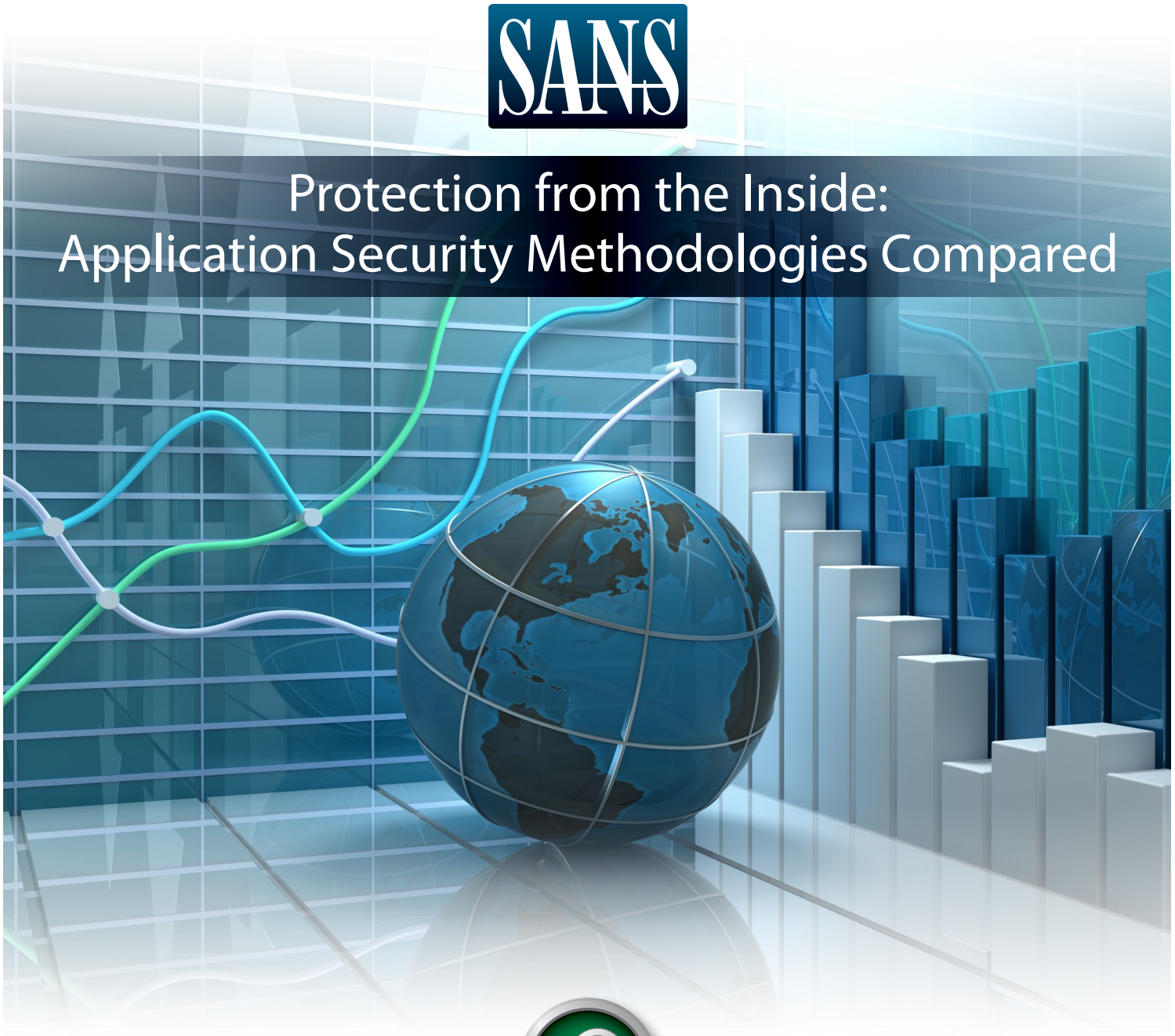

Protection from the Inside: Application Security Methodologies Compared

A SANS Product Review

Written by Jacob Williams

April 2015

Sponsored by
HP

Introduction

Web applications are a common source of compromise and the vector attackers often use to penetrate a network. They are often complex and developed by those with little understanding of security in software development. Fundamental misunderstandings of security by those responsible for custom code development in our environments too often lead to compromises, with disastrous results.

What will help keep organizations safe, while allowing the flexibility that web application designers demand? Many in the industry suggest that a web application firewall (WAF) is the answer. WAF filtering technology normally sits in front of a web application, inspecting incoming traffic for attack patterns and preventing those inputs from reaching the web application itself. However, a WAF is only as good as its signature base and pattern-matching engine, and bypassing WAF filtering is an active topic of security research.

So begins a cat-and-mouse game, where attackers research new and clever ways to create malicious inputs that cause undesired application behavior while bypassing the WAF's input filters. (After all, the WAF doesn't truly understand what the application will do with the input, so it must block any input that *could* cause an exploit, whether or not it would.)

