unlikely vulnerabilities and spends less time than warranted on the more likely ones. Combined with the tight time window to complete a test, this inefficiency limits the results that can be attained.

3.2 Static Application Security Testing (SAST)

Static testing, including code scanning, analyzes your application source code to identify deviations from recommended secure coding practices. Static testing is also often called "white box" testing. Unlike dynamic testing, a code scan "knows" everything, or nearly everything, about the application. Static testing can also identify a specific line of source code responsible for a security weakness, reducing the time developers need to spend debugging.

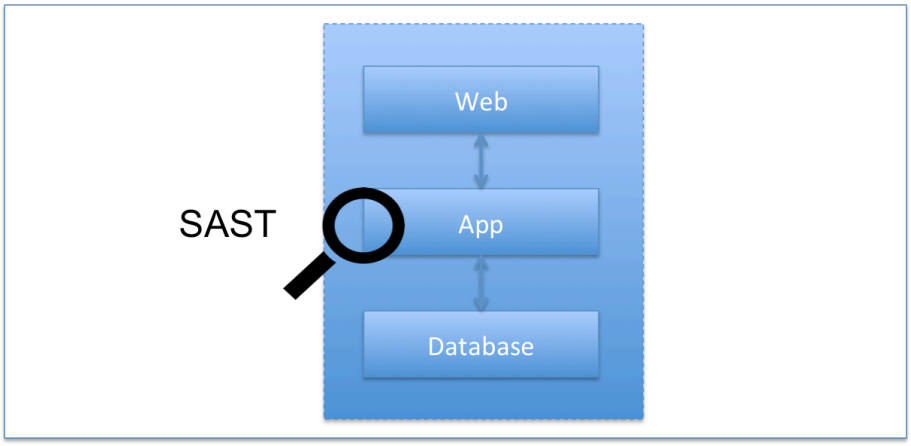

Figure 2. Static testing

Static testing is an important part of the arsenal, but it also has its challenges. These include:

Too many warnings. A static test report on a reasonably sized application can present many thousands of warnings – far more than a development team can (or should) spend time addressing. While SAST tools do attempt to score different types vulnerabilities in order of importance, the scoring is approximate at best. Literally anything in the report could be dangerous – if it weren't it would not be in the report at all.

Lack of visibility into third-party code. Scanning only works on the code it can see. An application may include opaque third-party code directly; or, in

today's loosely-coupled world, it may call on an opaque web service or other network API.

Lack of understanding of the environment. The application's own source code is only a portion of the overall stack. Operating system configuration, error handling, authentication systems, and a host of other factors all contribute to the overall security posture of a web application. Where a dynamic test exercises the entire stack, a static test puts a microscope on just one element.

3.3 Interactive Application Security Testing (IAST)

Gartner recognized an emerging hybrid of dynamic and static test methodologies, which they named Interactive Application Security Testing. The basic idea of "interactive" testing is to link the dynamic test that probes at the web layer to instrumentation that lives inside the application stack.

The typical implementation of interactive testing promises two benefits: first, to guide dynamic tests toward areas that are potentially more vulnerable, and second, to indicate which areas of source code are at fault when a dynamic test finds an issue.

The idea behind Interactive Application Security Testing is brilliant. Combining the best of the DAST and SAST worlds should, in theory, help overcome the limitations of both, allowing testers to find more high-priority vulnerabilities faster. Unfortunately, there are significant practical issues that get in the way of realizing the theoretical benefits of IAST. These issues include:

Installation of invasive software agents. The instrumentation portion of an interactive test suite is generally an agent that runs on the web server or on the application server. As such, the instrumentation itself can affect the performance of the application. In the worst case the difference between the instrumented application and the native application may mask real vulnerabilities and introduce spurious ones.

Limited technology support. In order to instrument a web server or application server, an interactive testing system must be designed and tested for a particular vendor, a particular technology, and a particular version. Developers who use innovative technologies, or even those who update to the latest recommended versions of common software stacks, have to wait for interactive test tools to catch up to them (or more likely, release their applications without running interactive testing).

