Understanding and Selecting Runtime Application Self-Protection

Version 1.1
Released: August 29, 2016

Securosis, L.L.C. 515 E. Carefree Blvd. Suite #766 Phoenix, AZ 85085 T 602-412-3051 info@securosis.com www.securosis.com

Author's Note

The content in this report was developed independently of any sponsors. It is based on material originally posted on the Securosis blog but has been enhanced, reviewed, and professionally edited.

Licensed by IMMUNIO

IMMUNIO is a pioneer in real-time application self-protection (RASP), providing automatic detection and protection against web application security vulnerabilities. IMMUNIO augments applications with the necessary protection services and hardens applications against common attacks targeting typical security weaknesses. The company's mission is to make truly effective real-time web protection technology easily available and widely deployed, and by doing so, stop the biggest source of breached data records. For more information visit www.immunio.io.

Copyright

This report is licensed under Creative Commons Attribution-Noncommercial-No Derivative Works 3.0.

http://creativecommons.org/licenses/by-nc-nd/3.0/us/

Table of Contents

Introduction 2

Defining RASP 2

Use Cases 4

Buyer Requirements 5

Technology Overview 6

Integration 10

Buyers Guide 14

About the Author 16

About Securosis 17

Introduction

During our 2015 research project into DevOps, developers consistently turned the tables on us, and asked us dozens of questions on embedding security into their process. We found DevOps provided a very real path to improve application security using continuous automated testing, run each time new code was checked in. However we were surprised to discover the degree in which developers and IT teams are taking a larger role in selecting security solutions, and bringing a new set of buying criteria to the table. For these groups, security products must do more than address application security issues; they need to mesh with continuous integration (CI) and continuous deployment (CD) approaches, with automated capabilities and better integration with developer tools.

But the biggest surprise was that *every* team asked us about RASP, Runtime Application Self-Protection. Each team was either considering RASP, or already engaged in a proof-of-concept with a RASP vendor. This was typically in response to difficulties with WAF and wanting more developer-centric security tools within their certification efforts — both prior to deployment and while in production.

The number of firms who have embraced either CD or DevOps application delivery is a very small subset of the larger market. But we found that once firms started with automated deployments — regardless of which flavor they chose — each firm hit the same issues, and 100% of the firms that had crossed that bridge were looking for better security tool integration. The ability to automate security, the ability to test in pre-production, configuration skew between pre-production and production, and the ability for security products to identify where issues were detected in the code. For these teams, the security tools needed to be as agile as their development processes, and thus seriously looking into RASP solutions.

Defining RASP

Runtime Application Self-Protection (RASP) is an application security technology which embeds into an application or application runtime environment, examining requests at the application layer to detect attacks and misuse in real time. RASP products typically contain the following capabilities:

- Unpack and inspect requests inside the application, rather than at the network or HTTP layer
- Monitor and block application requests; in some cases they can alter request to strip malicious content
- Full functionality through RESTful APIs
- Protect against all classes of application attacks, and detect whether an attack would succeed
- Pinpoint the module, and possibly the specific line of code, where a vulnerability resides
- Instrument application functions and usage

These capabilities partially overlap with white box & black box scanners, web application firewalls (WAF), next-generation firewalls (NGFW), and even application intelligence platforms. So the question you may be asking yourself is “Why would we need another technology that does some of the same stuff?" First glances can be deceiving as these approaches are *not the same* as RASP! It has to do with effectiveness, and RASP offers a number of approaches to detection that enable both more accurate attack identification and far less effort to install and maintain than commonly used firewalls. It has to do with security being fully integrated within an application, so it runs anywhere the application does, and scales right along with the application regardless if running on bare metal, virtualized or cloud environments. And it provides developers with a comfortable medium; security software that behaves as their own code. As one of our readers astutely noted, “If it’s not working inside the application, it’s not RASP.” In this research paper we will discuss why development organizations look for new solutions, but we will also discuss the fundamental advantages RASP provides as an application security platform.

Use Cases

There is no lack of data showing that applications are vulnerable to attack. Many applications are old and simply contain too many flaws to fix. You know, that back-office application that should never have been allowed on the Internet to begin with. These applications are often unsupported, with the engineers who developed them no longer available, or the platforms so fragile that they become unstable if security fixes are applied. In most cases it would be cheaper to re-write the application from scratch than patch all the issues, but economics seldom justify (or even permit) the effort. Other application platforms, even those considered 'secure', are frequently found to contain vulnerabilities after decades of use. Heartbleed, anyone? New classes of attacks, and even new use cases, have a disturbing ability to unearth previously *unknown* application flaws. We see two types of applications: those with known vulnerabilities today, and those which will have known vulnerabilities in the future.