What if, rather than monitoring for potentially malicious inputs, you could monitor the application itself and block only those inputs that actually changed the behavior or operation of the application? Such an approach would render filter bypasses impossible and increase true positive rates.

This approach is the idea behind runtime application self-protection (RASP), which Gartner defines as a security technology built or linked into an application runtime environment to control execution and prevent real-time attacks.[1] HP Application Defender (hereafter, "App Defender"), the focus of this review, adds this protection to servers hosting any Java or .NET application by loading an agent into the Java Virtual Machine (JVM) or .NET Common Language Runtime (CLR). The agent establishes program points App Defender uses to identify attacks in the application code itself, providing RASP functionality without touching the application code. Implementing App Defender is a simple matter of installing the agent and restarting the application server; the agent instruments the application at strategic locations in the code, automatically protecting vulnerable library calls before attackers can exploit them.

In this review, we compare App Defender to an unnamed WAF, examining their respective preventive and detective capabilities. Where WAFs simply put up a wall in front of the application, RASP protects the application from the inside out. Its instrumentation of the runtime environment enables the mitigation of vulnerabilities without access to the source code. When tested against the WAF, App Defender caught more events, reduced false positives and improved visibility into vulnerabilities, including those weaknesses we didn't know we had.

[1] "Runtime Application Self-Protection (RASP)," Gartner IT Glossary, retrieved February 27, 2015; www.gartner.com/it-glossary/runtime-application-self-protection-rasp

Before examining the results of our testing, we provide a definition of WAFs and their shortfalls and explain why RASP frameworks matter. WAFs intercept requests to a potentially vulnerable web application applying rules to evaluate whether a request contains input that might exploit the application; this process requires tedious configuration, and WAFs may fail open under high load, leaving web applications vulnerable at precisely the moment when they most need protection. For a WAF to function at its peak, you need to know what the vulnerable inputs to the web application are so you can apply the appropriate protections to these input fields.

In contrast, RASP frameworks integrate with the underlying code libraries and protect the vulnerable areas of the application at the source level. When a client makes a function call containing parameters that might cause harm to the web application, RASP intercepts the call at runtime—logging or blocking the call, depending on the configuration. This method of protecting a web application differs fundamentally from a WAF.

Consider this statement from Rackspace, which offers the option of protecting web applications with a hosted WAF:

"The [vendor name redacted] WAF will fail open, which means that while traffic to the web application will not be blocked in the event of a failure, traffic will also not be monitored for web attacks."

Uses for WAFs

Security consultants have a love–hate relationship with WAFs, because they are usually most effective the day they enter service and gradually become less effective over the course of subsequent months.

The reason for this decline in effectiveness is that WAF deployment often takes place in response to some penetration test or security incident after the organization performs a cost analysis and decides a WAF deployment is less expensive than fixing the application's source code. (In some cases, this decision is easy: The source is simply unavailable.) During the WAF deployment, everyone involved understands exactly which form fields and inputs are vulnerable and to which attack categories, but over time, this knowledge fades.

Many organizations lack the in-house expertise to conduct penetration tests every time they change the web application or WAF configuration (and miss the opportunity to ensure a vulnerability was not introduced). Other departments in the organization inevitably expand the WAF's role to protect additional applications, stressing the WAF beyond its original specifications. Eventually, the WAF is no longer able to keep up with demand under heavy load and fails. In order to ensure that web applications remain available, vendors typically set WAF appliances to "fail open" by default so the application continues to function. The worst part about this scenario is that the attacker can trigger a high-load condition remotely by sending abnormally large volumes of traffic, thereby triggering the fail-open.

RASP Capabilities

RASP and WAFs protect applications in fundamentally different ways. Consider WAFs to be the gloves and masks medical personnel use when dealing with an infected patient. These barriers might block germs from entering the body, but gloves and masks do not protect against all infections. RASP, in this example, is a vaccine that protects the application from attack, even if bad inputs get in.

Table 1 compares the basic characteristics and functions of RASP tools and WAFs.

Table 1. Comparison of RASP and WAF Characteristics

	RASP	WAF
Accuracy	Detection of malicious input only when passed to library calls where exploitation would occur. Monitors inbound and outbound data and logic flows.	Detection based on naïve pattern matching, without considering whether the input would be passed to vulnerable code.
Time to Value	No need to know locations of existing vulnerabilities in application code; can act as a virtual patch against a vulnerability.	Requires extensive testing and configuration to adequately cover the application.
Reliability	Will not fail open under high load—code is always instrumented, regardless of server load.	Single point of failure; likely to fail open under high load, leaving the formerly protected web application vulnerable.
Platforms	Any instrumented application.	Web applications.
Visibility	May provide detailed feedback to developers to show how to remediate code vulnerabilities.	Offers no detailed insight into the application.
Network Protocols	Protocol agnostic; handles HTTP, HTTPS, AJAX, SQL and SOAP with equal ease.	Must be able to understand the application's network communication.
Language Coverage	Theoretically language agnostic but requires complicated, language-specific builds—currently known products support .NET and Java.	Language agnostic; not bound by programming language type.
Maintenance	Automatically understands changes to the application.	Can gain application context through training only; requires regular maintenance to stay in sync with application changes.