Single-vendor syndrome. Mature IT shops have already spent significant money on SAST and DAST tools. In most cases the tools have been bought from different vendors, largely because the leaders in one approach are not strong in the other. To get the benefit of IAST you have to commit to a full single-vendor suite, which usually means compromising on other dimensions.

Lack of visibility into third-party code. The static portion of an IAST regime has all the same issues as SAST, as discussed above.

4 Adding Observability of SQL Traffic

DB Networks takes a novel approach to increase the effectiveness of Web application penetration testing. We begin with the realization that *the assets attackers desire are stored in databases*.

Customer information, of course, heads the list of sensitive data that attackers want to access, and customer information is always stored in databases. But even when the data in question is not customer credit card numbers or social security numbers, the database still plays a crucial role. Modern web sites are built on use database content-management systems, so the way for a hacktivist to deface your home page likely requires overwriting records in your content-management database. It's also typical for IT to use databases to store its own passwords and authorization credentials; compromising those databases can give an attacker *all* the keys to the kingdom.

Current DAST and SAST approaches, and even newer IAST systems, focus on the web and application tiers. They identify some of the vulnerabilities that let attackers through your outer layer defenses. They don't watch the database tier, though, so regardless of how much of this sort of testing is done, there's no way of knowing what attacks are *actually* getting through to the database layer.

DB Networks' DBN-6300 observes and decodes all the SQL transactions between the application and the database. It receives a copy of network traffic from a network tap or port mirror. The DBN-6300 then fully decodes the incoming SQL transactions and, using advanced behavioral analysis, alerts in real time when it identifies anomalous activity.

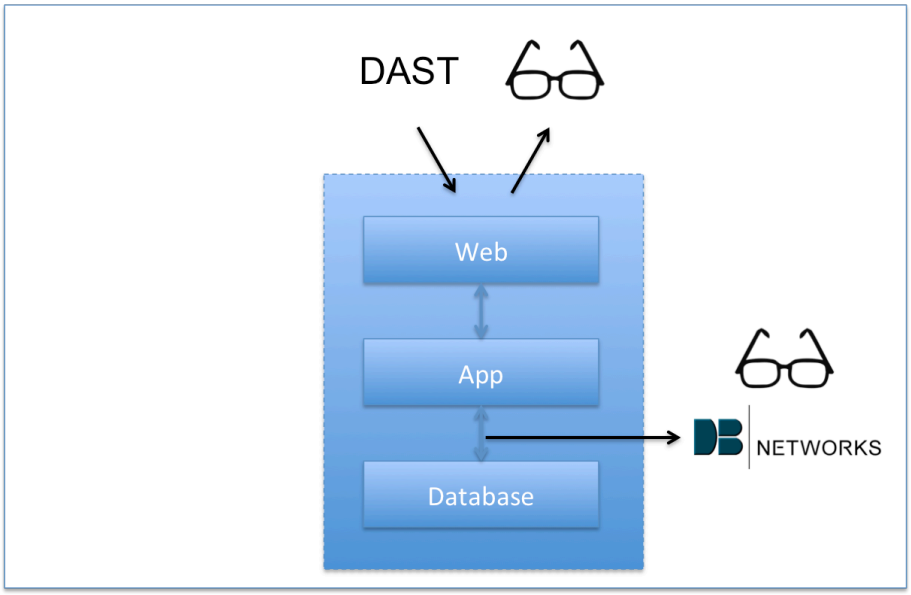

Figure 3. Dynamic testing with DB Networks visibility

4.1 A Case Study: Noticing the Near-Miss

In one typical situation, a CISO at a large health care provider commissioned a third-party pen tester to evaluate his organization's attack surfaces. The tester probed for SQL Injection vulnerabilities – consistently listed by the Open Web Application Security Project as the #1 most serious threat to web applications – but found none that he could confirm, so he reported the SQL Injection threat as relatively low for this site.