As you might expect, the primary function of RASP is to protect web applications against known and emerging threats; it is typically deployed to block attacks at the application layer, before vulnerabilities can be exploited. Astute readers will notice that these are, by and large, the classic use cases for Intrusion Detection Systems (IDS) and Web Application Firewalls (WAFs). As we said in the introduction, there are other tools in the market that already provide application security benefits; why look for something new? It's not what RASP does, rather how it does it. Let's look at what developers are asking for to help differentiate between the old and the new.

Differing Market Drivers

As RASP is a (relatively) new technology, current market drivers are tightly focused on addressing the security needs of one or two distinct buying centers which are unaddressed by existing security applications. But more to the point, RASP must go the extra mile to protect against these attacks *and* mesh well with the disruptive changes occurring in the development community. RASP offers a distinct blend of capabilities and usability options which makes it unique in the market.

- Demand for security approaches focused on development, enabling pre-production and production application instances to provide real-time telemetry (defects, vulnerabilities, usage profiles) back to development tools
- Need to fully automate application security, deployed in tandem with new application code
- More accurate detection and less management burden to maintain
- Technical debt, where essential applications contain many known vulnerabilities which must be addressed, either while defects are fixed by development, or blocked permanently if the application cannot be altered for various reasons
- Application security supporting development and operations teams who are not all security experts

Buyer Requirements

RASP takes security one large step from "security bolted on" toward "security from within". But to understand how the above drivers change the use cases, you first need to understand how RASP will be used. Let's dive into the deployment criteria that forge the buyer's requirements

- **APIs & Automation**: Most of our readers know what Application Programming Interfaces (APIs) are, and how they are used. Less clear is the greatly expanding need for programatic interfaces in security products, thanks to application delivery disruptions caused by cloud computing. Cloud service models — whether deployment is private, public, or hybrid — enable much greater efficiencies as networks, servers, and applications can all be constructed *and tested* as software. APIs are how we orchestrate building, testing, and deployment of applications. Security products like RASP — unlike IDS and most WAFs — offer their full platform functionality via APIs, enabling software engineers to work with RASP in the manner their native metaphor.

- **Development Processes**: As more application development teams tackle application vulnerabilities within the development cycle, they bring different product requirements than IT or security teams applying security controls post-deployment. It's not enough for security products to identify and address vulnerabilities — they need to fit the development model. Software development processes are evolving (notably via continuous integration, continuous deployment, and DevOps) to leverage advantages of virtualization and cloud services. And with security testing now happening in each phase of the process— development, build, QA, pre-deployment and so on — speed is imperative! RASP needs to not only embed within the application stack to ensure consistency between test and production, but it needs to work quickly.

- **Application Awareness**: As attackers continue to move up the stack, from networks to servers and then to applications, it is becoming more distinguish attacks from normal usage. RASP is differentiated by its ability to include application context in security policies. Many WAFs offer 'positive' security capabilities (*i.e.*: whitelisting valid application requests), but being embedded within applications provides additional application knowledge and instrumentation capabilities to RASP deployments. Further, some RASP platforms *help developers* by specifically reference modules or lines of suspect code. For many development teams, *potentially* better detection capabilities are less valuable than having RASP pinpoint vulnerable code.

- **Pre-Deployment Validation**: For cars, pacemakers, and software, it has been proven over decades that the earlier in the production cycle errors are discovered, the easier — and cheaper — they are to fix. This means testing in general, and security testing specifically, works better earlier into the development process. Rather than relying on vulnerability scanners and penetration testers *after* an application has been launched, we see more and more application security testing performed *prior* to deployment. Again, this is not impossible with other application-centric tools, but RASP is easier to build into automated testing.

Technology Overview

Here we discuss technical facets of RASP products, including how the technology works, how it integrates into an application environment, and the advantages of this approach. We will also outline some important considerations, such as platform support, which will impact your selection process. We will also consider a couple aspects of RASP technology which we expect to evolve over next couple years.

Deployment Models

Most RASP products are deployed as software, within an application software stack. This means they are essentially part of the application, and migrate and scale as the application does. These products work equally well on-premise and in cloud environments. Some solutions operate fully in a cloud replica of the application, as in the virtualization and replicated models mentioned above. Still others leverage a cloud component, essentially sending data from an application instance to a cloud service for request filtering. What doesn't happen is dropping an appliance into a rack, or spinning up a virtual machine and re-routing network traffic through RASP.

Application Integration

RASP works at the application layer, so each product needs to integrate with applications somehow. To monitor application requests and make sense of them, a RASP solution must have access not just incoming requests, but also what resources are used, and how. And as attackers do not focus solely in the application itself, but supporting systems, RASP needs to look at the bigger picture to provide thorough coverage.

There are several methods for monitoring application execution, and each approach deploys slightly differently, gathering a slightly different picture of how the application functions. Solutions are installed into the code production path, or monitor execution at runtime. To block and protect applications from malicious requests, a RASP solution must be part of the application.