The rest of this paper discusses the testing of threat detection in App Defender and the generic WAF. Table 2 shows the key differences in their performance.

Table 2. Key Differences Found in Testing

App Defender	WAF
Detected more attack classes	Lack of application-level instrumentation allowed some attack classes to escape detection
Accurately recognized most attack types	Generated some false positives, particularly when evaluating SQL injection suspects
Detected unknown vulnerabilities	Unable to detect unknown vulnerabilities in code, such as unhandled exceptions
Examined output as well as input	Focused solely on input
Lacks granular rule configuration	Granular configuration options make implementation complex

We discuss specific findings in the following sections.

Attack Classes Tested

We tested both App Defender and the WAF for their capability to detect the common classes of attacks, including SQL injections, cross-site scripting (XSS) and forceful browsing. The results appear in Table 3; a detailed discussion of our findings in each class follows the table.

Table 3. Comparison of Attack Classes Detected by App Defender and a Traditional WAF

Attack Class	App Defender Detection	WAF Detection
Cross-Site Scripting (XSS)	Yes	Yes
Command Injection	Yes	Yes
ShellShock	Yes	Yes
Query Injection	Sometimes; detected SQL injection strings only when passed to queries, but missed XPath injection	Excessive; naïve pattern matching detected all inputs, including false positives
Unhandled Exception	Yes	No
Missing Content-Type	Yes	Yes
Missing Accept Header	Yes	Yes
Unsupported Method	Yes	Yes
Vulnerability Scanners	Yes	Yes
Forceful Browsing	Sometimes; only for configured extensions	Superlative; offers better configuration options than App Defender
Method Call Failure	Yes	No
Sensitive Data Disclosure	Yes	No

It is also worth noting that App Defender is able to provide specific information, making remediation efforts much more efficient. A WAF cannot provide this insight.

Cross-Site Scripting

Cross-site scripting (XSS) is an attack where JavaScript input sent by the attacker is then reflected back to a website user. This input may then be used to steal authentication cookies from the user or redirect them to a malicious website.

The App Defender agent detects XSS attacks with little trouble. Our testing showed it detects an XSS attack by matching on the string **`<script`** anywhere in a request parameter and preventing the upload of any input containing this string. App Defender effectively prevents the insertion of new XSS input to the web application, but at the risk of triggering a false alert if users are allowed to upload JavaScript (to a message board, for instance).

Although App Defender does a commendable job defending against reflected XSS and preventing the insertion of new, stored XSS, it cannot shield against stored XSS attacks that rely on preloaded malicious content transferring from an unprotected database to a protected one. The WAF also failed to detect this stored XSS attack. In order for any solution to detect this type of attack, it would have to examine all JavaScript returned to the user for potential malicious applications, which would likely have either abysmal detection rates or astronomically high false positives and a large impact on availability.

Command Injection

Command injection is one of the most potentially damaging attacks on a web application; it occurs when developers accept untrusted user input and use it as part of a shell command without first filtering out dangerous characters. Unlike XSS (which attacks the browser) or SQL injection (which obtains data), command injection goes after the server directly, executing code in the context of the user running the web application. Unfortunately, some administrators still run their web servers as the root user and, in such a case, command injection vulnerabilities result in total compromise of the machine running the web application. Even in cases where a non-root user runs the web application, command injection vulnerabilities are devastating and allow attackers to pivot their attacks from the web server to internal hosts.

The App Defender agent detected command injection reliably, even when the actual command injection attack would not have been successful in executing code on the server because of syntax issues. We attempted multiple filtering evasions, but App Defender caught the attack every time. The WAF also caught the command injection attempts because they contained character combinations that benign user scripts typically lack.

ShellShock

The ShellShock vulnerability is akin to command injection, exploiting the behavior of the **bash** shell to execute arbitrary code when malicious function definitions pass to the shell through User-Agent strings or other request parameters.[2] Although web servers with up-to-date patches should be able to fend off ShellShock, the App Defender action of blocking for this attack by default is definitely appropriate.

App Defender detects ShellShock by searching for the string **( )** [space] **{** at the beginning of a form field, but its documentation does not address whether App Defender would detect the string *elsewhere* in a request.[3] We confirmed that this string correctly triggered alerts for ShellShock attacks, whatever its position in the request parameter.[4] The WAF also detected these characters at all positions in a form field and identified the ShellShock attacks.