In most cases the story would have ended there. This time, though, the CISO had also deployed DB Networks' DBN-6300 to monitor all database SQL transactions. The DBN-6300 alerted during the pen test. It provided the internal security team visibility at the database tier that discovered an extremely serious vulnerability – one that the pen tester was exercising but, being blind SQL injection, not seeing the results.

Here's what happened. The penetration test probed the web application using invalid URLs. Following best practice, the application did not return error messages that would have assisted the "attackers" in better understanding the system. Further, every invalid URL was logged by the web server's error-handling code, allowing for future forensic analysis. However, the error-handling code used a simple database to store these invalid URL strings; and this internal error-handling code was vulnerable to SQL injection.

The testers' efforts were not returning any visible results, but behind the scenes they had actually succeeded in sending SQL command strings by way of the

error-logging code path. It gets worse: the error log was stored as "just one more table" in a database instance whose primary purpose was to store very sensitive data (a choice the developer had made to save on database license expense). So a cleverly formed URL string could read, write, or delete the more sensitive data.

During the short duration of the penetration test, the precise combination of characters needed to compromise the database was not found, and as a result the tester had no idea how close he actually had come to cracking the system wide open. What the DBN-6300 revealed was that the SQL injection attempts had gotten within *one character* of compromising the database completely. And here is where the attacker-defender mismatch becomes crucial: while the dynamic test did not continue long enough to find the magic string, any patient attacker would certainly have found it.

This story does have a happy ending. Using the information gathered from the combination of dynamic testing and DBN-6300 visibility, the organization quickly fixed their error handling code.

This testing experience is similar to what airline safety agencies call a "near miss":

> an unplanned event that did not result in injury, illness, or damage – but had the potential to do so. Only a fortunate break in the chain of events prevented an injury, fatality or damage. (-*Wikipedia*)

A near miss is fortunate in two ways: first, that no damage was done, and second, that it provides the opportunity to put corrective action in place before the same failure presents itself again.

In a dynamic testing environment, near misses are extremely valuable. They point to security flaws that you would not otherwise know about but that are easily exploitable by determined attackers. Contrast the near miss for example with the thousands of "vulnerabilities" reported by a SAST tool – most of which are not really exploitable because they are hidden behind other protections – or the handful of successful attacks found by a DAST tool, which are real but which represent only a small subset of real threats. Figure 4 shows this relationship.

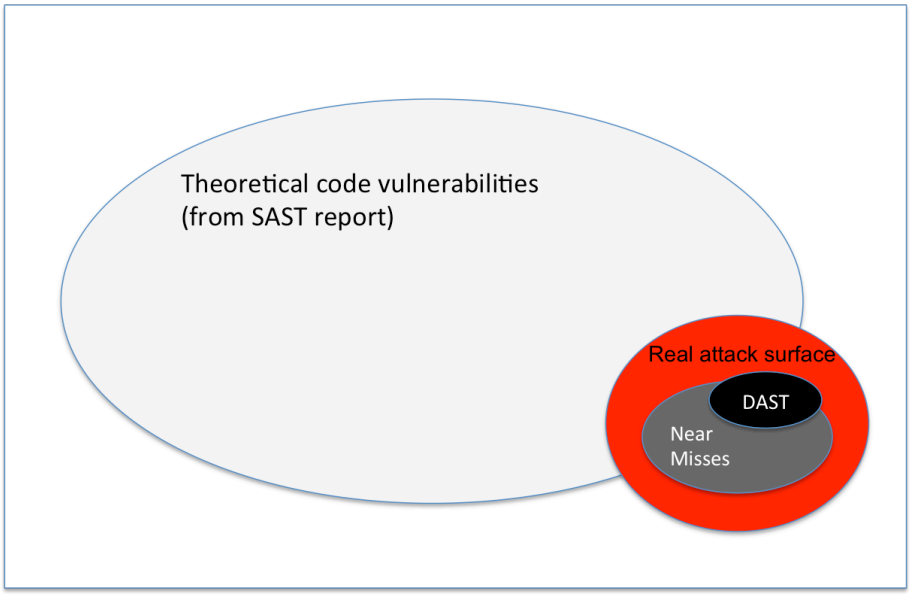

Figure 4. Using the near-miss to find important vulnerabilities

4.2 Accelerating Software Certification

The above case study showed how DB Networks' technology helps make an organization more secure. The same technology can also save time and money in an organization's software development life cycle (SDLC).

