waratek

Discussion Brief:

Runtime Application Self Protection (RASP) Evaluation Criteria

March 2016

RSAC Innovation Sandbox 2015 WINNER

Table of Contents

Introduction

RASP, or "Runtime Application Self Protection" is the evolution of two precursor technologies: Web Application Firewalls (WAF), and Static Application Security Testing (SAST), and related technologies.

WAF solutions, also known as Layer 7 firewalls, were the first industrial-scale solutions for application security. Operating at the highest network protocol layer (layer 7), WAFs attempt to increase the sophistication and accuracy of traditional "packet-filter" firewalls. The inventive rationale that gave birth to WAFs was simple: if the firewall can understand the application's protocol and parameter semantics, then more accurate exploit detection will result. For simple applications, WAF's thesis worked with some benefit. But as application complexity grew – accompanied by the rapid emergence of new application types and technologies such as JSON, REST, etc. – the ability of WAFs to provide accurate exploit detection without false positives and endless human-tuning has not materialized. Today it is rare to find a WAF deployed in unconditional blocking mode for any application, and this is testament to the inherent inaccuracy of WAF technologies.

SAST (and related) technologies grew out of a different community and direction. Popularized by source-code scanning products from Fortify and others, SAST (and later variations DAST and IAST) attempted to identify application security vulnerabilities via analysis of application source-code or application input validation. Unlike WAF solutions, which focus on "*exploit detection*", SAST (and related) technologies focus on "*vulnerability detection*" – that is, the identification of potential attack vectors which may or may not be exploitable. The difference is important: while SAST operates closer to the "application logic" (as it is has access to the source/binary-code), it cannot identify *exploits*, only *potential vulnerabilities*. The keyword here is "potential", as all SAST (and related) testing solutions produce substantial false-positive detections as, just like WAFs, they lack complete application context. The result is that SAST and related testing solutions have at least as high false-positive (inaccurate) detection rates as WAF technologies, which like WAFs, drives up the workload for, and reliance on, expensive human experts to sift through false positive detections.

RASP is the convergence of the WAF and SAST technology categories. Combining the intelligence of an application's code (like SAST) and the intelligence of an application's network traffic flows (like WAF), together with the real-time intelligence of an application's executing state, RASP has complete intelligence of everything that goes on inside the application (i.e. 100% intelligence of application execution).

Gartner defines this "100% intelligence" threshold as the distinguishing feature of RASP:

"[RASP]…sees all data coming in and out of the application, all events effecting the application, all executed instructions, and all database access".[1]

The consequence of this "100% intelligence" threshold is what makes RASP transformational: as RASP sees 100% of everything an application does and how it does it, RASP does not need any extra intelligence in order to achieve zero false positive (i.e. accurate) detection. As a result, RASP is the beginning of the 'accurate security' era.

[1] Gartner - Runtime Application Self-Protection: A Must-Have, Emerging Security Technology. 24 April 2012

RASP Approaches

As awareness of the category increases, a number of vendors have released new products that claim to be RASP implementations. Loosely speaking, there are three current approaches used by RASP technologies:

1. Servlet Filters
2. Instrumentation
3. Virtualization

Servlet Filters

A Servlet filter is an object that can intercept HTTP requests targeted at your web application. A servlet filter can intercept requests both for servlets, JSP's, HTML files or other static content. [2]

Filters are only really relevant for the top layer of the application stack and from a security perspective, they have very limited use. In practice, some vendors simply transplant the heuristic techniques of a WAF into a servlet filter with all of the associated inaccuracy problems.

Instrumentation

Some RASP implementations are based on instrumentation techniques.

The Java instrumentation API is not designed for controlling application execution. For example, the use of the instrumentation API in production applications to date has been exclusively in Application Performance Monitoring/Management (APM) tools like AppDynamics, NewRelic etc.

The instrumentation API is principally an *informational* API not a *control* API. Even with this limitation, the instrumentation API is infeasible to be used to instrument all Java API events and so all APM tools and all RASP vendors who use this approach instrument only a modest number of call points. Measurements of one Instrumentation-based RASP product showed that less than 0.1% of loaded methods were instrumented, unsurprisingly resulting in low accuracy rates and incomplete protection.

[2] http://tutorials.jenkov.com/java-servlets/servlet-filters.html

Virtualization

Waratek AppSecurity for Java is the only RASP solution that is implemented using virtualization. A more detailed explanation of the Waratek architecture can be found in the section "How Waratek Works" on page 7.

Challenges of Non-Virtualized Implementations

The single biggest challenge of filter or instrumentation based approaches to RASP is their inability to protect themselves. This is due to the limitations and inherent insecurity of an architecture that does not provide any true separation between the protection filter/agent and the application that is being protected.

In operating system terms, this is similar to running both a protection agent and a vulnerable application in user space, rather than implementing some form of Kernel/Userspace separation.