RASP offers the unique capability to perform context-sensitive detection, whereas traditional WAF solutions focus on all input fields or are limited to fields defined in advance.

Query Injection

Query injection uses inputted data to force the targeted system into an error condition that returns unintended data to the attacker. This can take the form of queries sent in real time or stored for future use. Query injection commonly appears in attacks against SQL databases but can occur in any data environment that supports a query language.

SQL injection attacks pass input that changes a SQL query's developer-intended structure into a pathway for the attacker. App Defender attempts to tokenize the query and look for **1 = 1** statements and other unusual SQL syntax, which are indicative of common SQL injection patterns. (App Defender's documentation correctly cautions that developers with lazy programming habits may make legitimate use of queries containing these strings.)

RASP offers the unique capability to perform context-sensitive detection. With traditional WAF solutions, detection must focus on all input fields or be limited to those fields an administrator defines in advance.

[2] ShellShock Vulnerability Checker, https://shellshocker.net; "Alert (TA14-268A)," US-CERT website, September 23, 2014; www.us-cert.gov/ncas/alerts/TA14-268A; "Vulnerability Summary for CVE-2014-6271," National Vulnerability Database, December 23, 2014; https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271

[3] There is almost no reason for this character string to appear in a legitimate request, so the risk of false-positive alerts is low.

[4] At the time of testing, the App Defender documentation did not reflect this capability.

The problem with the WAF approach is that creating a usable configuration requires some prior knowledge of the vulnerable fields, while the RASP approach suffers from potential false positives thanks to naïve pattern matching. However, because App Defender inspects input at the time it passes to the database query strings, it eliminates false-positive detections for fields not vulnerable to SQL injection so the administrator does not waste time chasing attacks that don't apply to his environment. On the other hand, the WAF still detected this as an attack, resulting in a false positive, as shown in Figure 1.

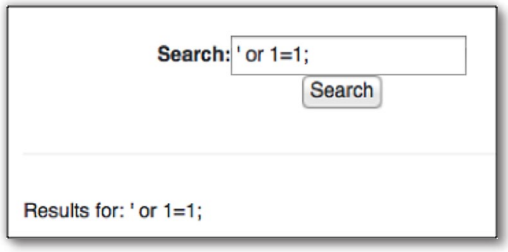

Figure 1. False-Positive SQL Injection

In addition to testing for naïve SQL patterns (such as **`or 1=1`**), we attempted several other tautologies, hoping to extract additional data from the database. Although such tautologies sometimes bypass WAF technologies, App Defender caught them all. The App Defender agent appeared to match our strings against blacklisted parameters, including any quotes and Boolean operators passed to the SQL query string. This result makes sense, because any query including a Boolean operation would necessarily change the structure of the query.

However, the naïve pattern matching of WAFs may sometimes be the best tool for the job; one example of this is testing for XPath injection, another class of query injection attacks.[5] App Defender failed to detect a basic XPath injection string that the WAF detected. (Our analysis of the underlying code appears to indicate that when the input strings do not pass directly to a SQL library call—as was the case in this example—the App Defender agent does not detect the query injection.) Figure 2 shows a typical XPath injection using tautologies.

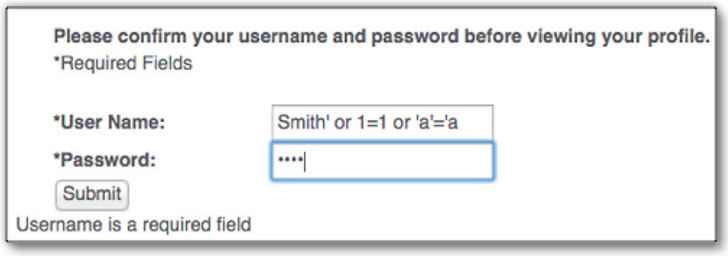

Figure 2. XPath Injection Using Tautologies

[5] The XPath query language extracts data from an XML database—used by some web applications to store data—in much the same fashion as SQL.

App Defender's application-level instrumentation detected all of our standard SQL injection attempts but missed query injection attacks that did not use standard SQL libraries. (Although our WAF also caught our SQL injection attempts, instrumentation through RASP offers better theoretical protection against SQL injection attacks than do firewall technologies like WAF. We expect RASP's low-level instrumentation would likely catch future evasion techniques that use SQL injection to bypass WAF.)

Forceful Browsing

App Defender protected successfully against forceful browsing attacks, although configuring this protection is not what some experienced users might anticipate or desire. The App Defender agent checks for these file extensions and only these extensions: **`.log`**, **`.bak`**, **`.old`**, **`_log`**, **`_bak`** and **`_old`**. Other extensions could reflect forceful browsing as well; extensions commonly seen during penetration tests include **`.1`**, **`.2`**, **`.2015`** (the year) and **`.012015`** (the month and year). Although it's not possible to configure App Defender to detect these additional extensions, most WAFs can block access to all such extensions.