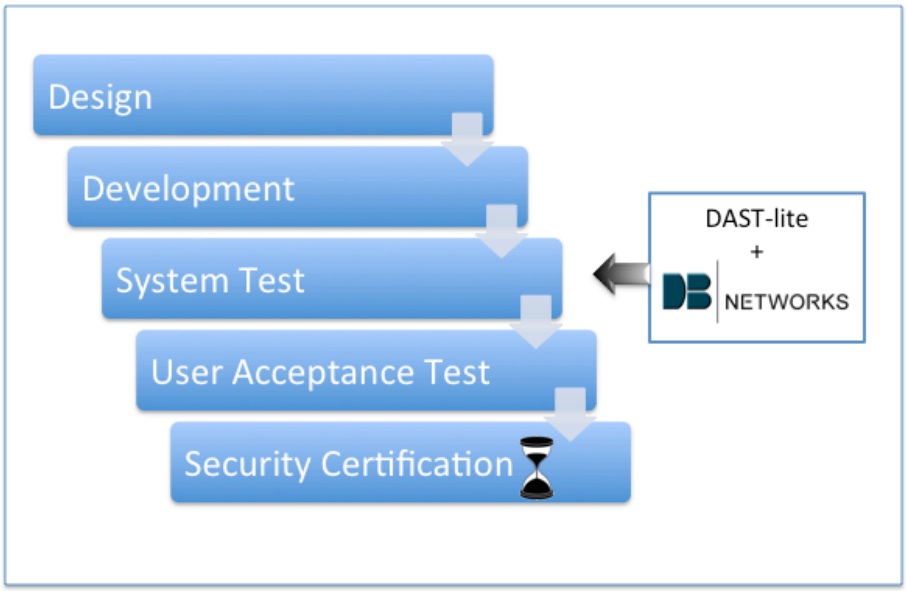

Figure 5. Augmenting the software development life cycle

A portion of a typical life cycle is shown in Figure 5. Certifying an application's security properties, which may happen either during User Acceptance Testing or in a separate step as shown here, can be a costly and time-consuming exercise.

Security vulnerabilities found late in the cycle are the most difficult to repair. Many vulnerabilities only show up in the certification step, however, because expensive dynamic testing and penetration testing are often run only in the certification step.

Introducing a lightweight dynamic-test step during the system test phase, with visibility provided by DB Networks, can find security vulnerabilities in the early stages of the development cycle, improving efficiency and helping make deployment schedules more predictable.

4.3 Intelligent Continuous Monitoring and Protection

It's often impractical to repeat a full code scan, penetration test, or other application security test for every change in your code base, your environment, or the external threat landscape. Application security tests, no matter how effective, always measure a snapshot of a single point in time.

Unlike other testing technologies, DB Networks' DBN-6300 offers the option of carrying over directly into production deployment once the penetration testing concludes. In a production system, you can deploy DB Networks' DBN-6300 appliances to provide real-time continuous analysis of SQL traffic. Using advanced behavioral analysis, the DBN-6300 will alarm the instant a rogue SQL transaction occurs. DB Networks can be used for application security testing only, in production only, or in both environments for maximum security.

Learn more

To find out more about how to gain deep visibility into your database networks and prevent database data loss, contact DB Networks at +1-800-598-0450 or email info@dbnetworks.com.

DB Networks®
15015 Avenue of Science
Suite 150
San Diego, CA 92128
Tel: +1-800-598-0450
www.dbnetworks.com

Version 03132018