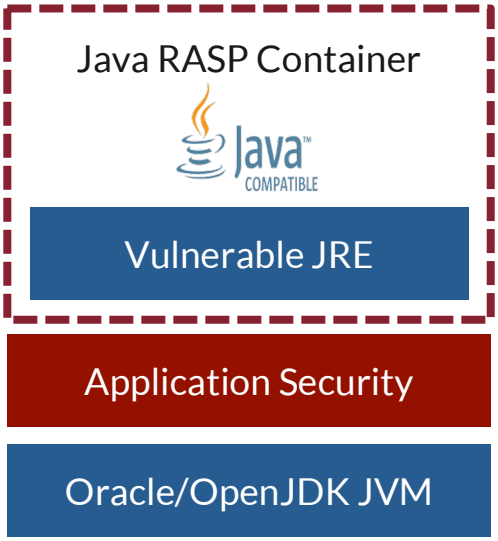
In the event of an exploit, filter or agent based RASP implementations can be directly attacked either by terminating its functionality or by accessing its memory or security policy configuration.

If a RASP solution cannot protect itself from attack, how can such a solution protect a known-vulnerable third-party application? RASP solutions which cannot protect themselves are not ***Self Protecting***, and do not belong to the "RASP" category at all. Unfortunately, the industry excitement that RASP has quickly generated in the application security community has coincided with several vendors releasing dubious "RASP solutions" which fail to meet the most basic RASP criteria.

Accuracy, performance and the ability to self-protect are examined in more detail in the "RASP Solutions must…" section below.

How Waratek Works

- Waratek provides *Runtime Application Self-Protection* technology for Java applications built on top of the Oracle JVM
- A Java Container is a protected in-JVM container with built in application security and quarantine controls
- The Java container separates apart the vulnerable JRE code (where the insecure Java APIs reside) from the low-level JVM (the JIT compiler and GC)
- Application security controls inserted between the Java Container and the JVM protect and quarantine the Java application



- Technically, the Waratek software is implemented as a 'pure-Java' in-JVM hypervisor or virtual machine monitor. However, rather than emulating hardware, the Waratek hypervisor emulates the entire Java specification
- Every Java container is a completely isolated abstraction of a "real" JVM and encapsulates *all* layers of the application stack – the user application, development frameworks, third party libraries, applications servers and even the JRE
- Each Java container has its own classpath, classloaders, namespace, environment variables, heap allocation and resource quotas
- Every container (guest) has its own JRE version that can be different to the host JVM version

A java application running on a conventional Oracle HotSpot/OpenJDK JVM can make use of a large set of Java APIs (the JRE class library) consisting of nearly 20,000 classes. Any of those classes can interact with the underlying operating system and JVM through native calls (JNI).

In contrast, an application running inside a Waratek RASP Container runs on a virtualized abstraction of the JVM and OS, so 20,000 guest classes virtualize onto only several-hundred classes in the host environment – a circa 98% reduction in attack-surface and risk profile for every Java application.

Some of the consequences of this architecture are as follows:

- Waratek Containers isolate the application's JRE (the untrusted Java APIs where vast majority of vulnerabilities reside) from the JVM. Running the application's JRE inside a RASP Container reduces the JVM/OS attack surface by between 95-99%.
- The guest software inside the JVC runs with its own JRE, and this guest JRE is not used by the Waratek software. So attacks, manipulations or vulnerabilities in the guest JRE cannot be used by the guest software to effect Waratek software in any way.
- The guest RASP Container (including the guest JRE) has a wholly independent namespace from the Waratek software on the host JVM, so guest software cannot access host types or classes.
- All reflection APIs are completely emulated by Waratek software, so guest software never has access to real reflection APIs.
- Waratek never allows foreign (non-Waratek) software to be loaded in host.

When evaluating the security of traditional virtualization technologies based on hardware virtualization (e.g. VMWare) or on operating system virtualization (e.g. LXC/Docker), many of the principal security concerns are a consequence of a multi-tenant architecture. If an attacker were to breach the containment provided by these technologies and take control of the hypervisor or LXC host, then they would have complete access to neighboring tenants that were running along side the compromised container.

These concerns do not apply to Waratek RASP Containers. Although the virtualization provided by Waratek's RASP Containers is so complete as to be technically able to run multiple Waratek RASP Containers inside a single JVM, this is not a supported deployment model.

Essential Criteria

Like precursor application security technologies, all RASP solutions must provide a basic set of application security features. These include the following:

- **Thwarting evasion techniques**
  Attackers have increasingly employed sophisticated obfuscation techniques to evade detection by application security technologies. RASP solutions must be able to detect and de-obfuscate inputs provided by attackers while protecting the application without false positive detections.
- **Protecting the major SANS CWE vulnerabilities**
  While the SANS list of CWE vulnerabilities numbers over 1000, 80% of application security vulnerabilities by volume are comprised of the SANS Top 25[3] CWE vulnerabilities. RASP solutions should protect against as many of these as possible, while focusing on the high priority web-application vulnerabilities such as SQLi, XSS, and CSRF.
- **Supporting on premise and cloud deployment**
  The last several years has seen the computing landscape change with increasing use of public cloud computing facilities. RASP solutions must be capable of both on-premise and off-premise configurations so that organizations may use RASP across the full range of deployment models that their applications require.
- **Automating and scaling operations**
  To comprehensively address application security requirements in a large organization, an application security technology must be deployable and operable at the scale of thousands of applications. RASP solutions in this regard are not unique, and all RASP solutions must support efficient deployment and operation for thousands of applications with automated and scriptable configuration and maintenance.

[3] https://www.sans.org/top25-software-errors/

“Transformational” Criteria

Beyond the basic feature-set that all application security technologies should provide, RASP solutions are strategic and so must be evaluated more stringently than traditional application security technologies. RASP solutions must in particular be capable of providing the ‘transformational’ security features that distinguish RASP from precursor technologies. These are discussed below.

RASP Solutions must…

1. Never generate a false positive

Challenge: All precursors technologies to RASP including WAF, IPS, and SAST/DAST generate significant number of false positive detections. A 2015 report from the Ponemon Institute measuring the cost of inaccurate security solutions revealed that organizations spend 21,000 hours per year – 400 hours per week – dealing with false-positive (inaccurate) security detections. [4]

For any security technology that does not guarantee zero false positive detection, two inevitabilities follow: (i) that security technology will only be deployable in monitoring/logging mode because of the risk of blocking legitimate application operation, and (ii) the “real” work of protecting an application will end up being performed by expensive teams of human beings.

Requirement: Of all 31 application security technologies tracked by Gartner, only RASP is rated as “transformational” [5]. Why? One compelling reason: true RASP technology never generates a false positive detection as it can access 100% of application execution intelligence. Gartner defines this complete intelligence capability as the distinguishing feature of RASP. At the root of RASP is a capability called “non-heuristic injection detection”, something that has been impossible for WAF and SAST technologies that rely on regular expressions (regex) and pattern matching. As a result, when evaluating a RASP solution pay very careful attention for any use of regex, pattern matching, or other heuristic techniques to detect exploits, as their use is a giveaway of recycled WAF technology.

[4] http://www.reuters.com/article/ga-damballa-idUSnBw165014a+100+BSW20150116
[5] Gartner - Hype Cycle for Application Security. 09 July 2015

2. Protect every layer of the application stack

Challenge: In the last several years, IT security teams have begun assuming growing application security responsibility, and with this has come a fast awakening that "application security" means more than just "business logic security". Precursor technologies to RASP, which could only observe application events via network traffic flows, necessarily focused on business-logic vulnerabilities, especially injection vulnerabilities like SQLi and XSS. Real application security means protecting every layer of the application stack – from the lowest-level JRE APIs, up through the various application server and framework layers, on through the 3rd party class libraries and open-source components, and finally the business logic layer itself.

Requirement: While most application security professionals will be aware of the high-profile security scares associated with names such as "Apache Struts" [6], "Apache Commons" [7] and a host of others, not all security professionals might be aware that the JRE APIs themselves – the same APIs upon which all of the world's Java code depends – have a new security vulnerability identified and patched every 100 hours on average. Unfortunately, that uncomfortable statistic is not all that different for any other major software component be it application servers, application frameworks, and of-course open-source libraries. As a result, RASP solutions must provide virtual patching capability for *all* layers of the software stack, including the Java JRE APIs themselves. For patching and compliance requirements, RASP solutions must provide "impact reduction" rule sets for eradicating zero-day CVSS 7.0 and above vulnerabilities, and "attack surface reduction" rule sets for eliminating up to 80% of all vulnerabilities.

3. Protect legacy platforms

Challenge: Large organizations with hundreds or thousands of applications are invariably operating a significant portion of older "legacy" applications. A unique risk of legacy applications is their dependence on old and unsupported Platform APIs (such as the Java JRE APIs) that are often end-of-life (e.g. out-of-support) and affected by a long list of widely published vulnerabilities. The extent of this problem is significant: in the last several years 99.9% of publicized breaches resulted from vulnerabilities known for one year or longer, and of the major database breaches in that time, 0% of the exploited databases had applied all of the recommended critical patch updates. In the context of RASP where the RASP software must operate "inside" the application, a RASP solution must use novel software methods to avoid being exposed to the same legacy Platform API vulnerabilities as the legacy application being protected.

Requirement: Today's shiny new application is tomorrow's legacy application, so when considering a RASP solution, it is necessary to evaluate how that solution will protect legacy

[6] https://securityintelligence.com/struts-vulnerabilities-analysis-parameters-cookie-interceptors-impact-exploitation/
[7] http://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/

platform software. For example, an old Java business application may be deployed on an old and unsupported Java Platform version, such as Java SE 5 or Java SE 6. Often, a legacy application may have a dependency on a specific legacy API version – such as Java SE 6 update 21 (Java 1.6.0_21)– and attempting to upgrade the legacy APIs to the latest version may break the application. A RASP solution must be completely isolated from the Platform API of the application being protected, otherwise the RASP software will be vulnerable to all of the weaknesses and exploits in the legacy Platform APIs. What this means in practice is that RASP solutions need to employ application virtualization techniques so as to quarantine and isolate vulnerable legacy Platform APIs, and not just vulnerable legacy business logic. When evaluating a RASP solution, pay careful attention that the RASP software uses separate Platform APIs from the application being protected, and that the Platform APIs being used by the RASP software can be independently updated or patched without touching the legacy application.

4. Protect itself

Challenge: RASP's unique place in the application stack – literally inside the application – means that RASP itself faces a unique set of threats that precursor technologies never faced. Cohabiting the same application stack as the vulnerable application which RASP is meant to be protecting, means that a poorly designed RASP solution will be susceptible to a wide range of attacks against itself, not just the vulnerable application. The severity of this risk cannot be overstated: if an application is vulnerable to any remote code injection exploit which the RASP software does not fully protect against, then that remote code injection exploit can be used to instantly deactivate the RASP protections for an application, opening the doors to a full raft of application and data attacks. Furthermore, outsider attacks are not the only threat to a non-isolated RASP solution. Insider attack from maliciously written application code, or compromised application artifacts introduced by a manipulated build system, can instantly deactivate an application's RASP protection if the RASP solution is not completely isolated from the vulnerable application.

Requirement: As RASP software cohabits the application stack of the vulnerable application it is seeking to protect, a RASP solution must be carefully designed to be inaccessible by any of the vulnerable application code. As a vulnerability can appear in any layer of the application stack – including the Java JRE APIs themselves – a RASP solution needs to use advanced software techniques such as application virtualization to isolate itself from the full software stack of untrusted code. One of the major "no-nos" that a RASP solution must avoid is sharing the vulnerable application's name-space or address-space. Failure to enforce fully isolated name-spaces/address-spaces for the RASP software, means the RASP solution itself can be instantly deactivated by direct application manipulation. When evaluating a RASP solution pay very careful attention to name-space and address-space isolation, validating this important requirement with thorough introspection testing.

5. Support live (no restart) rule updating

Challenge: Change in the IT industry is the daily norm, but in application security it is imperative. As new vulnerabilities are constantly identified in every layer of the application stack, it is necessary to respond by changing existing application security rules, or apply new application security rules, without requiring a target application to be restarted. Live rule updating, also known as 'virtual patching', has been provided by precursor technologies such as WAFs, for RASP solutions this can be considerably more complex to support as changing applied rules may require material changes to the executing state of an application stack.

Requirement: Live rule updating is a complex facility to implement in a RASP solution. A RASP solution operates by integrating itself into the executing state of an application, and by doing so, becomes part of the application environment in a similar way to an operating system. Changing RASP software behavior in tandem with rule updates will often require the RASP software to effect material changes to the executing state of an application, particularly in order to optimize performance. When evaluating a RASP solution make sure to confirm that when any given rule is applied, changed, or removed at runtime that the updated rule behavior is observed in a timely manner. In addition, be sure to confirm that any rule change of any rule category does not negatively impact application performance, either during the rule-change event, or after the new rule(s) are in force.

6. Provide a constant performance threshold that does not decrease as protection rules grow

Challenge: Performance, like beauty, is a subjective quality. What might be an acceptable performance overhead for one application might be intolerable for another. Therefore defining universal performance thresholds is not of value. Instead, what is important for application security controls is that their overhead must not grow as the number or complexity of protection rules grows.

Requirement: While defining universally acceptable performance thresholds is impossible, defining performance requirements for RASP is not. For a RASP solution to be practical, it must be able to operate with a near-constant performance overhead regardless of the number and complexity of protection rules in force. When evaluating a RASP solution, take care examining touted performance scores, making sure that scores generated are for all rules enabled in a "full protection" mode. In particular, take special care that any "sampling features" are disabled or not present: some RASP solutions are known to use performance tricks like partial sampling that turn off intelligence gathering as well as protection for some significant (often the majority) of an application's execution time, in order to increase measured performance. Performance tricks such as these are dangerous and should be avoided as they invariably introduce inaccurate (false positive) detection risk due to incomplete application intelligence, as well false (absent) protection for significant periods of time.