Method Call Failure

Method call failures exist only in programming languages that support object-oriented programming. These errors usually occur when an object of one type improperly calls a method from its base class. Bugs of this nature are particularly dangerous because they may remain hidden if they do not generate error logs or other obvious output. App Defender can detect method call failures when SQL exceptions occur during database transactions, an example of a method call failure that occurs without displaying any output to the user.

Testing for method call failures can identify underlying vulnerabilities as well as attacks. During our review, App Defender detected a number of previously unknown errors in the web application we used for our tests. None of these errors appeared in standard web application logs, and none displayed to the user. Although such errors in a web application are often invisible to users, an attacker might be able to exploit them by using novel or unusual attack code. App Defender detects these errors by instrumenting the API calls themselves, rather than examining the output, making it useful for identifying underlying application errors in a production environment.

An example of App Defender's output from a method call failure appears in Figure 3.

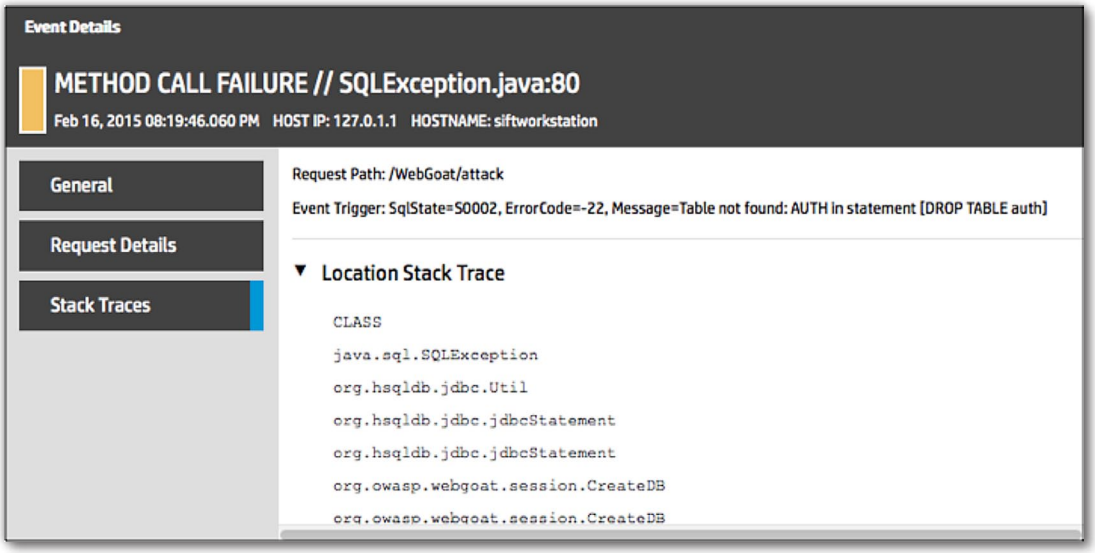

Figure 3. Method Call Failure

In contrast, no WAF, even one that filters responses, could detect method call failures if the web application did not display errors.

Unhandled Exceptions

Unhandled exceptions can give the attacker insight into how an application functions. One potential risk is that the stack trace output from the server may reveal the application's use of vulnerable libraries to the attacker and provide him with a roadmap for his assault.

We deliberately configured the web application we tested to throw unhandled exceptions, mimicking a common misconfiguration of web applications. Figure 4 displays a typical App Defender stack trace from an unhandled exception.

Even though exposures of sensitive data can lead to big fines, developers often do not realize where their apps have stored sensitive information.

