



SANS Institute

Information Security Reading Room

Runtime Application Self-Protection (RASP), Investigation of the Effectiveness of a RASP Solution in Protecting Known Vulnerable Target Applications

Alexander Fry

Copyright SANS Institute 2021. Author Retains Full Rights.

This paper is from the SANS Institute Reading Room site. Reposting is not permitted without express written permission.

Runtime Application Self-Protection (RASP), Investigation of the Effectiveness of a RASP Solution in Protecting Known Vulnerable Target Applications

GIAC (GWAPT) and RES 5500 Gold Certification

Author: Alexander J. Fry, alexanderfry@student.sans.edu

Advisor: Tanya Baccam

Accepted: April 15, 2019

Abstract

Year after year, attackers target application-level vulnerabilities. To address these vulnerabilities, application security teams have increasingly focused on shifting left - identifying and fixing vulnerabilities earlier in the software development life cycle. However, at the same time, development and operations teams have been accelerating the pace of software release, moving towards continuous delivery. As software is released more frequently, gaps remain in test coverage leading to the introduction of vulnerabilities in production. To prevent these vulnerabilities from being exploited, it is necessary that applications become self-defending. RASP is a means to quickly make both new and legacy applications self-defending. However, because most applications are custom-coded and therefore unique, RASP is not one-size-fits-all - it must be trialed to ensure that it meets performance and attack protection goals. In addition, RASP integrates with critical applications, whose stakeholders typically span the entire organization. To convince these varied stakeholders, it is necessary to both prove the benefits and show that RASP does not adversely affect application performance or stability. This paper helps organizations that may be evaluating a RASP solution by outlining activities that measure the effectiveness and performance of a RASP solution against a given application portfolio.

1. Introduction

Recent technological trends show a shift from traditional monolithic web applications to microservices written in Node.js and Spring Boot, single page web applications written in frameworks such as Angular and React, and JavaScript as the primary language of the Web, both on the client and the server which pushes data and business logic closer to the end user (OWASP, 2017). Applications continue to migrate from traditional data centers to the public cloud and take advantage of technologies such as containerization (e.g., Docker). In addition, development and operations teams are accelerating the pace of software release using continuous integration and/or continuous delivery. Continuous integration entails testing code changes whenever they are made which is typically when the code changes are submitted to the source code management system. Continuous delivery involves automating the deployment or release process, resulting in frequent changes to the application in production. The high rate of code change and fast pace of development have the potential to introduce software defects and security vulnerabilities that have not been adequately tested for in the code base.

While these technological trends may have the potential to improve security posture long-term, applications remain vulnerable. In fact, according to Verizon's Data Breach Investigations Report (2018), the majority of breaches were caused by web application attacks, making it the most common type of breach per pattern (Figure 1).

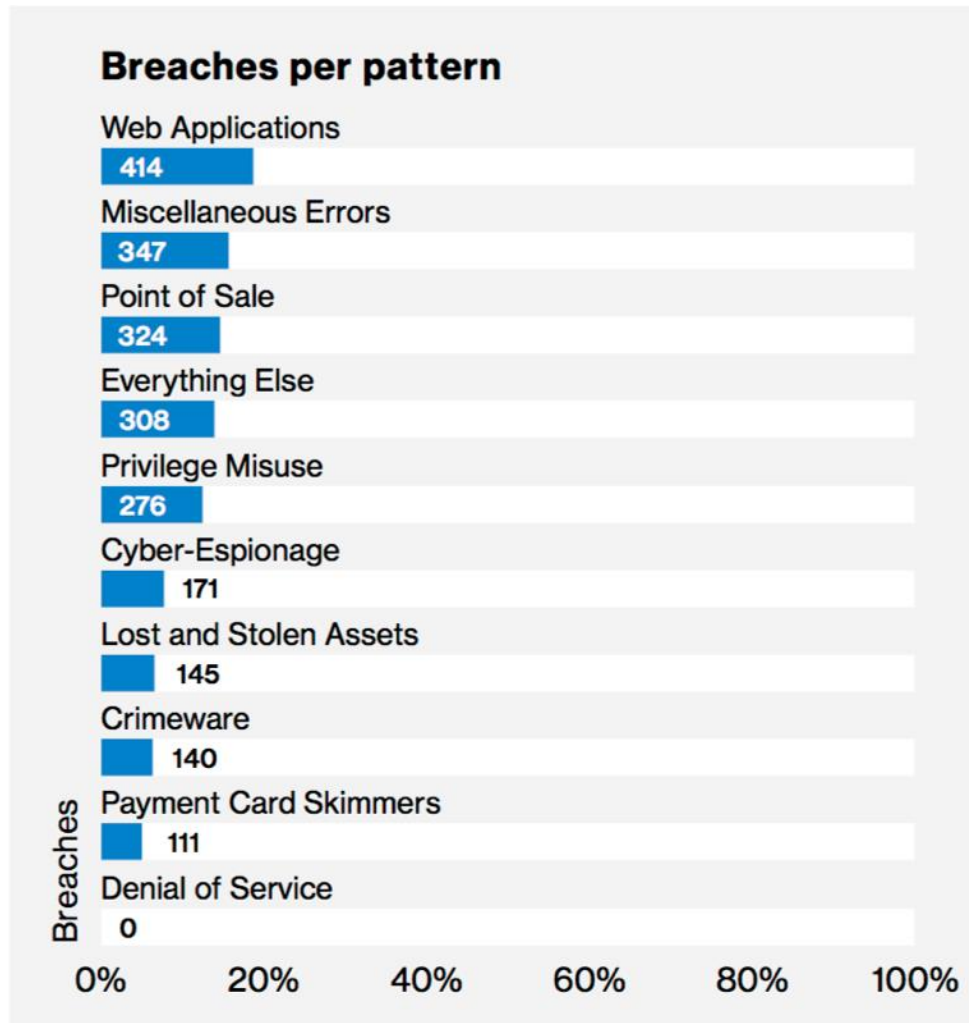


Figure 1. Percentage and count of breaches per pattern (n=2,216). From “2018 Verizon Data Breach Investigations Report,” 2018.

The reality of application attacks accounting for the majority of breaches necessitates better protection for production applications. Over the last couple of decades, network protection has moved closer and closer to the application – from the firewall to the intrusion prevention system to the web application firewall (WAF): “That evolution has involved looking deeper and deeper into HTTP, SOAP, XML, and other application-layer network protocols. The reason for this migration is simple: the better you understand applications, the more accurately you can detect and block application attacks” (Contrast, 2017). However, the latest advance in network protection, the WAF, lacks visibility into the running application.

The purpose of a WAF is to protect web applications and application programming interfaces (APIs) against a variety of attacks, including automated attacks (bots), injection attacks and application-layer denial of service (DoS). WAFs typically provide signature-based protection and support positive security models (automated whitelisting). Some WAFs also provide anomaly detection by first establishing a baseline of what constitutes normal application behavior. WAFs are deployed in front of web servers to protect web applications against external and internal attacks, to monitor and control access to web applications, and to collect access logs for compliance/auditing and analytics. Traditionally, WAFs were deployed as physical or virtual appliances, but WAFs are increasingly being delivered as a managed service and/or part of a public cloud offering such as Amazon Web Services (AWS) WAF (Gartner, Inc., 2019).

WAFs operate in front of the application and therefore lack the context needed to determine if a given input should be blocked. This need to approximate or guess the result of a given input results in a high degree of inaccuracy. This inaccuracy may lead to a given attack being successful. As Ullrich states: “Many web applications are directly exposed to external attacks and, while infrastructure systems such as web application firewalls exist, they are often considered inadequate for deterring a sophisticated attacker” (Ullrich, J., 2016).

Runtime Application Self-Protection (RASP) is the next step in the evolution. Gartner defines RASP as “a security technology that is built or linked into an application or application runtime environment and is capable of controlling application execution and detecting and preventing real-time attacks” (Gartner, 2012, November 4). RASP provides a level of visibility and accuracy that network security solutions cannot achieve by operating within the context of the application. Instead of monitoring the application for potentially malicious inputs, RASP only processes inputs that could change the behavior or operation of the application. This approach has the potential to increase accuracy without significantly impacting the performance of the application. RASP solutions predominantly support the Java programming language and frameworks and other languages and frameworks to varying degrees including C#, PHP, Ruby, Python, Node.js, Go, and others. Some RASP solutions require a change to the application code itself, depending on the programming language, while others do not.

Alexander Fry, alexanderfry@student.sans.edu

1.1. RASP Technology Approaches

In general, there are four categories of RASP technology approaches: Servlet Filters, Plug-ins & Software Development Kits (SDK)s; Binary Instrumentation; Java Virtual Machine (JVM) Replacement; and Virtualization.

1.1.1. Servlet Filters, Plug-ins, SDKs

This technique utilizes web server plug-ins or Java servlets, usually implemented into Apache Tomcat or .NET to handle inbound HTTP Requests. Plug-ins inspect requests before they reach application code, applying detection methods to each Request. Requests that match known patterns, known as attack signatures, are then blocked (Cisar, 2016). The Prevoty RASP solution uses either plug-ins or custom code changes via an SDK for the specific programming language. The plug-ins or SDKs “monitor application behavior, analyze incoming user-input and data payloads, detect threats and sanitize data for safe execution within the application(s)” (Prevoty, 2018).

1.1.2. Binary Instrumentation

Binary instrumentation introduces monitoring and control elements into applications. As an instrumented application runs, the monitoring elements identify security events, including attacks, and the control elements log events, generate alerts, and block attacks. Depending on the programming language and framework, these elements integrate without requiring changes to the application source code or container. Contrast Security, Inc. states that its RASP solution leverages the Java Instrumentation API, which requires no changes to the application source code or Java virtual machine (Contrast, 2017).

1.1.3. JVM Replacement

Some RASP solutions are installed by replacing the standard application libraries, JAR files, or JVM. This allows the RASP solution to passively monitor application calls to supporting functions, providing a comprehensive view of data and control flows, enabling the use of fine-grained detection rules that can be applied as requests are intercepted (Adrian Lane, 2016).

1.1.4. Virtualization

Virtualization is also known as containerized runtime protection. This type of runtime protection implements a virtual container and uses rules to govern how the application is protected. Waratek states that its “container-based solution has the advantage of allowing rule configurations to be completely separated from the application and has no impact on the application lifecycle or its normal operations.” In addition, this approach does not require changes to application artifacts such as source code, deployment descriptors or binaries (Waratek, 2018).

1.2. Deployment Modes

RASP can be deployed in different modes depending on the solution. The following modes of operation and terminology are particular to Contrast Protect: Off, Monitor, Block, and Block at Perimeter - Block(P). In the Off mode, neither monitoring nor blocking is enabled. In Monitor mode, the RASP solution reports on web application attacks but does not block any attack. The Monitor mode may also log any identified threats. In Block mode, the RASP solution reports and blocks web application attacks. Finally, in Block(P) mode, the agent blocks an attack before the application processes the request. This mode blocks common attack patterns such as Cross-site scripting while operating in a similar manner to a WAF. However, it may potentially block legitimate requests which may reduce accuracy.

1.3. Drivers of RASP Adoption

RASP adoption is driven by a number of factors including Digital Transformation; Vulnerability Management; DevSecOps; Security Visibility and Security Operations Center (SOC) Fatigue; and Technical Debt, Legacy and Commercial off-the-shelf (COTS) Applications.