- **Servlet Filters & Plugins**: Some RASP platforms are implemented as web server plug-ins or Java Servlets, typically installed into either Apache Tomcat, JBOSS or Microsoft .NET to process requests. Plugins filter requests before they execute functions like database queries or transactions, applying detection rules to each request received. Requests that match known attack signatures are blocked. This is a simple approach for retrofitting protection into the application environment, and is effective at blocking malicious requests, but it doesn't offer the depth application understanding possible with other integration approaches.

- **Library/JVM Replacement**: Some RASP products are installed by replacing the standard application libraries, JAR files, or even the Java Virtual Machine. This method basically hijacks calls to the underlying platform, whether library calls or the operating system. The RASP platform passively 'sees' application calls to supporting functions, applying rules as requests are intercepted. For example, in the case of JVM replacement, as classes are loaded into memory RASP inserts itself between any internal function or calls to external resources by replacing or overloading the native function. Calls are examined for known attacks as well as behavioral anomalies, but only those relevant to the requested action, reducing processing overhead and improving performance. With this model the RASP tool has a

more comprehensive view of application code paths and system calls than servlet filters mention above, and can even learn machine 'state' or sequence behaviors. This approach can provide complete visibility into application function, but it requires the solution to substitute or hijack all relevant interfaces to provide full visibility, balancing depth of coverage with performance.

- **Application Instrumentation**: With this deployment model RASP places sensors at key junctions within the application stack to see application behavior within — and between — custom code, application libraries, frameworks and underlying operating system. This approach is commonly implemented by using native application profiler/ instrumentation APIs to monitor application behavior at runtime. When a sensor is hit, RASP gets a callback and evaluates the request against the appropriate subset of policies relevant to the request and application context. This approach can provide complete visibility into application function, and nuanced application of policies to detect attacks and misuse. However, it does require the solution to monitor all relevant interfaces to provide full visibility, a balancing act between coverage and performance.

- **Virtualization or Replication**: This integration effectively creates a replica of an application, usually as either a virtualized container or a cloud instance, and create a complete view of application behavior at runtime. Functionally this is similar to Application Instrumentation above, but rather than using inspection/introspections APIs or overloading app functions, in this model the supporting runtime environment is fully virtualized so RASP can position itself in-between all call-outs or system requests. has complete visibility of an applications interaction with libraries, the OS and other applications. By monitoring — and essentially learning — application code pathways, all dynamic or non-static code is mimicked in the cloud. Learning and detection take place in this copy. As with replacement, application paths, request structure, parameters, and I/O behaviors can be 'learned'. Once learning is complete rules are applied to application requests, and malicious or malformed requests are blocked.

Detection

How RASP products detect attacks is complicated, with multiple techniques being employed depending upon the type of request being made. In fact, requests and associated parameters are subject to multiple types of inspection. The good news is that RASP is far more effective in detection of application attacks. Unlike other technologies which use signature based detection, RASP fully decodes parameters and external references, maps application functions and 3rd party code usage, and applies polices accordingly. This not only allows for more accurate detection, but helps with performance as which checks are performed are optimized to the context of the request and execution path within the code. As rules are enforced at the point of use, it is far easier to to understand proper usage and detect misuse.

Most RASP platforms employ structural analysis as well. They understand what framework is in use, and the common set of vulnerabilities the framework suffers. As RASP understands the entire application stack, it can detect variations in 3rd party code libraries — essentially a vulnerability scan — to determine if outdated code is being used. And RASP can quickly vet incoming requests and detect injection attacks. There are several approaches; one such approach is achieved by a form of tokenization — substituting parameters for a tokens — in order to quickly check that any given request matches it's intended structure. For example, tokenizing clauses and parameters in a SQL query, you can quickly detect when a 'FROM' or 'WHERE' clause has more tokens than it should, meaning the query has been altered.

Blocking

When an attack is detected, as RASP is within the application, most products throw and application error. In this way the the malicious request is not forwarded, and the application is responsible for a graceful response and maintaining application state. What is reported to the user is entirely up too the application developers. That said, this can create issues with server side application stability and user experience. RASP offers some capabilities to tune response behaviors, but this should be examined on a vendor by vendor basis.

Language Support

The biggest divide between RASP providers today is their platform support. For each vendor we spoke with during our research, language support was a large part of their product roadmap. Most provide full support for core platforms like Java and .NET; beyond that support still a little spotty with Python, PHP, Node.js, and Ruby.

Another part of the problem is the complexity of the environment; not just the server side but the client side as well. Over the last few years we have witnessed an expanding universe of frameworks, client side utilities, web-facing APIs and changing fashions for data encoding. Consider that RASP needs to parse XML and JSON, handle diverse clients running Javascript and Angular.js, micro-service architectures and possibly multiple versions of APIs all at the same time. The diversity of application environments makes it challenging for all RASP vendors to provide full support.