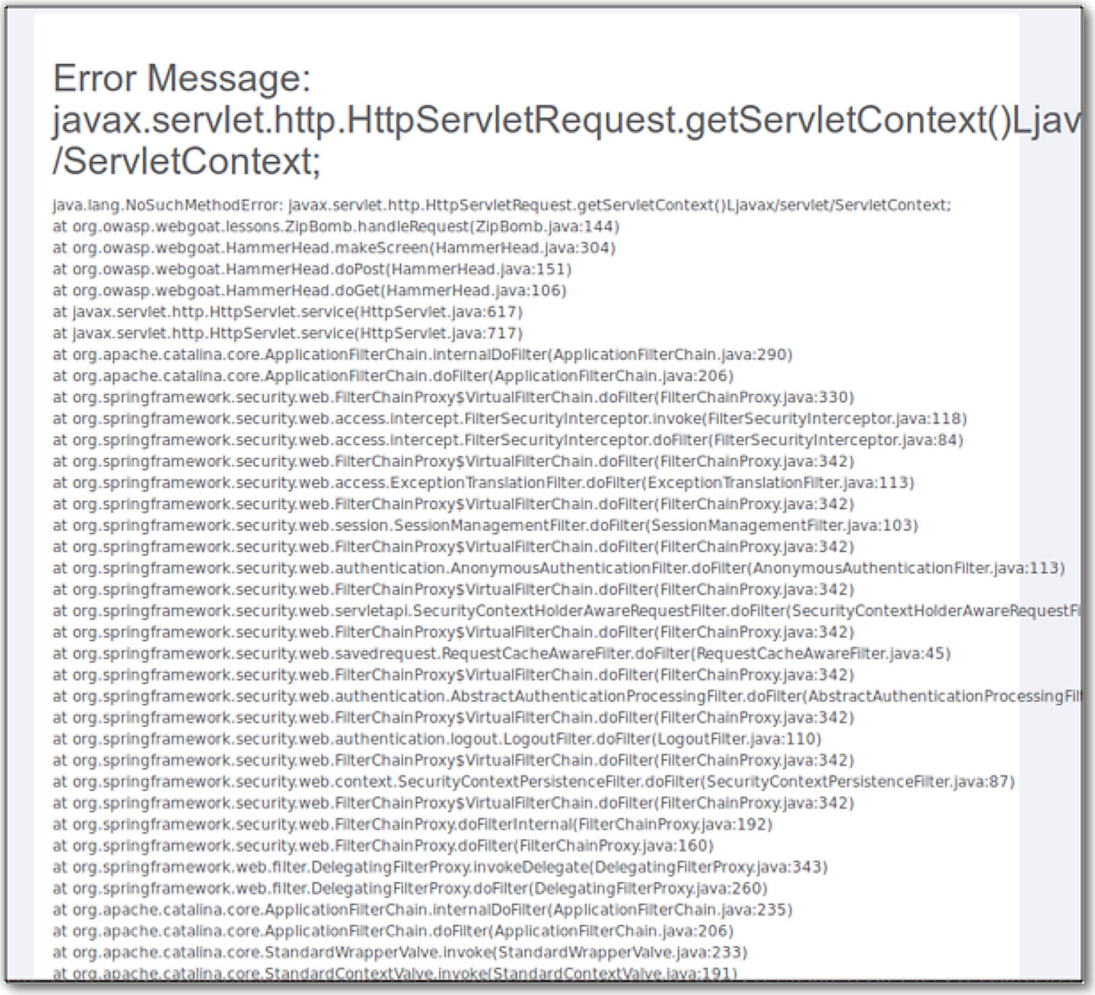

Figure 4. Unhandled Exception Revealed in Stack Trace

App Defender was able to detect these exceptions. Meanwhile, the WAF did not detect the unhandled exceptions. No such thing as a universal input that triggers an unhandled exception exists, leaving the WAF (focused on input) with no way to detect such exceptions.

Privacy Violations

Even though exposures of sensitive data can lead to big fines, developers often do not realize where their apps have stored sensitive information, and even audited applications may disclose sensitive data while under attack. Applications may write sensitive data to their log files, especially when the application fails in response to attack input. An attacker can then use his access to the web application to examine the application log files to retrieve sensitive data, including payment card numbers.

App Defender has two rule categories for detecting sensitive information: credit card numbers and Social Security numbers (SSNs), two of the data items attackers are most likely to want. The default action for both rules is to rewrite (or mask) the output, an operation that offers an optimal blend of usability and security. In the event the output of sensitive data was the result of normal operation and not an attack, the operation still proceeds, but with the sensitive data masked in the log file.

Figure 5 shows an example of App Defender detecting sensitive data as it is sent to server log files.[6]