1.3.1. Digital Transformation

Many businesses and industries are undergoing a digital transformation from paper processes and physical assets to those that are software-driven and digital. In his now-famous essay, “Why Software Is Eating the World”, Marc Andreessen writes that “more and more major businesses and industries are being run on software and delivered as online services.” He goes on to cite examples of how traditional paper-brick-and-

Alexander Fry, alexanderfry@student.sans.edu

mortar businesses are transforming themselves into software businesses and how these new software-based businesses are disrupting industries (Andreesen, M., 2011, August 20). The transformation continues as today's organizations develop and deploy custom-coded applications using the advanced software engineering approaches of continuous integration, continuous delivery, and continuous deployment. The goal of these approaches is for organizations to be responsive to their customers and other stakeholders by building, testing, and releasing software with greater velocity and frequency which can/may result in reduced cost, time and risk of delivery. However, at times, updates are released without the necessary oversight from quality assurance and security teams. This has the potential to introduce software defects and security vulnerabilities that have not been adequately tested for in the code base.

These security vulnerabilities could be known vulnerabilities that the organization did not have a chance to fix or an unknown, so-called, Zero Day vulnerability. In either case, if there is an attack against that vulnerability in production, the RASP solution may detect the attack and block the exploitation of the vulnerability. This would also provide insight as to whether or not a specific vulnerability and/or class of vulnerability is being targeted in an application. This information could be shared with security and business executives and could be used to demonstrate the importance of application security initiatives, validate the need for additional investments, and help the organization prioritize their remediation efforts (Contrast Security, 2017).

1.3.2. Vulnerability Management

Applications that were once written are now assembled. Contrast Labs analyzed 1,857 production software applications, which included several thousand different open source libraries, frameworks, and modules. They determined that libraries constitute the largest percentage of application source code (79%). However, they are not the largest source of vulnerabilities (2.7%). The greatest number of vulnerabilities come from custom code, which comprises only 21% of applications on average but is responsible for 97.3% of vulnerabilities. Researchers concluded that "While libraries only account for a small share of the vulnerabilities within an application it is still critical for organizations

to track these vulnerabilities, known as CVEs because they create risks for the organization” (Contrast Security, 2017, July 21).

New vulnerabilities are discovered frequently in both custom code and open source libraries. If a vulnerability is discovered in custom code and is responsibly disclosed to the organization that owns the application, the organization may have sufficient time to fix the vulnerability before it can be discovered and exploited by an attacker. However, if a vulnerability is discovered in a library, especially in code that is in widespread use, and the vulnerability is publicly known and remotely discoverable, the application will be at risk of compromise until a fix is released. In both cases, a RASP solution may afford some protection until the vulnerability can be remediated. For example, a researcher discovered an Apache Struts 2 vulnerability (CVE-2017-5638) in March 2017. Apache Struts 2 is an open-source web application framework for developing Java web applications and is in widespread use. Exploitation of CVE-2017-5638 could allow the execution of arbitrary operating system commands. Within ten days of the vulnerability announcement by the Apache Foundation, there was evidence that attackers were using automated means to discover and exploit the vulnerability. With automated exploit scanning being conducted so quickly after the announcement, there was little time to apply a patch even if it would be available. However, a few RASP vendors announced that their RASP solution blocked attacks against this specific vulnerability and class of remote code injection vulnerability.

In addition, a RASP solution may not require changes to the production environment in order to protect against this attack. For example, Prevoty stated that their RASP solution blocked this attack with “no virtual patching, definition or signature updates required” (Prevoty, 2017). The Prevoty RASP solution utilizes a Java plug-in that integrates with the application. At runtime, the Java application, via the Java plug-in, calls Prevoty’s APIs to preprocess application artifacts such as content, database queries, and changes to user state, just before code execution within the application. An analysis engine using Language Theoretic Security (LangSec) identifies unwanted input and prevents it from exercising unexpected pathways through code. In short, Prevoty blocks everything except the approved operating system commands generated from within the

application. The application would be protected in this manner until the organization applies a patched Apache Struts 2 release (Prevoty, 2017).

In the event a RASP solution would not block attacks against a specific vulnerability, the RASP vendor could quickly write a new rule to block exploitation and make it available to customers (Prevoty, 2017, March 13).

1.3.3. DevSecOps

DevOps is “the combination of cultural philosophies, practices, and tools that increase an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes” (AWS, 2019). In DevOps, organizations establish a workflow between the business and customer; ensure instant feedback; and build a culture of innovation with frequent product iterations (Kim, G., 2012, August 22). DevSecOps improves upon these concepts by introducing security into an organization’s software development and infrastructure management processes. In DevSecOps, organizations establish a security workflow between the business and customer; ensure instant security feedback; and build a security culture (Contrast Security, n.d.).

DevSecOps organizations employ a security approach that allows rapid product development and deployment to production without introducing friction. In this model, security teams do not impose manual security checks that interrupt the release cycle and act as “toll gates.” Instead, security practices take the form of high performance “guardrails.” RASP solutions support this “guardrails-not-gates” approach where security is automated and continuous and runs as a passive background process that does not interfere with release cycles. In production, a RASP solution notifies the security and operations teams according to pre-configured rules and provides them with highly accurate attack forensics to facilitate an effective response.

A RASP solution could be employed in pre-production environments as well. This provides an organization with the capability to test the security posture of an application before it is deployed to production. For example, an application scheduled for release may contain important new features, but also known vulnerable code that cannot

be remediated in the short term. The organization could test how effective the RASP solution would be in blocking attacks against the known vulnerable code before the application is released to production. If the RASP solution blocks these attacks, they could deploy the application to production and patch the vulnerability in a later release. This would help accelerate time-to-market for the software without compromising security.

1.3.4. Security Visibility and SOC Fatigue

Many organizations use performance monitoring software to measure the performance of their production applications. This software provides information about the running application such as transaction details with method calls and line numbers, as well as metrics like CPU utilization and memory usage (New Relic, 2019). This high level of detail allows operations teams to find root causes and fix issues quickly. RASP solutions could provide the same level of visibility into attacks as organizations now have with application performance. For example, security teams would have actionable and timely threat intelligence across the entire application portfolio. They would know with greater confidence that an attack is taking place, including the type of attack, details about the payload, where the attack is coming from, if the attack had been seen previously, the area of the application targeted and the effectiveness of the RASP solution in blocking the attack.

The SOC is the nerve center of the security team in a larger organization and monitors threats across the organization. The SOC receives alerts from Security Information and Event Management (SIEM) systems that ingest data from the network and host-based Intrusion Detection Systems/Intrusion Prevention Systems, WAFs, custom application logging, NetFlow and other sources. These sources generate thousands of alerts. This overabundance of alerts leads to what is referred to as Alert Fatigue; the SOC expends energy sifting through false positives, and often lacks the context to prioritize which alerts are important. Some organizations deal with alert fatigue by tuning the alert threshold to permit only the number of alerts they are capable of processing. The drawback of this approach is that significant alerts could be missed, increasing the probability of a successful attack going unnoticed. In addition, typically

the SOC has poor visibility into the application layer of the application portfolio and they would have very little data to analyze in the event of an incident. Without improved fidelity and greater context concerning application alerts, the SOC is a poor return on investment for application security.

A RASP solution provides greater context into attacks to help relieve SOC fatigue by making the SOC more effective in prioritizing alerts and responding to incidents. For example, a RASP solution could send attack events to the SIEM as well. The RASP solution could also be integrated with software like Slack or Pager Duty so that incident response personnel could respond more effectively to notifications. Notifications could be configured for specific applications and users using conditional parameters: Category, Impact, Likelihood, URL, Class, and Method. Once a notification or alert is received, the incident responder could create an exclusion or virtual patch, blacklist an IP address, add a new rule, or suppress the event (Contrast Security, 2019).

1.3.5. Technical Debt, Legacy and COTS Applications

Technical debt grows when organizations fail to maintain applications and systems. For custom-coded applications, it is necessary to update the programming language version, frameworks, application server, database management system and third-party libraries, typically, at least, on an annual basis. COTS applications are commercial applications acquired by an organization. For COTS applications, it is necessary to apply patches at a point-level release and accrue for major version changes every three years or less. This helps to keep applications secure and puts organizations in a better position to retain top talent and continue to service the application. An application with heavy technical debt may become unserviceable or pose a risk to conducting business if there is a failure (Price, E., 2017, September 12).

Applications with technical debt are typically referred to as legacy applications. A legacy application may be an application that was launched years ago, lacks a build process or pipeline, has limited documentation, is no longer developed or maintained, but is still mission-critical to the business. There may even be a scarcity of developers with the necessary skillset to maintain the application. In addition, for both COTS and legacy applications, the organization may not possess the source code to allow it to easily fix

vulnerabilities discovered in these applications. However, even if the vulnerability cannot be fixed in code, it is still necessary to defend the application against exploitation of that vulnerability. A RASP solution could afford some protection for legacy applications through virtual patching. Virtual patching is defined as “the capability to apply both routine and emergency security patches without the need to change code and with no downtime” (Waratek, 2018). This is an advantage when the system to be patched is a critical business system where disruption to normal operations or scheduled downtime is unacceptable.

1.4. Vendor Landscape

While the choice of a RASP vendor is outside the scope of this paper, it should be noted that RASP vendors may shift business strategy to align themselves with customers, compete more effectively, and put themselves in a better position for increased funding or acquisition. As the vendor changes strategy or organizational structure, it is important for customers to re-evaluate the solution to ensure it is still a good fit for their information security program. Keeping up with industry trends and vendor offerings is an important part of the vendor selection and education process. For example, the Forrester New Wave™ produces a Runtime Application Self-Protection report that may prove useful in evaluating the market presence of RASP vendors, as seen below in Figure 2.

THE FORRESTER NEW WAVE™
 Runtime Application Self-Protection
 Q1 2018



*A gray marker indicates incomplete vendor participation.

Figure 2. Strategy vs Offering. From “The Forrester New Wave Runtime Application Self-Protection, Q1 2018,” 2018.

2. Lab Environment

A controlled lab environment was created for testing the effectiveness of the RASP solution against two known vulnerable web applications, WebGoat and NodeGoat. From the available RASP solutions in the marketplace, Contrast Protect was selected because it supports the technologies and frameworks used in the web applications and allows for a trial period that is sufficiently long enough to conduct the testing described in this paper. WebGoat is a deliberately vulnerable Java web application that is maintained by OWASP and designed to teach web application security best practices. Similar to WebGoat, NodeGoat is a deliberately vulnerable Node.js web application

Alexander Fry, alexanderfry@student.sans.edu

maintained by OWASP that is also designed as a training tool. Node.js is a modern service-side framework written in JavaScript. The objective of testing a Node.js application is to measure the effectiveness of a RASP solution in protecting a modern web application framework. In addition to security testing, the performance of each web application, with and without the RASP solution integrated, was measured using the New Relic application performance monitoring tool.

Attacks against each application were directed at the known vulnerabilities and were automated and timed using Burp Suite. Further research was carried out by attempting to evade RASP detection, for example, by using escaping and encoding techniques to vary the attack signature for Reflected Cross-site Scripting (XSS) attacks. The evasion approach was guided by techniques used to bypass common input filters and should not be considered exhaustive (Portswigger, 2018). In addition, the decision to vary the attack signature of Reflected XSS attacks, as opposed to all attacks, was informed by the necessary time constraints in conducting the overall research, and because Reflected XSS is a common vulnerability between the NodeGoat and WebGoat applications, which provides comparative results. There are other types of attacks with common encoding and escaping techniques that could be explored in greater detail, but for the purposes of this research, will not be a focus here.

In addition, exploitable Server-Side Request Forgery (SSRF) vulnerabilities were developed for each vulnerable web application to ensure that the RASP solution was not biased towards the public set of known vulnerabilities. To measure effectiveness, the result of each attack was compared against the successful exploitation of the vulnerability in the vulnerable web application. A successful attack results in the exploitation of the vulnerability, while an unsuccessful attack fails to exploit the vulnerability. If the attack fails and produces a performance impact or an error condition such as an HTTP Status Code 500, the impact and error were documented. The application logging capability was set to the highest level of verbosity and monitored to capture any error or impact to the application.