If your application doesn't run on a standard platform you will need to discuss support in great detail with the vendors prior to purchase. Within the next year or two we expect this issue to largely go away, but for now it is a key decision factor for buyers.

Versioning and Virtual Patching

A big problem for many organizations is keeping development, QA, and production versions of underlying software in synch and up to date. This creates two fundamental issues; differing versions between environments creates unstable applications, and teams are reluctant to patch for fear of introducing new issues. Continuous deployment pipelines and DevOps are process oriented approaches to address these problems, but some of the RASP platforms help as well. First by detecting which versions of application libraries are out of date, and second by offering 'virtual' patches by blocking attacks against those supporting services. As most development and operations teams do not track the never ending stream of security patches, having a tool automate discovery and reporting is helpful. Second, in cases where patching a defect is not possible, blocking threats specific to known vulnerabilities provides a workaround and speeds deployment.

Performance and Scalability

RASP embeds within the application or the supporting application stack. In this way RASP should scale with the application, in the same manner as the application. For example, if the scalability model means more copies of the application running on more server instances, RASP — being embedded in the application — will be run atop more server instances as well. If deployed on virtual or cloud servers, RASP benefits from added CPU and memory resources along with the application.

From a latency perspective, RASP enforcement rules — both how they operate and the number of checks employed — needs to be considered. The more analysis applied to incoming requests grants better security, but comes at a cost of latency. If third party threat intelligence is not cached locally, or or external lookups are used, latency increases. If the sensors or integration points are purely event collection, and the events are passed to another external server for analysis, you balance added services with increased latency. As we recommend with all security products, don't trust vendor supplied numbers, rather run tests in your environment with traffic that represents real application usage.

Instrumentation

One huge advantage of RASP is it can instrument application usage. Part of this capabilities is what we mentioned above: RASP can catalog application functions, understand the correct number and type of parameters, and then apply policies within the code of the running application. But it also can understand runtime code paths, server interaction, nuances of frameworks, library usage, and custom code. This offers advantages in the ability to tailor detection rules, such as employing specific policies to detect attacks against the Spring framework. It can be set to block specific attacks against older versions of libraries, providing a form of virtual patching. This also offers non-security related

benefits to developers, quality assurance and operations teams to show how code is used, providing a runtime map which shows things like performance bottlenecks and unused code.

Management and Maintenance

RASP deployments are typically managed via a central application, offering a single point for setting policies and directing the behavior of each application instance. We found that most users leverage a web console to set policies for production servers, but API orchestration is available as well. Client side libraries and/or agents must be installed for each application, and registered with the administrative console. Updated rules are — in most cases — immediately pushed out to the enforcement point and protection is adjusted dynamically without requiring a restart. You'll need to check with the vendor on specifics around how agents are updated, and how new rules are made available when new threats are discovered.

RASP also provides reporting for detected attacks, policy settings, and — in some cases — the types of vulnerabilities in your application. These reports are of interest to development, security and compliance teams trying to get a handle on risk and application security effectiveness. You'll want to experiment with the policy and reporting interfaces to ensure they work for your needs, or if not, can be customized.

Training RASP

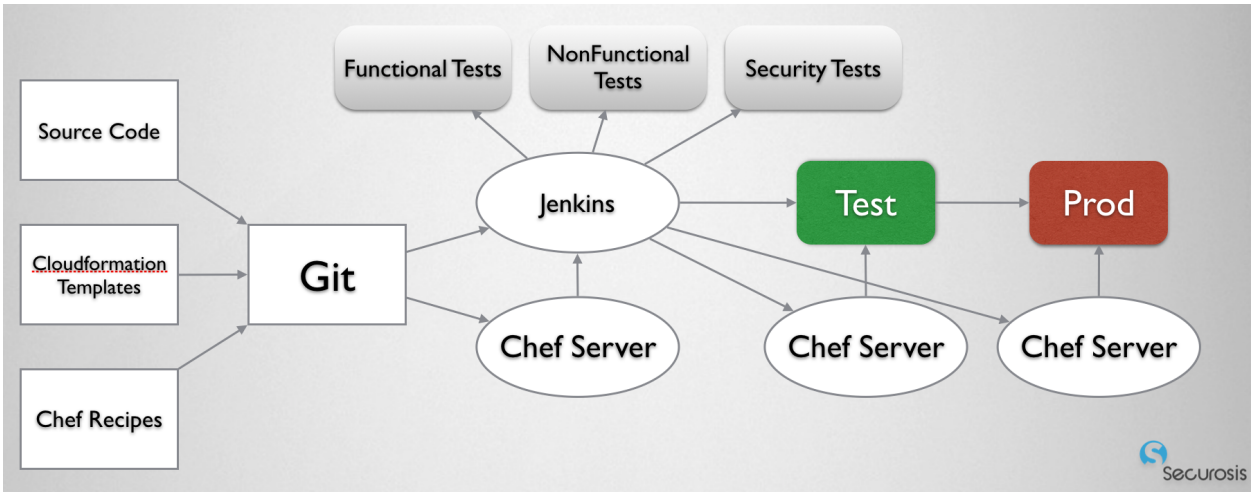
Some RASP platforms learn about the application they are protecting. In some cases this helps to refine detection rules, adapting generic rules to match specific application requests. Some are passive, learning from inbound requests, some actively crawl the application to discover all functions. As the RASP learns to 'understand' application state or recognize an appropriate *set of steps* within the application or appropriate *number of inputs* in a request. In most cases this is used to simply refine blocking capabilities for specific types of attacks, and happens without user interaction. For those that crawl application functions, determine how long this type of activity takes, and how this effects security in production and pre-deployment testing.