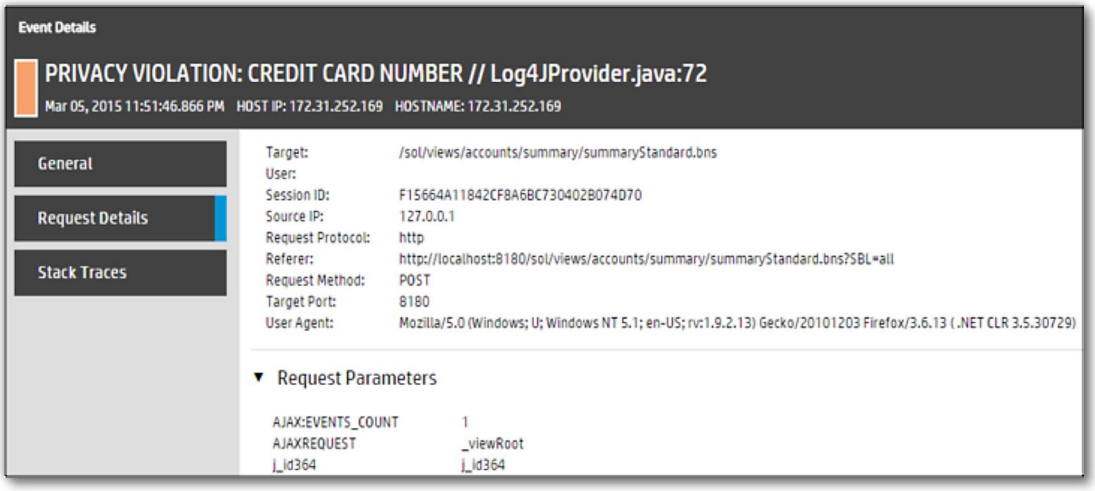

Figure 5. Sensitive Data Detection

Detecting and masking such values at the library call level could prevent a compliance violation in breach scenarios where sensitive data is not blocked even though the application itself was exploited. The credit card and SSN algorithms App Defender used matches the patterns; for example, ###-##-#### for an SSN. However, organizations need to configure App Defender to recognize and mask other patterns—such as the filtering of medical record numbers or bank account numbers, features currently missing from the product.

Not only did App Defender detect the sensitive data where the WAF did not, but also we believe its functionality may be useful to auditors who wish to find in the application unanticipated locations that hold sensitive data.

[6] App Defender's documentation incorrectly stated that it masks sensitive data in HTTP responses; actually, it masks such data only when writing to external files, because masking sensitive data returned in HTTP responses can cause problems with database updates.

Unsupported Method

App Defender protects against requests that employ infrequently used HTTP request methods; it treats any HTTP methods other than GET, POST and PUT as "unsupported" methods. (However, some poorly coded clients may send these request methods in mixed case, triggering false alerts.) Other requests may also be required for protected applications, which might legitimately support methods such as DELETE, OPTIONS or TRACE. RASP products need to support such functions in their libraries and should provide a means to configure alternative options to better handle such cases.

(Then again, our voyage through the configuration notes for the WAF we used made it clear that enabling the detection of unsupported HTTP requests on a traditional WAF is possible, but not for the faint of heart.)

Both App Defender and the WAF successfully detected unsupported HTTP methods. App Defender's information screen for this attack appears in Figure 6.

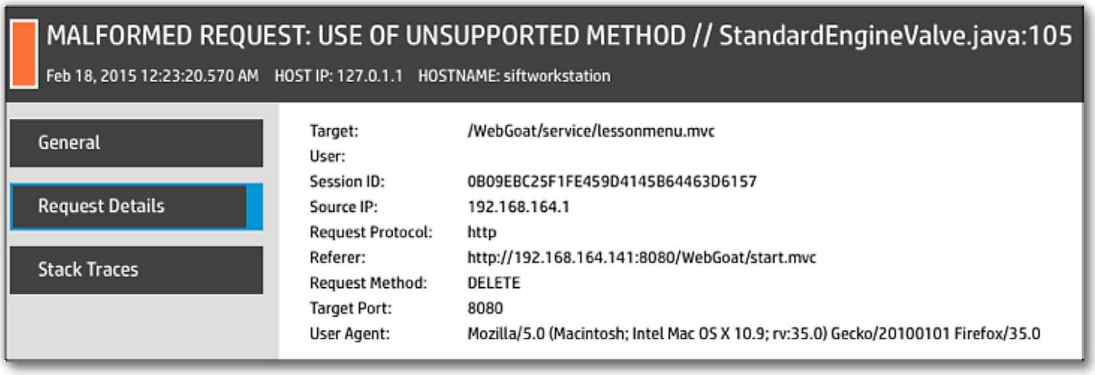

Figure 6. Unsupported HTTP Request Method

Missing Content-Type

Attackers may bypass file type upload restrictions by not specifying anything for content-type in the request header. Both App Defender and the WAF were able to detect this attack vector with ease. App Defender's display of one such attack appears in Figure 7.

Figure 7. Typical Display of a Missing Content-Type Attack

Missing Accept Header

HTTP does not strictly require accept headers—which tell the application server what languages or character sets the client supports—but virtually all legitimate clients send them as part of the request. Many automated scanning engines fail to include these headers, making this a potential early warning sign of an automated attack. Both App Defender and the WAF detected the missing Accept header. App Defender's detailed output is shown in Figure 8.