Before the attacks could be carried out against the “Goat” applications (WebGoat and NodeGoat), it was necessary to prepare the environment and applications for the

attack. These preparatory steps consisted of: Installation of Goat Applications; Integration of Goat Applications with Contrast; Attack Automation; and Vulnerability Development and are found in the supplemental material GitHub repository: <https://github.com/alexanderfry/rasp-research-paper>.

3. Attack Protection

Once the lab environment was ready, attack traffic was run against the “Goat” applications to measure the effectiveness of the RASP attack protection.

Contrast Protect characterizes the status of attacks as one of the following:

- **EXPLOITED** - The word “EXPLOITED” is the past tense of the word exploit and, in this context, signifies that an attack against a specific vulnerability was successful. The exploit itself is computer code, or a sequence of commands that takes advantage of a vulnerability and causes unintended functionality or unanticipated behavior to occur in the application (Technopedia, 2019).
- **PROBED** – Contrast defines a “PROBED” attack as one that “did not cause any adverse actions within the application” and was not “BLOCKED.” It is possible for a “PROBED” attack to exploit a vulnerability.
- **BLOCKED** – An attack that is prevented from exploiting a vulnerability. This action typically results from a Contrast Protect rule set to Block mode.
- **BLOCKED (P)** - An attack that is prevented from exploiting a vulnerability. This action typically results from a Contrast Protect rule set to Block(P) mode.

3.1. NodeGoat - Contrast Protect in Monitor Mode

NodeGoat is launched with the Contrast Protect Agent using the command `npm run contrast -- --agent.logger.level.debug`. It is confirmed that Monitor mode is enabled for all Node.js rules by editing the policy located in Applications->Policy->Protect.

The next step is to run the Burp Macro containing the NodeGoat attacks. Contrast Protect detects this as an Active Attack, shown in red in the upper right-hand corner, and lists the Attack Events in aggregate on the Monitor screen. “Live” and “All Environments” are selected, and the “Show probed” box is checked to ensure that all attack events are shown.

The Attack Events screen lists two of the attacks as “PROBED” and two of the attacks as “EXPLOITED” (Figure 3).

Source IP	Result	Application	Server	Rule	Time	URL	Attack Value
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	NoSQL Injection	23 minutes ago	/allocations/2	["threshold":1;return 1==1"]
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	23 minutes ago	/profile	<script>alert(1)</script>
:ffff:127.0.0.1	EXPLOITED	owasp-nodejs-goat	Nohochacym.local	NoSQL Injection	23 minutes ago	/allocations/2	1;return 1==1
:ffff:127.0.0.1	EXPLOITED	owasp-nodejs-goat	Nohochacym.local	Server-Side JavaScript Injection	23 minutes ago	/contributions	res.end(require('fs').readFileSync('/etc/passwd').toString())

Figure 3. Attack Events Screen with NodeGoat in Monitor Mode

3.1.1. Reflected XSS Attack Events Analysis and Cookie Stealer Exploit

The Reflected XSS attack event was characterized as “PROBED.” The Overview tab for this attack event states that “This attempted attack did not require specific blocking because it did not cause any adverse actions within the application.” This begs the question of how Contrast Protect determines if an attack event is characterized as “PROBED” or “EXPLOITED” and if it is deserving of blocking. The proof of concept exploit used in the Reflected Cross-Site Scripting attack is a relatively harmless script that creates an alert dialog with the number 1, `<script>alert(1)</script>`. However, attackers commonly use malicious exploits that attempt to compromise the user’s session. As an attempt to change the way the attack event is characterized, the proof of concept

exploit is changed to steal the session cookie and log the cookie to an attacker-controlled server. To accomplish this new proof-of-concept exploit, an attacker-controlled server is simulated by running an HTTP server on 127.0.0.1 port 8000, *python -m SimpleHTTPServer 8000*. The proof-of-concept exploit is replaced with a cookie stealing exploit: `<script>document.write('');</script>`. A script like this would typically be delivered to a victim via a phishing e-mail or drive-by attack. Upon executing this proof of concept exploit, the script delivers the cookie in an HTTP Request to the HTTP server (Figure 4).

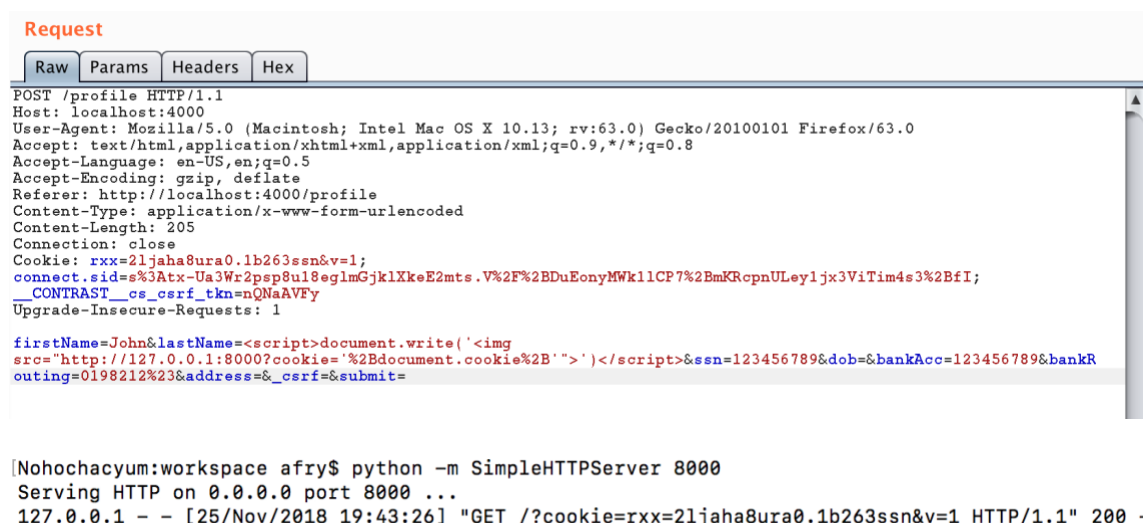


Figure 4. Cookie Stealer HTTP Request in Burp Repeater and Cookie Received

However, the Reflected XSS attack event is still listed as “PROBED” in the Contrast Portal.

3.1.2. NoSQL Attack Events Analysis

In continuing the analysis, a comparison of the two NoSQL Injection attack events shows that this is part of the same attack, but was identified as two different query strings, possibly as a result of how the page is rendered by the application. The overview of the exploited NoSQL Injection attack event describes where the suspicious value was seen entering the application through the HTTP Request Query String “threshold” and how that value altered the meaning of the NoSQL query. The code snippet and line

number provide context to remediate or patch the NoSQL Injection vulnerability (Figure 5).

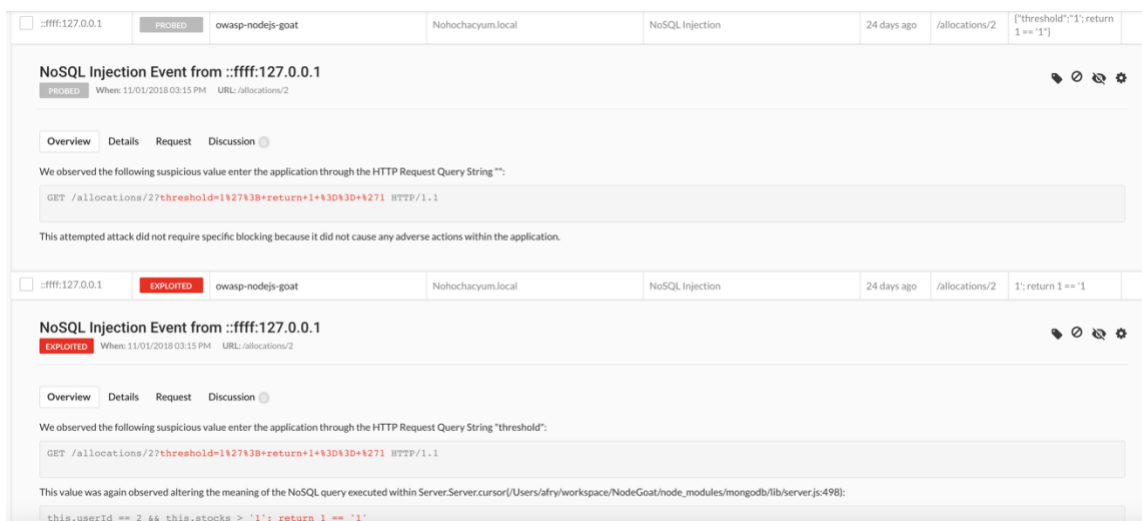


Figure 5. NodeGoat NoSQL Injection Attack Events

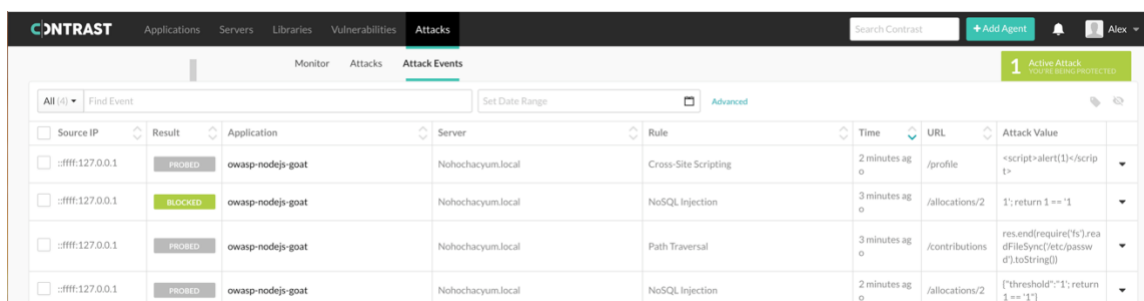
3.1.3. Remaining Attack Events Analysis

Contrast Protect created four attack events for attacks that attempted to exploit three vulnerabilities: NoSQL Injection, Server-Side JavaScript Injection, and Reflected XSS. Contrast Protect did not identify attacks against Insecure Direct Object Reference, Missing Function Level Access Control, and Unvalidated Redirect and does not provide rules to protect against exploitation of these vulnerabilities. The first two vulnerabilities are considered business logic flaws and are typically identified through manual testing. However, the third vulnerability, Unvalidated Redirect, has a rule in Contrast Assess, the pre-production product, but not in Contrast Protect. The Contrast Assess rule identifies the Unvalidated Redirect vulnerability as, “Unvalidated Redirect from Untrusted Sources on /learn page”. This signifies that the Contrast agent software that is used by both Assess and Protect is capable of identifying this vulnerability but a Protect rule has not yet been written.

3.2. NodeGoat - Contrast Protect in Block Mode

All of the attack events have been analyzed in Monitor mode. Now the rules are changed to Block mode to test if the attacks would be blocked. It is confirmed that Block mode is enabled for all Node.js rules by viewing the policy located in Applications->Policy->Protect.

The Burp Macro was run again. All four attack events were observed in Block mode. However, the Attack Events screen shows that only one attack was “BLOCKED”. The other three attacks are shown as “PROBED.” In addition, the Server-Side JavaScript Injection vulnerability is identified as a Path Traversal vulnerability (Figure 6).



Source IP	Result	Application	Server	Rule	Time	URL	Attack Value
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacyum.local	Cross-Site Scripting	2 minutes ago	/profile	<script>alert(1)</script>
:ffff:127.0.0.1	BLOCKED	owasp-nodejs-goat	Nohochacyum.local	NoSQL Injection	3 minutes ago	/allocations/2	1; return 1 == '1'
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacyum.local	Path Traversal	3 minutes ago	/contributions	res.end(require('fs').readFileSync('/etc/passwd').toString())
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacyum.local	NoSQL Injection	2 minutes ago	/allocations/2	["threshold":1]; return 1 == '1'

Figure 6. Attack Events Screen with NodeGoat in Block Mode

To confirm that all attacks were blocked, the HTTP Responses in Burp Proxy were inspected. All HTTP Responses returned HTTP status code “403 – Forbidden”, preventing the exploitation of the vulnerabilities.

3.3. NodeGoat - Contrast Protect in Block at Perimeter Mode

Contrast Protect was then changed to “Block at Perimeter” Block (P) mode for the rules that support this option: Path Traversal, Cross-Site Scripting, and NoSQL Injection.

Then the Burp Macro was run again. Three attack events were observed in Block and Block(P). The Attack Events screen confirmed that three attacks were blocked at perimeter “BLOCKED (P).” Both Block and Block(P) mode identified the Server-Side JavaScript Injection vulnerability as a Path Traversal vulnerability (Figure 7).

Source IP	Result	Application	Server	Rule	Time	URL	Attack Value
:ffff:127.0.0.1	BLOCKED (P)	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	11 minutes ago	/profile	<script>alert(1)</script>
:ffff:127.0.0.1	BLOCKED (P)	owasp-nodejs-goat	Nohochacym.local	Path Traversal	11 minutes ago	/contributions	res.end(require('fs').readFileSync('/etc/passwd').toString())
:ffff:127.0.0.1	BLOCKED (P)	owasp-nodejs-goat	Nohochacym.local	NoSQL Injection	11 minutes ago	/allocations/2	['threshold':1]:return 1==1

Figure 7. Attack Events Screen with NodeGoat in Block and Block(P) Mode

It was further confirmed that all attacks were blocked by inspecting the HTTP Responses for each attack in Burp Proxy. All Responses returned HTTP status code “403 – Forbidden”, preventing the exploitation of the vulnerability.

3.4. NodeGoat – Attempt to Evade RASP Detection

All XSS attacks were blocked by Contrast Protect. However, attackers commonly use evasion techniques such as encoding and escaping to evade detection and blocking. To test the effectiveness of Contrast Protect in detecting XSS attacks using these common evasion techniques, first the policy was changed back to Monitoring mode. Then, attacks were conducted using nine variations of the Reflected XSS proof of concept exploit: `<script>alert(1)</script>`. All nine of the XSS attacks were detected by Contrast Protect (Figure 8).

Source IP	Result	Application	Server	Rule	Time	URL	Attack Value
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	3 minutes ago	/profile	<script>alert(document.cookie)</script>
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	5 minutes ago	/profile	<script>alert(1);replace(/eval/);</script>
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	7 minutes ago	/profile	<script>eval(atob('amF2YXN0cm9udDphbGVyIDCgRQ'))</script>
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	8 minutes ago	/profile	<script>eval('a'+'lert(1)');</script>
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	NoSQL Injection	8 minutes ago	/profile	['firstName':'John','lastName':'<script>eval('a'+'lert(1)');</script>','address':'','city':'','submits':'']
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	10 minutes ago	/profile	<script>eval('a'+'lert(1)');</script>
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	13 minutes ago	/profile	<script>alert(0072'u0074(1)</script>
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	14 minutes ago	/profile	<script>alert(00cert(1)</script>
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	19 minutes ago	/profile	<script>alert(0072'u0074(String.fromCharCode(88,83,83))</script>
:ffff:127.0.0.1	PROBED	owasp-nodejs-goat	Nohochacym.local	Cross-Site Scripting	22 minutes ago	/profile	<script>alert(00cert(String.fromCharCode(88,83,83))</script>

Figure 8. All Evasion Attempts Detected in Monitor Mode

After switching to Block and Block(P) mode, all nine of the attacks were blocked by Contrast Protect and verified in both the Browser and Burp Proxy.

3.4.1. NodeGoat - SSRF Attack Events Analysis

The Vulnerability Development section of the supplemental material described how the SSRF vulnerability was developed and demonstrated exploitability. To test the ability of Contrast Protect to block attacks against this vulnerability, the SSRF attacks were run against NodeGoat with a Burp Macro. After running the attacks, the Contrast Portal was reviewed to ascertain if the SSRF attack events had been detected. However, the monitor screen displayed “No Attacks Detected.” Even going back to “Last Hour” and “Last Day” did not show any attacks. This confirms that attacks against the SSRF vulnerability are not detected by the current Contrast Protect ruleset (Figure 9).

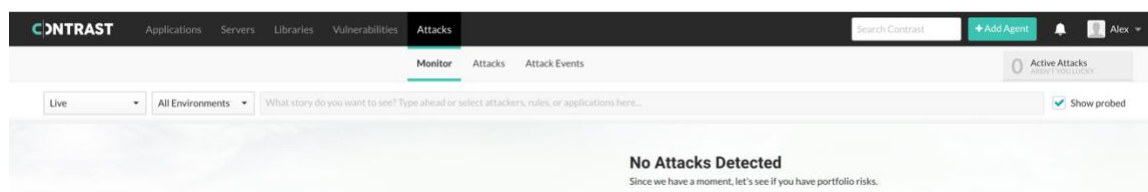


Figure 9. NodeGoat SSRF No Attacks Detected

As a further confirmation, Contrast Assess was enabled and the SSRF attack was run again. The Vulnerabilities tab should show that the vulnerability is picked up by Contrast Assess. However, there was no SSRF vulnerability listed nor was there any vulnerabilities listed for the “/research” page (Figure 10).

The screenshot shows the Contrast web interface with the 'Vulnerabilities' tab selected. A table lists 14 vulnerabilities. The first vulnerability, 'Unvalidated Redirect from Untrusted Sources on "/research" page', is highlighted in orange, indicating a 'MEDIUM' severity. Other vulnerabilities include 'Pages Without Anti-Clickjacking Controls' (NOTE), 'Anti-Caching Controls Missing' (NOTE), 'Forms Without Autocomplete Prevention' (NOTE), 'Parameter Pollution' (MEDIUM), 'Hardcoded Cryptographic Key' (MEDIUM), 'Cross-Site Scripting from Untrusted Sources' (HIGH), 'NoSQL Injection from Untrusted Sources' (CRITICAL), and 'Unsafe Code Execution from Untrusted Sources' (HIGH).

Vulnerability	Severity	Application	Last Detected	Status
<input type="checkbox"/> Unvalidated Redirect from Untrusted Sources on "/research" page	MEDIUM	owasp-nodejs-goat	2 minutes ago	Reported
<input type="checkbox"/> Pages Without Anti-Clickjacking Controls on 9 pages	NOTE	owasp-nodejs-goat	6 minutes ago	Reported
<input type="checkbox"/> Anti-Caching Controls Missing on 9 pages	NOTE	owasp-nodejs-goat	6 minutes ago	Reported
<input type="checkbox"/> Forms Without Autocomplete Prevention on 6 pages	NOTE	owasp-nodejs-goat	6 minutes ago	Reported
<input type="checkbox"/> Parameter Pollution on 1 page	MEDIUM	owasp-nodejs-goat	6 minutes ago	Reported
<input type="checkbox"/> Hardcoded Cryptographic Key in ./config/env/all.js line 5	MEDIUM	owasp-nodejs-goat	7 minutes ago	Reported
<input type="checkbox"/> Cross-Site Scripting from Untrusted Sources on "/allocations/userId" page	HIGH	owasp-nodejs-goat	4 days ago	Reported
<input type="checkbox"/> NoSQL Injection from Untrusted Sources on "/allocations/userId" page	CRITICAL	owasp-nodejs-goat	4 days ago	Reported
<input type="checkbox"/> Cross-Site Scripting from Untrusted Sources on "/login" page	HIGH	owasp-nodejs-goat	24 days ago	Reported
<input type="checkbox"/> Unvalidated Redirect from Untrusted Sources on "/learn" page	MEDIUM	owasp-nodejs-goat	24 days ago	Reported
<input type="checkbox"/> 'md5' hash algorithm used at /Users/afry/workspace/NodeGoat/node_modules/...	MEDIUM	owasp-nodejs-goat	24 days ago	Reported
<input type="checkbox"/> 'sha1' hash algorithm used at /Users/afry/workspace/NodeGoat/node_modules/...	MEDIUM	owasp-nodejs-goat	24 days ago	Reported
<input type="checkbox"/> Unsafe Code Execution from Untrusted Sources on "/contributions" page	HIGH	owasp-nodejs-goat	24 days ago	Reported

Figure 10. NodeGoat SSRF No Research Page Vulnerabilities Listed

To verify that the Contrast agent is monitoring the “/research” page, the SSRF vulnerability is converted to an Unvalidated Redirect vulnerability, for which Contrast Assess has a rule.

The SSRF attack was run again. This time, the vulnerability was identified as an Unvalidated Redirect in the “/research” page (Figure 11). This an indication that SSRF is a gap in both the Contrast Assess and Protect ruleset coverage for Node.js applications.

This screenshot is similar to Figure 10 but shows the results after an SSRF attack. The 'Unvalidated Redirect from Untrusted Sources on "/research" page' vulnerability is now listed with a 'MEDIUM' severity and was detected '2 minutes ago'. The other vulnerabilities remain the same as in Figure 10.

Vulnerability	Severity	Application	Last Detected	Status
<input type="checkbox"/> Unvalidated Redirect from Untrusted Sources on "/research" page	MEDIUM	owasp-nodejs-goat	2 minutes ago	Reported
<input type="checkbox"/> Pages Without Anti-Clickjacking Controls on 9 pages	NOTE	owasp-nodejs-goat	6 minutes ago	Reported
<input type="checkbox"/> Anti-Caching Controls Missing on 9 pages	NOTE	owasp-nodejs-goat	6 minutes ago	Reported
<input type="checkbox"/> Forms Without Autocomplete Prevention on 6 pages	NOTE	owasp-nodejs-goat	6 minutes ago	Reported
<input type="checkbox"/> NoSQL Injection from Untrusted Sources on "/login" page	CRITICAL	owasp-nodejs-goat	6 minutes ago	Reported
<input type="checkbox"/> Parameter Pollution on 1 page	MEDIUM	owasp-nodejs-goat	6 minutes ago	Reported
<input type="checkbox"/> Hardcoded Cryptographic Key in ./config/env/all.js line 5	MEDIUM	owasp-nodejs-goat	7 minutes ago	Reported
<input type="checkbox"/> Cross-Site Scripting from Untrusted Sources on "/allocations/userId" page	HIGH	owasp-nodejs-goat	4 days ago	Reported
<input type="checkbox"/> NoSQL Injection from Untrusted Sources on "/allocations/userId" page	CRITICAL	owasp-nodejs-goat	4 days ago	Reported
<input type="checkbox"/> Cross-Site Scripting from Untrusted Sources on "/login" page	HIGH	owasp-nodejs-goat	24 days ago	Reported
<input type="checkbox"/> Unvalidated Redirect from Untrusted Sources on "/learn" page	MEDIUM	owasp-nodejs-goat	24 days ago	Reported
<input type="checkbox"/> 'md5' hash algorithm used at /Users/afry/workspace/NodeGoat/node_modules/...	MEDIUM	owasp-nodejs-goat	24 days ago	Reported
<input type="checkbox"/> 'sha1' hash algorithm used at /Users/afry/workspace/NodeGoat/node_modules/...	MEDIUM	owasp-nodejs-goat	24 days ago	Reported
<input type="checkbox"/> Unsafe Code Execution from Untrusted Sources on "/contributions" page	HIGH	owasp-nodejs-goat	24 days ago	Reported

Figure 11. NodeGoat Unvalidated Redirect Vulnerability in Research Page

3.4.2. NodeGoat – Attack Protection Summary

Contrast Protect, in Monitor mode, labeled a Reflected XSS attack that is capable of stealing a user's session cookie and sending it to an attacker-controlled server as "PROBED" rather than "EXPLOITED". Contrast Protect consistently labeled Reflected XSS attacks as "PROBED" rather than "EXPLOITED" and labeled the NoSQL Injection attack as "EXPLOITED." A reason for the difference in labeling could be that Contrast Protect considers Injection attacks, such as the NoSQL Injection attack, as higher impact, which is deserving of the "EXPLOITED" label. After all, injection attacks target the service-side and could compromise the application itself, the underlying operating system or lead to a data breach, whereas Reflected XSS target the client-side or individual users.

Contrast Protect, in Monitor Mode, identified a single NoSQL Injection attack as two separate attack events. One of the attack events was labeled "PROBED" and the other "EXPLOITED". This could be caused by how the page was rendered by the application or how it was blocked by Contrast, resulting in two separate data flows.

Contrast Protect missed Insecure Direct Object Reference, Missing Function Level Access Control, Unvalidated Redirect and SSRF attacks. There are no Protect rules for Insecure Direct Object Reference and Missing Function Level Access Control, but these vulnerabilities are considered business logic flaws and are typically identified through manual testing, so it is reasonable that no rules exist for these vulnerabilities. The Unvalidated Redirect and SSRF vulnerabilities are capable of being identified by Contrast. Contrast Assess has a Node.js rule for Unvalidated Redirect and a Java rule for SSRF. However, Contrast Protect missed both of these attacks and does not have a rule for them. Therefore, the lack of a rule for Unvalidated Redirect and SSRF is a gap in Node.js protection.

Contrast Protect, in Block and Block(P) Mode, characterized an attack against a Server-Side JavaScript Injection vulnerability as an attack against a Path Traversal vulnerability. The file system path used in the Server-Side JavaScript vulnerability proof

of concept exploit, */etc/passwd*, is often used to test for Path Traversal vulnerabilities. Contrast Protect misidentifies this file system path as a component of a Path Traversal exploit.

3.5. WebGoat – Contrast Protect in Monitor Mode

It is necessary to monitor the application to ensure that Contrast Protect detects the attacks. The application is launched with the Contrast Protect Agent enabled. The Agent build number is 3.5.8.5149. The command to launch WebGoat is *java -Dcontrast.standalone.appname=Webgoat -Dcontrast.override.appname=Webgoat -javaagent:contrast.jar -jar webgoat-server-v8.0.0.SNAPSHOT.jar -server.port=9080 -server.address=192.168.0.21*.

Once WebGoat is running, Monitor mode is enabled for all Java rules by editing the policy located in Applications->Policy->Protect.

The next step was to run the Burp Macro containing the WebGoat attacks. Contrast Protect detects this as an Active Attack, shown in red in the upper right-hand corner, and lists the Attack Events in aggregate on the Monitor screen. “Live” and “All Environments” are selected and the “Show probed” box is checked to ensure that all attack events are shown.

The Attack Events screen shows six attack events. One of the attack events is listed as “PROBED” and five are listed as “EXPLOITED” (Figure 12).

Source IP	Result	Application	Server	Rule	Time	URL	Attack Value
192.168.0.12	PROBED	Webgoat	ubu64	Cross-Site Scripting	6 minutes ago	/WebGoat/C...	4128.3214.0002.1999 <script>alert('my javascript here')</script>
192.168.0.12	EXPLOITED	Webgoat	ubu64	XML External Entity Processing	6 minutes ago	/WebGoat/x...	...[<!ENTITY js SYSTEM file:///...
192.168.0.12	EXPLOITED	Webgoat	ubu64	XML External Entity Processing	6 minutes ago	/WebGoat/x...	...[<!ENTITY root SYSTM file:///...
192.168.0.12	EXPLOITED	Webgoat	ubu64	SQL Injection	6 minutes ago	/WebGoat/S...	1 OR 1=1
192.168.0.12	EXPLOITED	Webgoat	ubu64	SQL Injection	6 minutes ago	/WebGoat/S...	Smith: SELECT * FROM user_system_data WHERE '1'='1
192.168.0.12	EXPLOITED	Webgoat	ubu64	SQL Injection	6 minutes ago	/WebGoat/S...	' OR '1'='1

Figure 12. Attack Events Screen with WebGoat in Monitor Mode

The Overview tab from the Probed Attack Events Screen states that “This attempted attack did not require specific blocking because it did not cause any adverse actions within the application.” This is the same response that was received for the NodeGoat application. In NodeGoat, when the proof of concept exploit for Cross-Site Scripting was modified to steal the session cookie, the Contrast response remained the same.

Contrast Protect created six attack events for attacks attempting to exploit six vulnerabilities: SQL Injection, XXE and Cross-site Scripting. It missed the Insecure Direct Object Reference vulnerability. There are no Protect rules for the Insecure Direct Object Reference vulnerability because it is considered a business logic flaw and is typically identified through manual testing.

3.6. WebGoat – Contrast Protect in Block Mode

After the attack events have been analyzed in Monitor mode, the rules are changed to Block Mode to test if the attacks would be blocked. Block mode is enabled for all Java rules by editing the policy located in Applications->Policy->Protect.

The Burp Macro was run again. All six attack events were observed in Block mode. The Attack Events screen shows that five attack events were “BLOCKED.” The Cross-Site Scripting attack event is shown as “PROBED.” Selecting the Cross-Site Scripting attack event provides additional information in four tabs: Overview, Details, Request and Discussion. The Overview shows the attack event observed by Contrast Protect and states, “This attempted attack did not require specific blocking because it did not cause any adverse actions within the application.” This was the same explanation given when Contrast Protect was run in Monitor mode. The following is a screenshot of the Attack Events Screen in Block Mode (Figure 13).

Source IP	Result	Application	Server	Rule	Time	URL	Attack Value
192.168.0.12	PROCEED	Webgoat	ubu64	Cross-Site Scripting	6 minutes ago	/WebGoat/C...	4128 3214 0002 1999 <script>alert('my javascript here')</script>
192.168.0.12	BLOCKED	Webgoat	ubu64	XML External Entity Processing	6 minutes ago	/WebGoat/x...	..[+ENTITY root SYSTEM file:///..]
192.168.0.12	BLOCKED	Webgoat	ubu64	XML External Entity Processing	6 minutes ago	/WebGoat/x...	..[+ENTITY js SYSTEM file:///..]
192.168.0.12	BLOCKED	Webgoat	ubu64	SQL Injection	6 minutes ago	/WebGoat/S...	Smith: SELECT * FROM user_system_data WHERE '1'='1
192.168.0.12	BLOCKED	Webgoat	ubu64	SQL Injection	6 minutes ago	/WebGoat/S...	1 OR 1=1
192.168.0.12	BLOCKED	Webgoat	ubu64	SQL Injection	6 minutes ago	/WebGoat/S...	' OR '1'='1

Figure 13. Attack Events Screen with WebGoat in Block Mode

To confirm that each attack event was blocked, the HTTP Responses in Burp Proxy were inspected. Five of the attacks failed, preventing the exploitation of the vulnerabilities. However, the Cross-Site Scripting attack succeeded (Figure 14).

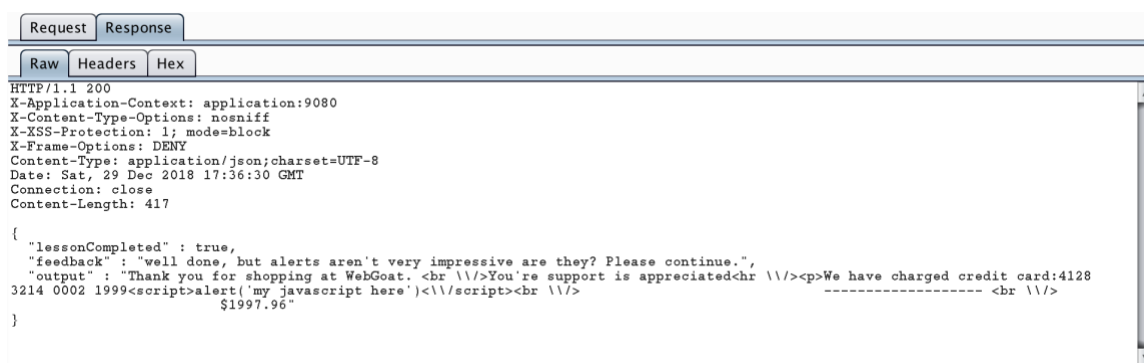


Figure 14. Successful Cross-Site Scripting Attack in Block Mode

3.7. WebGoat – Contrast Protect in Block and Block (P) Mode

Contrast Protect was changed to Block(P) mode to test how the findings changed. The rules that don't support Block(P) mode are left in Block mode. The Cross-Site Scripting and SQL Injection vulnerabilities have a Block(P) rule, and the XML External Entity Processing vulnerabilities have a Block rule.

The Burp Macro was run again. The monitor screen shows all seven attack events. The Attack Events screen confirms that all attacks were blocked. The Cross-Site

Scripting attacks and SQL Injection attacks were Blocked at Perimeter “BLOCKED (P)” whereas the XML External Entity Processing (XXE) attacks were “BLOCKED” (Figure 15).

Source IP	Result	Application	Server	Rule	Time	URL	Attack Value
192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	4 minutes ago	/WebGoat/er...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+...27my+javascript+her e%27%3C%2Fscript%3E&field2=111
192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	4 minutes ago	/WebGoat/C...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+...27my+javascript+her e%27%3C%2Fscript%3E&field2=111
192.168.0.12	BLOCKED	Webgoat	ubu64	XML External Entity Processing	4 minutes ago	/WebGoat/x...	..[<ENTITY js SYSTEM file:///...]
192.168.0.12	BLOCKED	Webgoat	ubu64	XML External Entity Processing	4 minutes ago	/WebGoat/x...	..[<ENTITY root SYST EM file:///...]
192.168.0.12	BLOCKED (P)	Webgoat	ubu64	SQL Injection	4 minutes ago	/WebGoat/S...	Smith: SELECT * FRO M user_system_data W HERE '1'='1
192.168.0.12	BLOCKED (P)	Webgoat	ubu64	SQL Injection	4 minutes ago	/WebGoat/S...	1 OR 1=1
192.168.0.12	BLOCKED (P)	Webgoat	ubu64	SQL Injection	4 minutes ago	/WebGoat/S...	'OR '1'='1

Figure 15. Attack Events Screen with WebGoat in Block and Block(P) Mode

In this test run, two XSS attack events were identified compared to the one XSS attack event that was identified in Monitor and Block mode. Figure 16, below, shows that the first XSS attack event was the error page, /WebGoat/error.html, and the second was the application path for the XSS vulnerability, /WebGoat/CrossSiteScripting/attack5a.

Cross-Site Scripting Event from 192.168.0.12 (guest1)

BLOCKED (P) When: 12/29/2018 06:49 PM URL: /WebGoat/error.html

Overview Details Request Discussion

We observed the following suspicious value enter the application through the HTTP Request Query String:

```
GET /WebGoat/error.html?QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+1999%3Cscript%3Ealert(%27my+javascript+here%27)%3C%2Fscript%3E&field2=111 HTTP/1.0
```

We blocked this attack.

Cross-Site Scripting Event from 192.168.0.12 (guest1)

BLOCKED (P) When: 12/29/2018 06:49 PM URL: /WebGoat/CrossSiteScripting/attack5a

Overview Details Request Discussion

We observed the following suspicious value enter the application through the HTTP Request Query String:

```
GET /WebGoat/CrossSiteScripting/attack5a?QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+1999%3Cscript%3Ealert(%27my+javascript+here%27)%3C%2Fscript%3E&field2=111 HTTP/1.0
```

We blocked this attack.

Figure 16. Two XSS Attack Events with WebGoat in Block and Block(P) Mode

The additional XSS attack event appears to be the same attack reflected in/on the WebGoat error page in addition to the vulnerable page in the WebGoat application. The following Internal Server Error in the HTTP Response caused the additional XSS attack event (Figure 17).

```

Request  Response
Raw  Headers  Hex  HTML  Render
HTTP/1.1 500
Content-Type: text/html; charset=utf-8
Content-Language: en
Content-Length: 3032
Date: Sat, 29 Dec 2018 22:56:22 GMT
Connection: close

<doctype html><html lang="en"><head><title>HTTP Status 500 - Internal Server Error</title><style type="text/css">h1
{font-family:Tahoma,Arial,sans-serif;color:white;background-color:#525D76;font-size:22px;} h2
{font-family:Tahoma,Arial,sans-serif;color:white;background-color:#525D76;font-size:16px;} h3
{font-family:Tahoma,Arial,sans-serif;color:white;background-color:#525D76;font-size:14px;} body
{font-family:Tahoma,Arial,sans-serif;color:black;background-color:white;} b
{font-family:Tahoma,Arial,sans-serif;color:white;background-color:#525D76;} p
{font-family:Tahoma,Arial,sans-serif;background:white;color:black;font-size:12px;} a {color:black;} a.name {color:black;} .line
{height:1px;background-color:#525D76;border:none;}</style></head><body><div>HTTP Status 500 - Internal Server Error</div><div class="line" /><p><b>Exception Report</b><p><b>Message</b> Attack detected</p><p><b>Description</b> The server
encountered an unexpected condition that prevented it from fulfilling the
request.</p><p><b>Exception</b></p><p><pre>com.contrastsecurity.agent.plugins.rasp.AttackBlockedException: Attack detected
com.contrastsecurity.agent.plugins.rasp.d.a(AttackListener.java:620)
com.contrastsecurity.agent.plugins.rasp.d.a(AttackListener.java:566)
com.contrastsecurity.agent.plugins.rasp.d.a(AttackListener.java:505)
com.contrastsecurity.agent.plugins.rasp.d.a(AttackListener.java:215)
com.contrastsecurity.agent.http.HttpManager.onStart(HttpManager.java:201)
com.contrastsecurity.agent.plugins.frameworks.j2ee.J2EEController.onFirstRequestHandlerInvoked(J2EEController.java:425)
org.springframework.security.web.FilterChainProxy.doFilter(FilterChainProxy.java)
org.springframework.web.filter.DelegatingFilterProxy.invokeDelegate(DelegatingFilterProxy.java:347)
org.springframework.web.filter.DelegatingFilterProxy.doFilter(DelegatingFilterProxy.java:263)
org.springframework.web.filter.RequestContextFilter.doFilterInternal(RequestContextFilter.java:99)
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
org.springframework.web.filter.HttpPutFormContentFilter.doFilterInternal(HttpPutFormContentFilter.java:109)
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
org.springframework.web.filter.HiddenHttpMethodFilter.doFilterInternal(HiddenHttpMethodFilter.java:81)
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:197)
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
org.springframework.boot.actuate.autoconfigure.MetricsFilter.doFilterInternal(MetricsFilter.java:106)
org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:107)
</pre><p><b>Note</b> The full stack trace of the root cause is available in the server logs.</p><div class="line" /><h3>Apache
Tomcat/8.5.29</h3></body></html>

```

Figure 17. XSS Attack Response with WebGoat in Block and Block(P) Mode

It was further confirmed that all other attacks were blocked by inspecting the HTTP Responses for each attack in Burp Proxy.

3.8. WebGoat – Attempt to Evade RASP Detection

All XSS attacks were blocked by Contrast Protect. However, attackers commonly use evasion techniques such as encoding and escaping to evade detection and blocking. To test the effectiveness of Contrast Protect in detecting XSS attacks using these common evasion techniques, first the policy was changed back to Monitoring mode.

Then, attacks were conducted using six variations of the Reflected XSS proof of concept exploit: `<script>alert(1)</script>`. This was the same set of nine proof of concept exploits used to test evasion in NodeGoat. For WebGoat, only six of the nine attacks succeeded in exploiting the Reflected XSS vulnerability. The other three attacks caused an exception: “java.lang.IllegalArgumentException: Invalid character found in the request target. The valid characters are defined in RFC 7230 and RFC 3986.” However, all six of the successful attacks were detected by Contrast Protect (Figure 18).

Source IP	Result	Application	Server	Rule	Time	URL	Attack Value
192.168.0.12	PROBIDED	Webgoat	ubu64	Cross-Site Scripting	7 minutes ago	/WebGoat/C...	4128 3214 0002 1999 <script>alert(document.cookie)</script>
192.168.0.12	PROBIDED	Webgoat	ubu64	Cross-Site Scripting	7 minutes ago	/WebGoat/C...	4128 3214 0002 1999 <script>alert(1)/replacel/.eval)</script>
192.168.0.12	PROBIDED	Webgoat	ubu64	Cross-Site Scripting	8 minutes ago	/WebGoat/C...	4128 3214 0002 1999 <script>eval(atob(amF2YXNjbmlwdDphbGVydCgxKQ));</script>
192.168.0.12	PROBIDED	Webgoat	ubu64	Cross-Site Scripting	8 minutes ago	/WebGoat/C...	4128 3214 0002 1999 <script>eval('al'ert(1));</script>
192.168.0.12	PROBIDED	Webgoat	ubu64	Cross-Site Scripting	11 minutes ago	/WebGoat/C...	4128 3214 0002 1999 <script>aleuu0072uu007(String.fromCharCode(88,83,83))</script>
192.168.0.12	PROBIDED	Webgoat	ubu64	Cross-Site Scripting	11 minutes ago	/WebGoat/C...	4128 3214 0002 1999 <script>auu006cert(String.fromCharCode(88,83,83))</script>

Figure 18. All Evasion Attempts Detected in Monitor Mode

After switching to Block and Block(P) mode, all six of the attacks were blocked by Contrast Protect and were verified in both the Browser and Burp Proxy. Each attack generated two attack events. The additional XSS attack event appears to be the same attack reflected in/on the WebGoat error page in addition to the vulnerable page in the WebGoat application. This same behavior was witnessed with the default proof of concept exploit for XSS. In the following screenshot, the second column from the left shows that each of the twelve attacks were blocked “BLOCKED(P)” and the rightmost column shows each attack payload (Figure 19).

CONTRAST Applications Servers Libraries Vulnerabilities Attacks Search Contrast Add Agent Alex									
All (4712) Find Event		Set Date Range		Advanced					
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	7 minutes ago	/WebGoat/er...	=4128+3214+0002+... let(document.cookie)%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	7 minutes ago	/WebGoat/C...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+... let(document.cookie)%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	16 minutes ago	/WebGoat/er...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+... (1);replace(/eval%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	16 minutes ago	/WebGoat/C...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+... (1);replace(/eval%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	21 minutes ago	/WebGoat/er...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+... NjcmwDphGvYdCgkKQ);%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	21 minutes ago	/WebGoat/C...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+... NjcmwDphGvYdCgkKQ);%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	25 minutes ago	/WebGoat/er...	=4128+3214+0002+... (1);%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	25 minutes ago	/WebGoat/C...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+... (1);%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	28 minutes ago	/WebGoat/C...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+... fromCharCode(88,83,83)%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	28 minutes ago	/WebGoat/er...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+... fromCharCode(88,83,83)%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	32 minutes ago	/WebGoat/C...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+... fromCharCode(88,83,83)%3C%2Fscript%3E&field2=111	▼
<input type="checkbox"/>	192.168.0.12	BLOCKED (P)	Webgoat	ubu64	Cross-Site Scripting	32 minutes ago	/WebGoat/er...	QTY1=16QTY2=16QTY3=16QTY4=16field1=4128+3214+0002+... fromCharCode(88,83,83)%3C%2Fscript%3E&field2=111	▼

Figure 19. All Evasion Attempts Blocked in Blocked(P) Mode

3.8.1. WebGoat - SSRF Attack Events Analysis

The Vulnerability Development section of the supplemental material describes how the SSRF vulnerability was developed and demonstrated exploitability. To test the ability of Contrast Protect to block attacks against this vulnerability, the SSRF attacks were run against WebGoat with a Burp Macro. After running the attacks, the Contrast Portal was reviewed to ascertain if the SSRF attack events had been detected. However, the monitor screen displayed “No Attacks Detected.” Even going back to “Last Hour” and “Last Day” did not show any attacks. This appears to confirm that attacks against the SSRF vulnerability are not detected by the current Contrast Protect ruleset.

Alexander Fry, alexanderfry@student.sans.edu

As further confirmation Contrast Assess was enabled and the SSRF attack was run again. The Vulnerabilities tab shows that Contrast Assess identifies the SSRF vulnerability. The Policy tab shows that the SSRF rule is enabled. This confirms that SSRF is a gap in the current Contrast Protect ruleset coverage for Java applications (Figure 20).

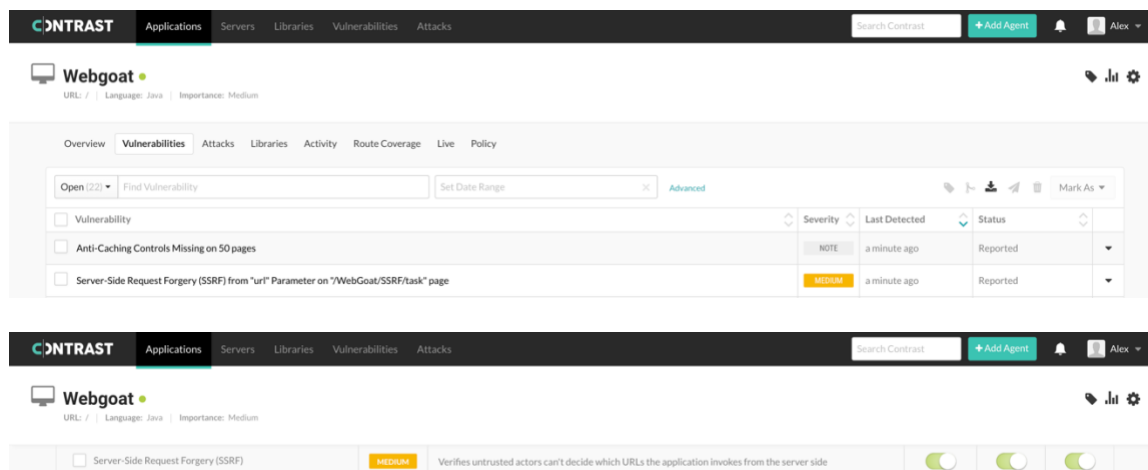


Figure 20. WebGoat SSRF Vulnerability

3.8.2. WebGoat – Attack Protection Summary

Contrast Protect, in Block mode, did not block the proof of concept XSS attack. Instead of “BLOCKED”, the attack was labeled as “PROBED.” This appears to be an anomaly as it blocked all other Reflected XSS attacks in Block or Block(P) mode.

Contrast Protect missed Insecure Direct Object Reference and Server-Side Request Forgery attacks. There is no Protect rule for Insecure Direct Object Reference, but this vulnerability is considered a business logic flaw and is typically identified through manual testing, so it is reasonable that no rule exists. SSRF vulnerabilities are capable of being identified by Contrast. This was proven using the Contrast Assess product. However, Contrast Protect missed this attack and does not have a rule for SSRF. Therefore, the lack of a rule for SSRF is considered a gap in Java protection.

Contrast Protect, in Block(P) mode, caused an Internal Server Error, HTTP 500, when blocking the XSS attack. The Contrast Agent created an AttackBlockedException that appears to have led to the Internal Server Error.

4. Performance Measurement

Application performance was measured for the Goat applications under different traffic and attack scenarios both with and without Contrast Protect enabled. To make the generation of metrics repeatable and consistent, the Goat applications were run in VMs isolated from other running applications; Burp Macros were created to automate the sending of HTTP traffic and the traffic was sent from a separate host on the same wired network, switch and VLAN as the VM. To obtain the most realistic results, this test should ideally be run in an environment that mirrors production. The following ten tests were run three times each:

- Not Run – Normal application traffic sent to the application. Run without Contrast Agent starting the application.
- Off - Normal application traffic with Contrast Protect in Off mode.
- Monitor - Normal application traffic with Contrast Protect in Monitor mode.
- Block - Normal application traffic with Contrast Protect in Block mode.
- Block at Perimeter (Block(P)) – Normal application traffic with Contrast Protect in Block at Perimeter mode (for those rules that have a Block at Perimeter mode).
- Not Run (Attack) – Attack traffic sent to the application. Run without Contrast Agent starting the application.
- Off (Attack) - Attack traffic with Contrast Protect in Off mode.
- Monitor (Attack) - Attack traffic with Contrast Protect in Monitor mode.
- Block (Attack) - Attack traffic with Contrast Protect in Block mode.
- Block at Perimeter (Block(P)) (Attack) – Attack traffic with Contrast Protect in Block at Perimeter mode (for those rules that have a Block at Perimeter mode).

The VM hosting the Goat applications was restarted consistently before and after each test run to reset the environment. New Relic Application Performance Monitoring (APM) software was used to gather the following metrics: per request times, memory usage and CPU utilization.

The per-request transaction time value was calculated as the geometric mean of the transaction times of the three runs (x_1 , x_2 , x_3) for each of the test cases. The geometric mean is calculated by multiplying the three runs together and taking the cube root: $GM = \sqrt[3]{(x_1 * x_2 * x_3)}$.

A geometric mean was selected as the most accurate statistical method to analyze the data because the resultant transaction time value is closest to the central value and is not skewed to the higher or lower values in the data set (Suhas, D. 2017, May).

NodeGoat testing was conducted using version 1.35.0 of the Contrast Agent. The NodeGoat VM was running Ubuntu 14.04.5 LTS 64-bit on VMware Fusion 8.5.10 with four processor cores and 6144 MB of RAM. The underlying hardware was a Mac Pro (Early 2008) with 2 x 2.8 GHz Quad-Core Intel Xeon processors, 16 GB 800 MHz DDRD FB-DIMM memory on OS X Version 10.11.6.

WebGoat testing was conducted using build 3.5.8.5149. The WebGoat VM was running Ubuntu 18.04.1 LTS 64-bit on VMware Fusion 11.0.2 with four processor cores and 6144 MB of RAM. The underlying hardware was a MacBook Pro (15-inch, 2017) with a 2.9 GHz Intel Core i7 processor, 16 GB 2133 MHz LPDDR3 memory on OS X Version 10.14.2.

Performance measurement visualizations can be found in the supplemental material GitHub repository: https://github.com/alexanderfry/rasp-research-paper/blob/master/PERFORMANCE_MEASUREMENT.md.

4.1. NodeGoat Normal Traffic

Each row of Figure 21 below displays the peak transaction time value for the area of performance that was measured: Node.js, MongoDB, Web external and NodeGoat overall application response. The Web external response time was not recorded (N/R) in the New Relic Portal for the test cases when the Contrast Agent was enabled; the reason for this is unknown. Web external is defined as:

the portion of time spent in transactions to external applications from within the code of the application you are monitoring. That can be a call to a 3rd party

company or it could be a call to another microservice within the company. It serves to help you understand how much performance impact there is for code executing outside the application you are measuring (Carpenter, S. ,2017, February).

A review of the documentation from both New Relic and Contrast Security did not provide any additional information concerning the troubleshooting of this issue. However, a decision was made that Node.js response time and overall NodeGoat response time were the most important performance metrics, so no additional troubleshooting was pursued.

	<i>Not Run</i>	<i>Off</i>	<i>Monitor</i>	<i>Block</i>	<i>Block(P)</i>
<i>Node.js</i>	5.68	48.24	170.61	140.00	137.33
<i>MongoDB</i>	1.61	2.16	2.24	1.21	1.24
<i>NodeGoat</i>	31.96	49.53	171.93	140.33	137.33
<i>Web External</i>	22.41	N/R	N/R	N/R	N/R

Figure 21. NodeGoat Normal Traffic, Response Time Per Request in Milliseconds (ms)

There was a 749 percent increase in Node.js response time from when NodeGoat was run without Contrast and when the Contrast Agent was enabled in Off mode. The performance impact on Node.js more than doubled in the Monitor, Block, and Block(P) modes. Of all the active modes: Monitoring, Block and Block(P); the Monitoring mode created the most overhead on the response times across the application. Each row of Figure 22 below shows the percentage increase in response times in Node.js and the NodeGoat application for each of the Contrast Protect Modes compared to when Contrast was not run.

	<i>Off</i>	<i>Monitor</i>	<i>Block</i>	<i>Block(P)</i>
<i>Node.js</i>	749	2,904	2,365	2,318
<i>NodeGoat</i>	55	438	339	330

Figure 22. NodeGoat Normal Traffic, Response Time Percentage Increase

The Node.js VM V8 Heap (used) memory ranged from 19.4 to 88 MB. The memory usage with Contrast was approximately 68.6 MB more than without Contrast, an increase of 354 percent. All modes including Off appeared to use approximately the same amount of memory.

The peak Node.js VM CPU utilization with Contrast was approximately 10 percent more than without Contrast. All modes including Off appeared to use approximately the same processing power.

In summary, when normal traffic was run through NodeGoat with the Contrast Agent enabled, it experienced a significant performance impact. The smallest impact measured was a Node.js response time increase of 749 percent and an overall application response time increase of 55 percent. The memory usage increased 354 percent and the CPU usage increased 10 percent.

4.2. NodeGoat Attack Traffic

Each row of Figure 23 below displays the peak transaction time value for the area of performance being measured: Node.js, MongoDB, Web external and overall NodeGoat application response. As in the NodeGoat Normal Traffic performance measurement, the Web external response time was not recorded (N/R) in the New Relic Portal for the test cases when Contrast Agent was enabled - the cause of this is unknown.

	<i>Not Run</i>	<i>Off</i>	<i>Monitor</i>	<i>Block</i>	<i>Block(P)</i>
<i>Node.js</i>	8.54	33.18	180.12	177.42	159.75
<i>MongoDB</i>	2.40	3.10	4.86	1.33	1.32
<i>NodeGoat</i>	19.81	35.33	182.62	177.69	160.04
<i>Web External</i>	11.34	N/R	N/R	N/R	N/R

Figure 23. NodeGoat Attack Traffic, Per Request Times in Milliseconds (ms)

There was a 288 percent increase in Node.js response time from when NodeGoat was run without Contrast and when the Contrast Agent was enabled in Off mode. The performance impact on Node.js more than doubled in the Monitor, Block, and Block(P) modes. Of all the active modes: Monitoring, Block and Block(P); the Monitoring mode created the most overhead on the response times across the application. Each row of Figure 24 below shows the percentage increase in response times in Node.js and the NodeGoat application for each of the Contrast Protect Modes compared to when Contrast was not run.

	<i>Off</i>	<i>Monitor</i>	<i>Block</i>	<i>Block(P)</i>
<i>Node.js</i>	288	2,009	1,978	1,771
<i>NodeGoat</i>	78	822	797	708

Figure 24. NodeGoat Attack Traffic, Response Time Percentage Increase

The Node.js VM V8 Heap (used) memory ranged from 27.2 to 97.2 MB. The memory usage with Contrast was approximately 70 MB more than without Contrast, an increase of 257 percent. All Modes including Off appeared to use the same amount of memory.

The peak Node.js VM CPU utilization with Contrast was approximately 10 percent more than without Contrast. All Modes including Off appeared to use approximately the same processing power.

In summary, when attack traffic was run through NodeGoat with the Contrast Agent enabled, it experienced a significant performance impact. The smallest impact measured was a Node.js response time increase of 288 percent and an overall application response time increase of 78 percent. The memory usage increased 257 percent and the CPU usage increased 10 percent.

4.3. WebGoat Normal Traffic

Each row of Figure 25 below displays the peak transaction time value for the area of performance being measured: JVM, HSQLDB, and overall WebGoat application response. As in the NodeGoat performance measurement, the Web external response time was not recorded (N/R) in the New Relic Portal for any of the test cases - the cause for this is unknown.

	<i>Not Run</i>	<i>Off</i>	<i>Monitor</i>	<i>Block</i>	<i>Block(P)</i>
<i>JVM</i>	16.40	25.77	24.72	24.60	29.49
<i>HSQLDB</i>	2.48	4.14	3.68	3.81	4.00
<i>WebGoat</i>	18.90	29.38	28.39	28.40	33.27

Figure 25. WebGoat Normal Traffic, Per Request Times in Milliseconds (ms)

There was a 57 percent increase in the JVM response time from when WebGoat was run without Contrast and when the Contrast Agent was enabled in Off mode. Of all the active modes: Monitoring, Block and Block(P); the Block(P) mode created the most overhead on the response times across the application. Each row of Figure 26 below shows the percentage increase in response times in the JVM and the WebGoat application for each of the Contrast Protect Modes compared to when Contrast was not run.

	<i>Off</i>	<i>Monitor</i>	<i>Block</i>	<i>Block(P)</i>
<i>JVM</i>	57	51	50	80
<i>WebGoat</i>	55	50	50	76

Figure 26. WebGoat Normal Traffic, Response Time Percentage Increase

The peak memory usage with Contrast was approximately 186 MB more than without Contrast. All Modes including Off appeared to use approximately the same amount of memory.

The peak CPU usage with Contrast was approximately 44 percent more than without Contrast.

In summary, when normal traffic was run through WebGoat with the Contrast Agent enabled, it experienced a significant performance impact. The smallest impact measured was a JVM response time increase of 50 percent and an overall application response time increase of 50 percent. The memory usage increased 28 percent and the CPU usage increased 44 percent.

4.4. WebGoat Attack Traffic

Each row of Figure 27 below displays the peak transaction time value for the area of performance being measured: JVM, HSQLDB, Web external and overall WebGoat application response. The Web external response time was recorded in the New Relic Portal for all test cases - it is unknown as to why this was different for WebGoat Attack Traffic compared to WebGoat Normal Traffic or the NodeGoat Attack and Normal Traffic tests.

	<i>Not Run</i>	<i>Off</i>	<i>Monitor</i>	<i>Block</i>	<i>Block(P)</i>
<i>JVM</i>	20.50	29.96	33.33	31.59	31.55
<i>HSQLDB</i>	3.08	4.57	4.92	4.72	4.75
<i>WebGoat</i>	25.52	36.11	38.90	37.88	37.76
<i>Web External</i>	1.78	1.57	1.84	1.42	1.42

Figure 27. WebGoat Attack Traffic, Per Request Times in Milliseconds (ms)

There was a 46 percent increase in the JVM response time from when WebGoat was run without Contrast and when the Contrast Agent was enabled in Off mode. Of all the active modes: Monitoring, Block and Block(P); the Monitor mode created the most overhead on the response times across the application. Each row of Figure 28 below shows the percentage increase in response times in the JVM and the WebGoat application for each of the Contrast Protect Modes compared to when Contrast was not run.

	<i>Off</i>	<i>Monitor</i>	<i>Block</i>	<i>Block(P)</i>
<i>JVM</i>	46	63	54	54
<i>WebGoat</i>	42	52	48	48

Figure 28. WebGoat Attack Traffic, Response Time Percentage Increase

The peak memory usage with Contrast was approximately 123 MB more than without Contrast. All Modes including Off appeared to use approximately the same amount of memory.

The peak CPU usage with Contrast was approximately 44 percent more than without Contrast.

In summary, when attack traffic was run through WebGoat with the Contrast Agent enabled, it experienced a significant performance impact. The smallest impact measured was a JVM response time increase of 46 percent and an overall application response time increase of 42 percent. The memory usage increased 18 percent and the CPU usage increased 44 percent.

5. Evaluation Procedure

Organizations that want to evaluate RASP solutions may find it helpful to follow a procedure based on the research in this paper. The following procedure helps an organization evaluate a RASP solution against business and performance requirements.

First, define the business uses cases that the RASP solution should solve. For example, the RASP solution should provide security visibility and support logging and

notification of attacks in production. It should block all attacks for the applications which are known to be vulnerable in production. And it should block all attacks for unknown vulnerabilities for which the RASP solution has rules.

Next, define the performance requirements that the RASP solution should meet. For example, the RASP solution should not increase per request times by more than 10%.

Then, select applications from the portfolio that should be protected with a RASP solution. Choose a subset of these applications as candidates for the trial, ensuring that all technology stacks are represented, e.g., Node.js, Java, .NET. Consider including known vulnerable applications for comparison.

Select RASP vendors that support the technology stacks. Then, contact the RASP vendors to explore a trial, choosing vendors that best support the trial requirements including the expected timeframe for testing.

Define success criteria and rank the criteria in order of importance. The following are example criteria: provides accurate attack detection; reduces risk with increased attack visibility; provides comprehensive attack results; ease of use; saves time with faster delivery of attack results; and, meets performance requirements.

Create a lab environment that mirrors production as closely as possible. Integrate the selected applications with the RASP solution and configure the RASP solution logging at the highest level of verbosity.

Next, create scripts or macros to automate normal and attack traffic against the selected applications. Ensure that the attacks exploit known vulnerabilities. If no known vulnerabilities exist, develop vulnerabilities and exploits. Run and time the scripts and macros to ensure that the attacks are repeatable and consistent.

Run the attacks against the applications in each of the different modes, e.g., Monitor, Block, Block(P). Monitor application logs and RASP solution logs for errors and other output. Compare the effectiveness of the RASP solution against the attacks in each of the different modes.

Measure the performance of the applications. First, turn off verbose logging for the RASP solution and applications to better mirror a production operating environment.

Run both normal traffic and attack traffic automated tests against the application. Run the tests without the RASP solution integrated to create a baseline. Run the tests in each of the different RASP modes, e.g., Off, Monitor, Block, Block(P). Then, compare the performance of the RASP solution in each of the different modes.

If the RASP solution uses a central application instance to collect vulnerability data, consider where this will be hosted. If it will be hosted on-premises instead of using the vendor's shared instance, estimate the costs that an on-premise environment will add to the total cost of ownership.

6. Areas for Further Research

There are other areas of RASP technology that deserve further research. Two of these are Evasion; and the Complementary Use of RASP with Web Application Firewall (WAF).

6.1. Evasion

This paper explored encoding and escaping XSS attacks to evade detection. There are other types of attacks with common evasion techniques that could be explored in greater detail. For example, SQL Injection attacks can be encoded to bypass `magic_quotes()` as well as WAFs (Netsparker Ltd., 2019). In addition, research could build off of the automated attacks run against the vulnerable web applications to create a test harness that varies the attacks using different evasion techniques. The test harness could run against the vulnerable web applications integrated with different RASP solutions. The industry already conducts standardized testing using similar approaches for classes of products such as Next Generation Firewalls (NSS Labs, 2017, December 7).

6.2. The Complementary Use of RASP with WAF

A number of organizations have existing investments in WAF technology. A WAF could be used with a RASP solution to provide greater attack intelligence and WAF enhancement. Contrast Security refers to this as a targeted defense. In this model, a cloud-based WAF would be used as a perimeter device and as the first line of defense

against application attacks. During an attack, a WAF alert would put the attack on the SOC's radar and inform the SOC if the WAF blocked the attack or took other actions. The RASP solution would send application-specific information to the SIEM such as source code impacted, data flow, stack trace, backend connections, libraries and frameworks, configuration and potential vulnerabilities. The SIEM would correlate the information from both the WAF and RASP using the common HTTP Request.

The combined data from both devices would provide greater context and intelligence about the attack and attacker. The SOC would be more informed and could more accurately flag an attack as something worth investigating. They would also be able to provide more specific guidance to application teams to fix an exploitable vulnerability (Contrast Security, 2018, March 20).

7. Conclusion

The majority of breaches are caused by web application attacks. Organizations are building, testing, and releasing applications with greater frequency, with gaps in test coverage resulting in vulnerabilities being pushed to production. Many custom-coded applications consist of more open source third-party code than first-party code, exposing organizations to open source risk. Zero-day vulnerabilities are discovered frequently in legacy, COTS and custom-coded applications.

With automated exploit scanning being conducted so quickly after the discovery of vulnerabilities, there is little time to apply a patch even if it is available. However, RASP solutions have the potential to block unknown attacks with no virtual patching, definition, or signature updates required. In the event a RASP solution would not block attacks against a specific vulnerability, the RASP vendor could quickly write a new rule to block exploitation and make the rule available to its customers. RASP has the potential to provide greater protection for production applications than existing solutions.

RASP is still not a perfect security solution. It focuses on common application security weaknesses and does not replace a human being for the discovery of business logic flaws. In addition, it takes effort by an organization to evaluate RASP offerings to ensure they meet business and performance requirements. However, RASP solutions are

evolving and have become more comprehensive in the types of weaknesses they protect against and support more programming languages and frameworks. Finally, there is great promise in RASP solutions complementing existing investments in WAF to provide greater attack intelligence and WAF enhancement.

References

- Adrian Lane. (2016, May 17). *Understanding and Selecting RASP: Technology Overview*. Retrieved from <https://securosis.com/blog/understanding-and-selecting-rasp-technology-overview>
- Amazon Web Services (AWS). (2019). What is DevOps?. Retrieved January 5, from <https://aws.amazon.com/devops/what-is-devops/>
- Andreesen, M. (2011, August 20). Why Software Is Eating the World. *The Wall Street Journal*[New York]. Retrieved from <https://www.wsj.com/articles/SB10001424053111903480904576512250915629460>
- Asadoorian, P. (2016). Getting a Grasp on RASP. Retrieved from IANS website: <https://www.iansresearch.com/insights/reports/getting-a-grasp-on-rasp>
- Bird, J. (2017). 2017 State of Application Security: Balancing Speed and Risk. Retrieved from SANS Institute website: <https://www.sans.org/reading-room/whitepapers/analyst/2017-state-application-security-balancing-speed-risk-38100>
- Carpenter, S. (2017, February). What does "Web external" metric means? [Web log post]. Retrieved from <https://discuss.newrelic.com/t/what-does-web-external-metric-means/45743/3>
- Cisar, P., & Cisar, S. M. (2016). The framework of runtime application self-protection technology. 2016 IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI). doi:10.1109/cinti.2016.7846383
- Contrast Security. (n.d.). Contrast Enables DevOp Teams to Deliver Security-as-Code. Retrieved January 1, 2019, from <https://www.contrastsecurity.com/devops>
- Contrast Security. (2019). Administration | Contrast Open Docs. Retrieved January 5, from <https://docs.contrastsecurity.com/admin-orgsettings.html#org-notify>
- Contrast Security. (2018, March 20). Get the Most Out of Your WAF Investment | Technical Brief. Retrieved January 1, 2019, from <https://www.contrastsecurity.com/get-the-most-out-of-your-waf-investment>

Contrast Security. (2017, July 21). *Whitepaper / State of Application Security: Libraries*. Retrieved from <https://www.contrastsecurity.com/state-of-application-security-libraries>

Contrast Security. (2017). RASP Technical Brief. Retrieved from Contrast Security, Inc. website: <https://www.contrastsecurity.com/rasptechbrief>

Gartner, Inc. (2019). *Web Application Firewalls Reviews*. Retrieved from <https://www.gartner.com/reviews/market/web-application-firewalls/vendors>

Gartner. (2012, November 4). Runtime Application Self-Protection (RASP) - Gartner IT Glossary. Retrieved January 1, 2019, from <https://www.gartner.com/it-glossary/runtime-application-self-protection-rasp>

Kim, G. (2012, August 22). The Three Ways: The Principles Underpinning DevOps - IT Revolution. Retrieved from <https://itrevolution.com/the-three-ways-principles-underpinning-devops/>

Netsparker Ltd. (2019). SQL Injection Cheat Sheet. Retrieved January 1, 2019, from <https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/>

New Relic. (2019). New Relic: Application Performance Monitoring and Management. Retrieved from <https://newrelic.com/products/application-monitoring>

NSS Labs. (2017, December 7). Next Generation Firewall (NGFW) Test Methodology. Retrieved from https://www.nsslabs.com/default/assets/example-reports/ngfw/NSS_Labs_Next_Generation_Firewall_Methodology_v8_0.pdf

OWASP. (2016). *Tutorial - OWASP NodeGoat Project*. Retrieved November 1, 2018, from <http://nodegoat.herokuapp.com/tutorial>

OWASP. (2017, June 26). *Server-Side Request Forgery - OWASP*. Retrieved from https://www.owasp.org/index.php/Server_Side_Request_Forgery

OWASP. (2017). OWASP Top 10 2017. Retrieved from https://www.owasp.org/index.php/Top_10-2017_Top_10

PortSwigger. (2018). *Bypassing Signature-Based XSS Filters*. Retrieved from <https://support.portswigger.net/customer/portal/articles/2590820-bypassing-signature-based-xss-filters-modifying-script-code>

Prevoty. (2017, March 13). Prevoty Automatically Protects Against the Latest Struts 2 Vulnerability and Attacks Targeting Remote Code Injection Vulnerabilities.

- Retrieved from <http://www.marketwired.com/press-release/prevoty-automatically-protects-against-latest-struts-2-vulnerability-attacks-targeting-2202301.htm>
- Prevoty. (2017, September). *AppSec in an Open Source World 101*. Retrieved from https://s3-us-west-2.amazonaws.com/prevotyexternaldocs/Whitepapers/AppSec_in_an_Open_Source_World_101.pdf
- Prevoty. (2018, May 17). Prevoty Technical Overview. Retrieved from Prevoty, Inc. website: <https://www.prevoty.com/prevoty-technical-overview>
- Price, E. (2017, September 12). *How to shed the technical debt of legacy code*. Retrieved from <https://www.devbridge.com/articles/shed-the-technical-debt-of-legacy-code/>
- Suhas, D. (2017, May 18). Using the Right Mean for Meaningful Performance Analysis | Data Analysis [Blog post]. Retrieved from <http://blog.catchpoint.com/2017/05/18/using-mean-performance-analysis/>
- Tirosh, A., Zumerle, D., & Horvath, M. (2018). Magic Quadrant for Application Security Testing. Retrieved from Gartner website: <https://www.gartner.com/doc/reprints?id=1-4TFRCQV&ct=180319&st=sb>
- Ullrich, J. (2016). 2016 State of Application Security: Skills, Configurations and Components. Retrieved from SANS Institute website: <https://www.sans.org/reading-room/whitepapers/analyst/2016-state-application-security-skills-configurations-components-36917>
- Verizon. (2018). Data Breach Investigations Report (11th Edition). Retrieved from Verizon website: https://www.verizonenterprise.com/resources/reports/rp_DBIR_2018_Report_en_xg.pdf
- Waratek. (2018). Application Security Using Runtime Protection. Retrieved from Waratek Ltd. website: <https://cdn.aws.waratek.com/v2/wp-content/uploads/2018/02/WP-RASP-Intro-20180206.pdf>
- Williams, J. (2015). Protection from the Inside: Application Security Methodologies