Integration

In this section we outline how RASP fits both into the technology stack, and development processes used to deliver applications. We will close this section with a discussion of how RASP differs from other security technologies, and discuss advantages and tradeoffs between differing approaches.

As we mentioned in the introduction, our research into DevOps unearthed many questions on RASP. The questions came from non-traditional buyers of security products; application developers and product managers. Their teams, by and large, were running Agile development processes. And they wanted to know if RASP could effectively block attacks and fit within their process.

The majority were leveraging automation to provide Continuous Integration — essentially automated building and testing of applications as new code was checked in. Some had gone as far as Continuous Deployment (CD) and DevOps. To address this development-centric perspective, we offer the diagram below to illustrate a modern Continuous Deployment / DevOps application build environment. Consider each arrow a script automating some portion of source code control, building, packaging, testing, or deployment of an application.



Security tools that fit this model are actively being sought by development teams. They need granular API access to functions, quick production of test results, and delivery of status back to supporting services.

Build Integration

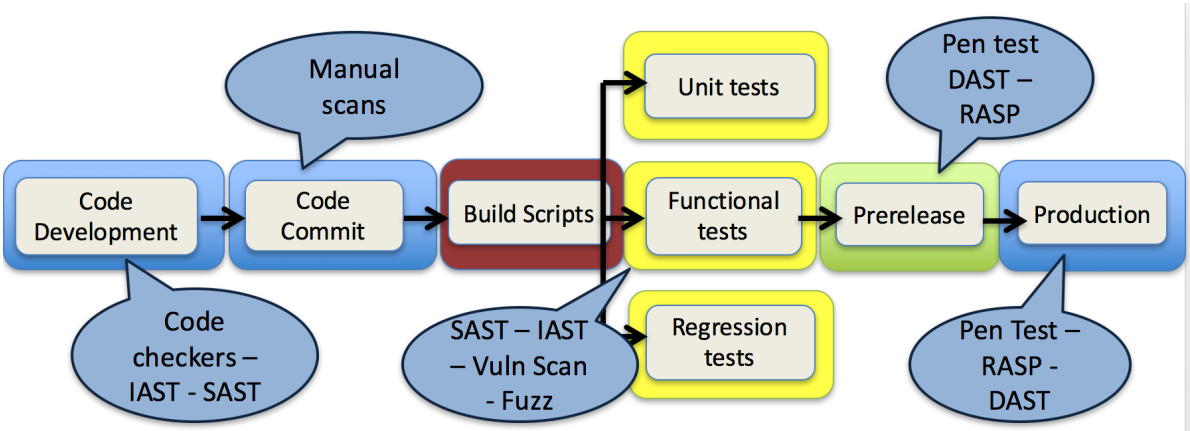
- **Installation**: As we mentioned back in the technology overview section, RASP products differ in how they embed within applications. They all offer APIs to enable script-based configuration and enablement of policies, but how and

where they fit in differ slightly between products. Servlet filters, plugins, and library replacement are embedded as the application stack is assembled. Configuration occurs as the application is launched. These approaches augment an application 'stack' to perform detection and blocking as if the application had been patched. Virtualization and JVM replacement approaches augment run-time environments, modifying the subsystems that run your application modified to handle monitoring and detection. In all cases these, be it on-premise or as a cloud service, the process of installing RASP is pretty much identical to the build or deployment sequence you currently use.

- **Runtime Rules & Policies**: We found the majority of RASP offerings include canned rules to detect or block most known attacks. Typically we'd talk about a *blacklist* of attack profiles which map closely to the OWASP Top Ten application vulnerability classes, but in the case of RASP, talk of white or black lists grossly underserves RASP capabilities. Protection against common variants of standard attacks, such as SQL injection and session mis-management, is of course included. But the benefits of monitoring all application behavior means subjective misuse like account hijacking is also possible. Once RASP is installed rules are immediately enforced. You can enable or disable individual rules as you see fit. Some vendors offer specific packages for critical attacks, mapped to specific CVEs such as Heartbleed. Bundles for specific threats, rather than by generic attack classes, help security and risk teams demonstrate policy compliance, and make it easier to understand which threats have been addressed. But when shopping for RASP technologies you need to evaluate the provided rules and custom rule management carefully. You will want to verify that you can augment or add rules as you see fit — rule management is a challenge for most security products, and RASP is no different.

- **Coverage capabilities**: During our research we found uneven RASP coverage of common platforms. Some started with Java or .Net, and are iterating to cover Python, Ruby, Node.js, and others. Your search for RASP technologies may be strongly influenced by available platform support. We find that more and more, applications are built as collections of micro-services across distributed architectures. Application developers mix and match languages, choosing what works best in different scenarios. If your application is built on Java you'll have no trouble finding RASP technology to meet your needs. But for mixed environments you will need to carefully evaluate each product's platform coverage.

Development Process Integration



Software development teams leverage many different tools to promote security within their overarching application development and delivery processes. The graphic below illustrates the major phases teams go through. The callouts

map the common types of security tests at specific phases within an Agile, CI, and DevOps frameworks. Keep in mind that it is still early days for automated deployment and DevOps. Many security tools were built before rapid and automated deployment existed or were well known. Older products are typically too slow, some cannot focus their tests on new code, and others do not offer API support. So orchestration of security tools — basically what works where — is far from settled territory. The time each type of test takes to run, and the type of result it returns, drives where it fits best into the phases above.

RASP is designed to be bundled into applications, so it is part of the application delivery process. RASP offers two distinct approaches to help tackle application security. The first is in the pre-release or pre-deployment phase, while the second is in production. Either way, deployment looks very similar. But usage can vary considerably depending on which is chosen.

- **Pre-release testing**: This is exactly what it sounds like: RASP is used when the application is fully constructed and going through final tests prior to being launched. Here RASP can be deployed in several ways. It can be deployed to *monitor only,* using application tests and instrumenting runtime behavior to learn how to protect the application. Alternatively RASP can monitor while security tests are invoked in an attempt to break the application, with RASP performing security analysis and transmitting its results. Development and Testing teams can learn whether RASP detected the tested attacks. Finally, RASP can be deployed in full blocking mode to see whether security tests were detected and blocked, and how they impacted the user experience. This provides an opportunity to change application code or augment the RASP rules *before* the application goes into production.

- **Production testing**: Once an application is placed in a production environment, either before actual customers are using it (using Blue-Green deployment) or afterwards, RASP can be configured to block malicious application requests. Regardless of how the RASP tool works (whether via embedded runtime libraries, servlet filters, in-memory execution monitoring, application instrumentation or virtualized code paths), it protects applications by detecting attacks in *live* runtime behavior. This model essentially provides execution path scanning, monitoring all user requests and parameters. Unlike technologies which block requests at the network or web proxy layer, RASP inspects requests at the *application* layer, which means it has full access to the application's inner workings. Working at the API layer provides better visibility to determine whether a request is malicious, and more focused blocking capabilities than external security products.

- **Runtime protection**: Ultimately RASP is not just for testing, but for full runtime protection and blocking of attacks. Not just the typical cross-site scripting (CSS), SQL-Injection (SQLi), cross-site request forgeries (XSRF) or other drive by attacks, but malicious code execution, weak authentication, improper session management, use of vulnerable 3rd party software, and misuse of custom code. And it understands attacks specific to the platforms (*e.g.*: .NET, Java) and application frameworks (e.g.: Spring, Struts, Play). RASP is in a unique position to protect application from a broad set of application attacks. Again, monitoring and protecting at the application layer provides subtle context and behavioral clues that provide substantial improvement in detection with low to no false positives.

To WAF or not to WAF

Why did the market develop this brand-new security technology? Especially when existing technologies — most notably Web Application Firewalls (WAF) — already provided similar functions. Both block attacks on web-facing applications. They are both focused on known attack vectors, but WAF is focused primarily on *blacklists* of attack patterns, with some closing gaps with optional *whitelisting* of known (approved) application functions. RASP is fundamentally different in that it understand the application, code paths, structure and application composition. All of this means more accurate detection, and less reliance on black lists, heuristics and other approaches that are known to have inherent weaknesses

in detection as well as generation of false positive. So why spend time and money on a new tool? Because, at it's core, RASP is more effective at attack detection.

Second, WAF's have a history of being difficult to maintain. WAF management teams that WAFs are fine for basic filtering of common web borne attacks, but to be really effective, you need to enable 'positive' security rules, or 'white listing' of application requests. And therein lies the problem, as these policies must be updated as the application is updated. Get it wrong and you break the application. This constant need to re-learn and update was a big part of dissatisfaction with ongoing maintenance. Most RASP variants are effective immediately upon installation, and can be updated without application restarts.

Most organizations we spoke with stated their basic requirement was for something to work within their development pipeline. WAF's lack of APIs for automatic setup, the time needed to learn application behavior, and most importantly the ability to pinpoint vulnerable code modules, were all cited as reasons WAF failed to satisfy developers. Granted, these requests came from more 'Agile' teams, more often building new applications than maintaining existing platforms. Still, we heard consistently that RASP meets a market demand unsatisfied by other application security technologies.

It is important to recognize that these technologies can be complementary, not necessarily competitive. There is absolutely no reason you can't run RASP alongside your existing WAF. Some organizations continue to use cloud-based WAF as front-line protection, while embedding RASP into applications. Some use WAF to provide "threat intelligence", DoS protection, and network security, while using RASP to fine-tune application security — often supplanting WAF's 'white listing' capability. Still others double down with overlapping security functions, much the way many organizations use layered anti-spam filters, accepting redundancy for broader coverage or unique benefits from each product. WAF platforms have a good ten-year head start, with broader coverage and very mature platforms, so some firms are loath to throw away WAF until RASP is fully proven, or until RASP is viewed as an acceptable compensating control for regulations like PCI-DSS.

Buyers Guide

There are several key areas for buyers to evaluate in order to match a RASP platform with your needs. With new technologies it is not always clear where the 'gotchas' are. We find many security technologies meet basic security goals, but after they have been on-premise for some time, you discover management or scalability nightmares. To help you avoid some of these pitfalls, we offer the following outline of evaluation criteria. The product you choose should provide application protection, but it should also be flexible enough to work in your environment. And not just during Proof of Concept (PoC) – but every day.

- **Language Coverage**: Your evaluation should ensure that the RASP platforms you are considering all cover the programming languages and platforms you use. Most enterprises we speak with develop applications on multiple platforms, so ensure that there is appropriate coverage for all your applications – not just the ones you focus on during the evaluation process.

- **Protection**: Protection applications from a broad range of attacks the core value RASP provides. . Sure, some of you will use RASP for monitoring and instrumentation – at least in the short term – but protecting applications is at the heart of RASP's value. Evaluating how well a RASP product blocks is essential. The goal here is twofold: make sure the RASP platform is detecting the attacks, and then determine if its blocking action negatively affects the application. We recommend penetration testing during the PoC, both to verify that common attack vectors are handled, and to gauge RASP behavior when attacks are discovered. Some RASPs simply block the request and return an error message to the user. In most cases RASP throws an application error, and allows the application to handle the process from there. At least one product altered user sessions and redirect users to login again, or jump through additional hoops before proceeding. Most vendors consider attack detection techniques part of their "secret sauce", so we are unable to detail all of the nuances here. It's important attacks are detected with a low rate of false positives, but it's equally important to protect applications without disrupting application function or requiring massive code re-writes to enable.

- **Performance**: Being embedded into applications enables RASP to detect threats at different locations within your app, with context around the operation being performed. Does the RASP detect and enforce in place, or does it pass along the request to a central enforcement point for analysis? How long does it take to analyze different attack vectors and what is the average latency? The details behind how detection works vary between vendors, but there is a crucial balancing act going on between effective detection and low latency. Each user request may generate dozens of checks, possibly including external references. This latency can easily impact user experience, so sample how long analysis takes *for different classes of threats*! Each code path will apply a different set of rules, so you will need to test several different paths, measuring both with and without RASP. You should do this under load to ensure that detection facilities do not bottleneck application performance. And you'll want to understand what happens when some portion of RASP fails, and how it responds – does it "fail open"?

- **Policy Coverage**: It's not uncommon for one or more members of a development team to be proficient with application security. That said, it's unreasonable to expect developers to understand the nuances of new attacks and the details behind every CVE. Vulnerability research, methods of detection, and appropriate methods to block attacks are large parts of the value each RASP vendor provides. Your vendor spends days – if not weeks – developing each

policy embedded into their tool. During evaluation, it's important to ensure that critical vulnerabilities are addressed. But it is arguably more important to determine how – and how often – RASP vendors update policies, and verify they include ongoing coverage.

- **Policy Management**: Two facets of policy management come up most often during our discussions. The first is identification of which protections map to specific threats. Security, risk, and compliance teams all ask, "Are we protected against XYZ threat?" You will need to show that you are. Evaluate policy lookup and reporting. The other is tuning how to respond to threats. As we mentioned above under 'Blocking', most vendors allow you to tune responses either by groups of issues, or on a threat-by-threat basis. Evaluate how easy this is to use, and whether you have sufficient options to tailor responses.

- **Scalability**: Most web applications scale by leveraging multiple application instances, distributing user requests distributed via a load balancer. As RASP is typically built into the application, it scales right along with it, without need for additional changes. But if RASP leverages external threat intelligence, you will want to verify this does not hamper scalability. For RASP platforms where the point of analysis – as opposed to the point of interception – is outside your application, you need to verify how the analysis component scales. For RASP products that work as a cloud service using non-deterministic code inspection, evaluate how their services scale.

- **API Compatibility**: Most interest in RASP is prompted by a desire to integrate into application development processes, automating security deployment alongside application code, so APIs are a central feature. Ensure the RASP products you consider are compatible with Jenkins, Ansible, Chef, Puppet, or whatever automated build tools you employ. On the back end make sure RASP feeds information back into your systems for defect tracking, logging, and Security Information and Event Management (SIEM). This data is typically available in JSON, syslog, and other formats, but ensure each product provides what you need.

- **Stability**: Most RASP technologies are fully effective once installed, but it should be noted at least on of the products we reviewed required some time to learn the application. This delays deployment and you should consider if this will be a problem for your use cases. Also, we noted that most products can update rules and policies into the RASP engine, at runtime, without requiring the application to be restarted. This means that security can be more agile than the application it secures. This is a desirable trait to consider when selecting a platform.

Regardless of where you deploy RASP, you need to test to ensure it is delivering on its promise. We advocate an ongoing testing process to ensure your policies are sound, and that you ultimately block what you need to block. Of course you can use other scanners to probe an application to ensure RASP is working prior to deployment, and other tools (such as Havij and SQLmap) to automate testing, but that's only half the story. For full confidence that your apps are protected, we still recommend actual humans banging away at your applications. Penetration testing, at least periodically, helps verify your defenses are effective.

About the Author

Adrian Lane, Analyst and CTO

Adrian Lane is a Senior Security Strategist with over 25 years of industry experience. He brings over a decade of C-level executive expertise to the Securosis team. Mr. Lane specializes in database architecture and data security. With extensive experience as a member of the vendor community (including positions at Ingres and Oracle), in addition to time as an IT customer in the CIO role, Adrian brings a business-oriented perspective to security implementations. Prior to joining Securosis, Adrian was CTO at database security firm IPLocks, Vice President of Engineering at Touchpoint, and CTO of the secure payment and digital rights management firm Transactor/Brodia. Adrian also blogs for Dark Reading and is a regular contributor to Information Security Magazine. Mr. Lane is a Computer Science graduate of the University of California at Berkeley with post-graduate work in operating systems at Stanford University.

About Securosis

Securosis, L.L.C. is an independent research and analysis firm dedicated to thought leadership, objectivity, and transparency. Our analysts have all held executive level positions and are dedicated to providing high-value, pragmatic advisory services.

Our services include:

- The Securosis Nexus: The Nexus is an online environment to help you get your job done better and faster. It provides pragmatic research on security topics, telling you exactly what you need to know, backed with industry-leading expert advice to answer your questions. The Nexus was designed to be fast and easy to use, and to get you the information you need as quickly as possible. Access it at <https://nexus.securosis.com/>.

- Primary research publishing: We currently release the vast majority of our research for free through our blog, and archive it in our Research Library. Most of these research documents can be sponsored for distribution on an annual basis. All published materials and presentations meet our strict objectivity requirements and conform with our Totally Transparent Research policy.

- Research products and strategic advisory services for end users: Securosis will be introducing a line of research products and inquiry-based subscription services designed to assist end user organizations in accelerating project and program success. Additional advisory projects are also available, including product selection assistance, technology and architecture strategy, education, security management evaluations, and risk assessment.

- Retainer services for vendors: Although we will accept briefings from anyone, some vendors opt for a tighter, ongoing relationship. We offer a number of flexible retainer packages. Services available as part of a retainer package include market and product analysis and strategy, technology guidance, product evaluation, and merger and acquisition assessment. We maintain our strict objectivity and confidentiality. More information on our retainer services (PDF) is available.

- External speaking and editorial: Securosis analysts frequently speak at industry events, give online presentations, and write and/or speak for a variety of publications and media.

- Other expert services: Securosis analysts are available for other services as well, including Strategic Advisory Days, Strategy Consulting Engagements, and Investor Services. These tend to be customized to meet a client's particular requirements.

Our clients range from stealth startups to some of the best known technology vendors and end users. Clients include large financial institutions, institutional investors, mid-sized enterprises, and major security vendors.

Additionally, Securosis partners with security testing labs to provide unique product evaluations that combine in-depth technical analysis with high-level product, architecture, and market analysis. For more information about Securosis, visit our website: <http://securosis.com/>.