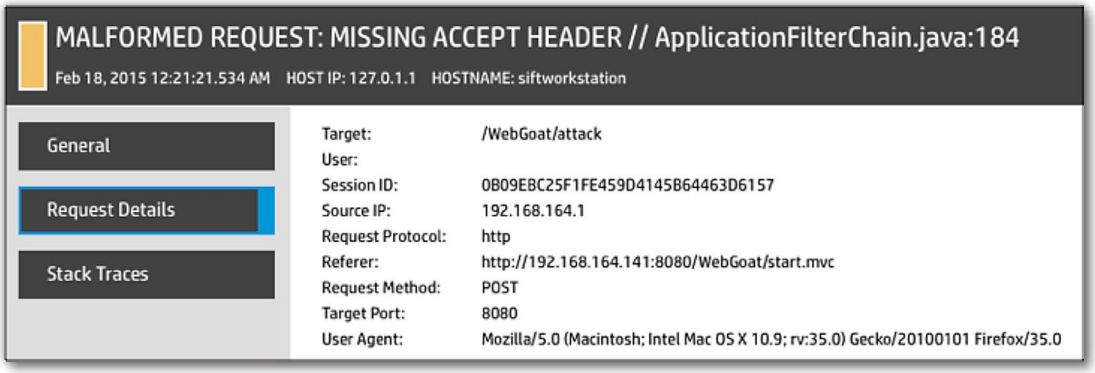

Figure 8. Missing Accept Header in HTTP Request

Known Vulnerability Scanners

Vulnerability scanners often embed a predefined user agent string in their probes and—for obvious reasons—security analysts seek to prevent attackers from using them. Both App Defender and the WAF detected the user agent strings of known vulnerability scanners, although the list of vulnerability scanners App Defender can detect is not documented and not configurable. For this reason, some security analysts may prefer the flexibility of a fully configurable WAF solution. Figure 9 shows a typical message generated by App Defender when detecting a vulnerability scan through examination of the user agent string.

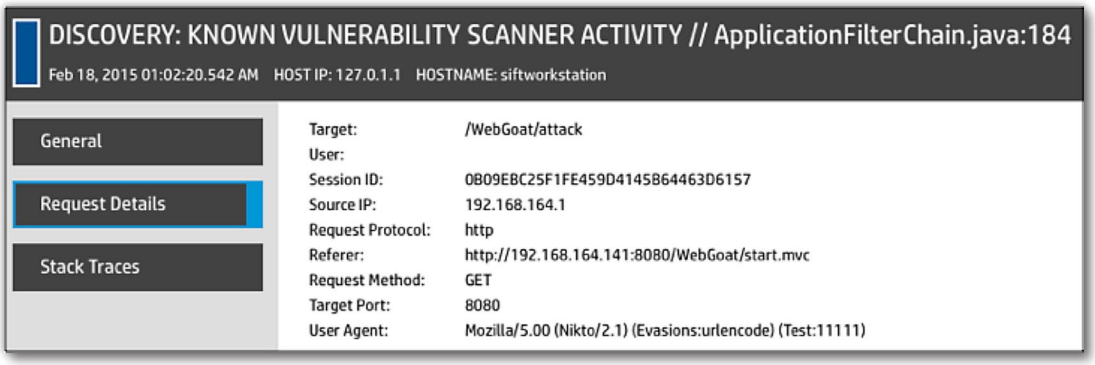

Figure 9. Vulnerability Scanner Detected

In summary, our tests proved App Defender's capability to protect against attack classes that the WAF was unable to even see.

Conclusion

HP Application Defender outperformed the traditional WAF in our tests by protecting against vulnerability classes that the WAF missed. We can comfortably extrapolate the results from the WAF we tested to other WAF products that inspect the parameters passed in HTTP requests.

Moreover, App Defender proved its worth in detecting actual vulnerabilities we didn't know we had in our applications and provided specific insight for remediation—something a WAF was simply unable to accomplish. App Defender offers superior insight into the applications it protects, doing so far better than a WAF.

App Defender detected actual vulnerabilities we didn't know we had in our applications, while providing specific insight for remediation.

App Defender's capability to instrument at the API layer allows it to detect attacks the traditional WAF missed. It reported fewer false positives than the traditional WAF, thanks to its capability to perform context-sensitive matching.

Although we were unable to configure App Defender as granularly as a true web application expert might like, most commercial WAF engines also lack such granular configuration capabilities, so this capability is not truly a market differentiator. Custom configurations of App Defender may provide more granular configurations than are available through the standard user interface, alleviating some of our concerns.

Thanks to its plug-and-play approach, App Defender stands above any traditional WAF, by protecting web applications out of the box with minimal (if any) configuration needed. This feature could substantially reduce risk by enabling application protection immediately upon deployment. Finally, App Defender protected against unhandled exceptions, method call failures and sensitive data: three attack classes the WAF couldn't even *see*. It had fewer false positives for SQL injection thanks to the context-sensitive protection.

Those looking to adopt a WAF to protect their potentially insecure web applications should examine RASP solutions, such as App Defender, as effective application protection alternatives. Organizations that have already deployed a WAF but find that attackers are bypassing it or experiencing too many false positives should also consider RASP solutions to augment their protection portfolios.

About the Author

Jacob Williams is founder and principal consultant at Rendition Infosec and a certified SANS instructor and course author. He has more than a decade of experience in secure network design, penetration testing, incident response, forensics and malware reverse engineering. Before founding Rendition Infosec, he worked with various government agencies in information security roles. Jake is a two-time victor at the annual DC3 Digital Forensics Challenge.

Sponsor

SANS would like to thank its sponsor: