

Getting Started with IAST and RASP using Contrast Security

Overview

This lab will provide a basic introduction to Interactive Application Security Testing (IAST) and Runtime Application Self-Protection (RASP) using the Contrast Security Community Edition, showing how the platform works against known vulnerable locations in WebGoat. By observing how Contrast operates against known vulnerable locations with known payloads, users can gain the experience that will help them use Contrast effectively against other applications.

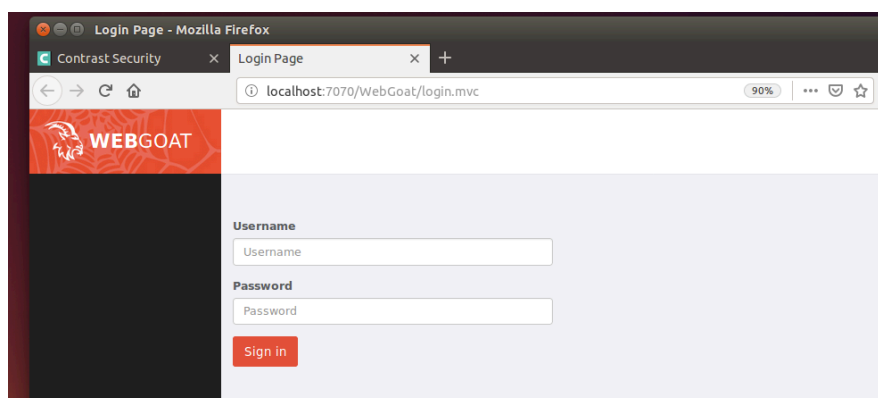
This lab will walk through a few basic use cases using the Contrast Platform:

- Contrast Assess – IAST which continuously discovers vulnerabilities as you write and test your applications
- Contrast Protect – RASP which continuously monitors for attacks against your applications, and can block them
- Contrast OSS – which continuously monitors your usage of Open Source Software, and assesses your risks due to OSS usage

Note: This lab uses the Contrast Security Community Edition, which provides one free license for use against one application.

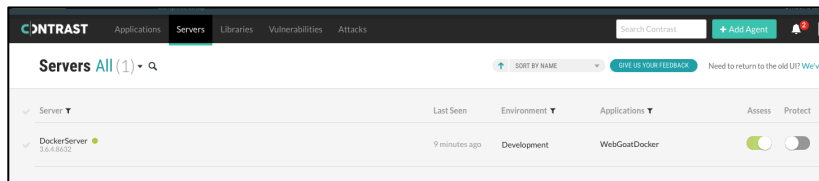
Let's Get Started

You should now be seeing the Contrast UI and WebGoat in adjacent browser tabs or windows:



Log in to your Contrast Community Edition instance, and click the **Servers** page. Within the Contrast UI, you should see your **DockerServer** in the Servers area. Please raise your hand or type into the webinar chat window if you don't.

Note: if **Protect** is enabled, disable it for now by sliding its green slider switch – your server settings should look like this:



LAB 1: Discovering an XML External Entity (XXE) Vulnerability with IAST

The first two exercises will show you how Contrast Assess uses IAST as a live, always on vulnerability discovery engine to detect vulnerabilities in applications through simple UI interactions. First, log in to WebGoat using the guest or admin accounts show in the WebGoat UI.

Let's see how Contrast detects XXE vulnerabilities. Open the **Parameter Tampering** section in WebGoat then click on **XML External Entity (XXE)**. Type something into the **From:** box. (Notice that it doesn't matter what you type - it need not be an attack or anything special – it's a simple normal interaction in the WebGoat UI). Now check the **Vulnerabilities** page in the Contrast UI again. Drill into the **XML External Entity Injection (XXE) from Request Body** vulnerability and examine the **Overview**, **Details**, **HTTP Info**, **How to Fix**, **Notes**, and **Discussion** tabs.

Note: Vulnerabilities are persisted objects in Contrast, but they can be safely deleted (or marked as Not a Problem, Remediated, Fixed, etc). Don't worry if you delete a Vulnerability – you can re-create it by restarting the WebGoat lesson and performing the same UI interaction that surfaced it before. Try it! And think about how this will help you test code fixes for vulnerabilities in your own applications. 😊

Lab 1: Discovering SQL Injection Vulnerability with IAST

In the WebGoat navigation area, open **Injection Flaws**, then click **String SQL Injection**. In the **Enter Your Last Name** field, enter the word **Smith**. Notice once again that this is **not** an attack or anything special – it's a simple normal interaction in the WebGoat UI:

WEBGOAT

String SQL Injection

[Show Source](#) [Show Solution](#) [Show Plan](#) [Show Hints](#) [Restart Lesson](#)

SQL injection attacks represent a serious threat to any database-driven site. The methods behind an attack are easy to learn and the damage caused can range from considerable to complete system compromise. Despite these risks, an incredible number of systems on the internet are susceptible to this form of attack.

Not only is it a threat easily instigated, it is also a threat that, with a little common-sense and forethought, can easily be prevented.

It is always good practice to sanitize all input data, especially data that will used in OS command, scripts, and database queries, even if the threat of SQL injection has been prevented in some other manner.

General Goal(s):

The form below allows a user to view their credit card numbers. Try to inject an SQL string that results in all the credit card numbers being displayed. Try the user name of 'Smith'.

Enter your last name:

```
SELECT * FROM user_data WHERE last_name = 'Smith'
```

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0

Now look in the **Vulnerabilities** area of the Contrast UI. What do you see? Do you see more than one vulnerability? If so, why? Drill into the SQL Injection vulnerability details to see how Contrast coaches developers to write safer code at the earliest stages of the SDLC.

Takeaway: IAST discovers vulnerable *routes*, 24x7, in the background, as you interact with your applications – that means manual UI usage, Selenium scripts, Postman, JMeter/Blazemeter, QA Regression, Neoload, Puppeteer, etc – IAST makes every interaction double as a security test. Contrast uses this to help developers write and check in safer code during their normal work processes.

LAB 3: Performing a SQL Injection Attack, and then Blocking it with RASP

Return to the **Injection Flaws** -> **String SQL Injection** lesson in Webgoat. This time, try the following input: **Smith' or '1'='1** This will show all users:

Enter your last name:

SELECT * FROM user_data WHERE last_name = 'Smith' or '1'='1'




USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Joe	Snow	987654321	VISA		0
101	Joe	Snow	2234200065411	MC		0
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0
103	Jane	Plane	123456789	MC		0
103	Jane	Plane	333498703333	AMEX		0
10312	Jolly	Hershey	176896789	MC		0
10312	Jolly	Hershey	333300003333	AMEX		0
10323	Grumpy	youaretheweakestlink	673834489	MC		0
10323	Grumpy	youaretheweakestlink	33413003333	AMEX		0
15603	Peter	Sand	123609789	MC		0
15603	Peter	Sand	338893453333	AMEX		0
15613	Joesph	Something	33843453533	AMEX		0

Congratulations! You've just successfully performed a SQL Injection attack, and ensured that Webgoat Inc. will make the front page of Securityboulevard.com tomorrow. 😊

Now we'll show how Contrast Security's **Protect** RASP can detect and block such an attack.

Enabling Contrast Protect

On the **servers** page, find your server via its green icon and ensure both sliders are green:

Servers All (1) 🔍				↑ SORT BY NAME
✓ Server ▼	Last Seen	Environment ▼	Applications ▼	Assess Protect
✓ DockerServer  3.6.7-10617	1 minute ago	Development	WebGoatDocker	 

In the Applications area, make sure that **Protect** is turned on:

WebGoatDocker •
URL: / | Language: Java | Importance: Medium

Overview Vulnerabilities Libraries Activity Live Policy

D

Custom Code Score	35
Library Score (Vulnerability-only)	87
Overall Score	61

DEVELOPMENT

Protect ON

Assess ON

Servers 1

20 Vulnerabilities ▾ By Severity ▾

15 1 4

Vulnerability Trend ? New - Last 7 Days ▾

In the Contrast UI, find the **Policy** tab for your application, and click **Protect**:

WebGoatDocker •
URL: /WebGoat | Language: Java | Importance: Medium

Overview Vulnerabilities Libraries Activity Live Policy

F

Custom Code Score	84
Library Score (Vulnerability-only)	84
Overall Score	84

84 Library

DEVELOPMENT

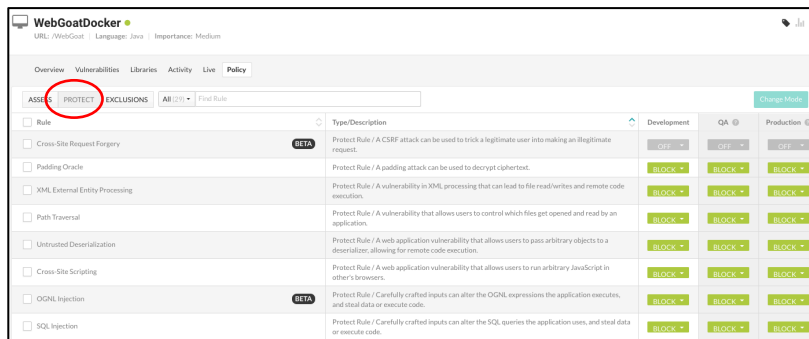
Protect ON

Assess ON

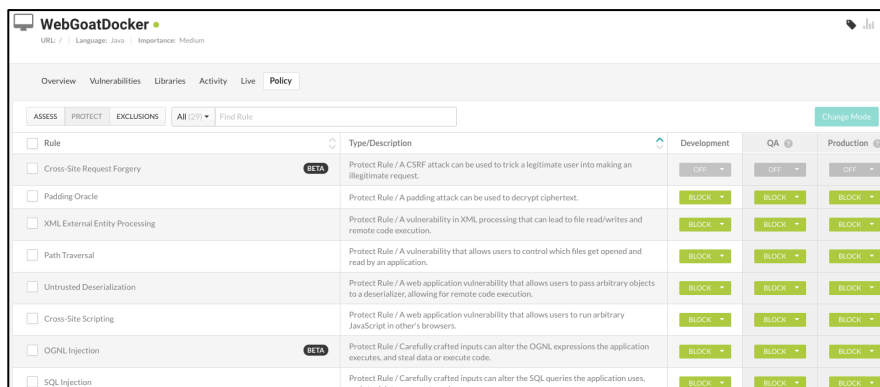
QA

No Servers D

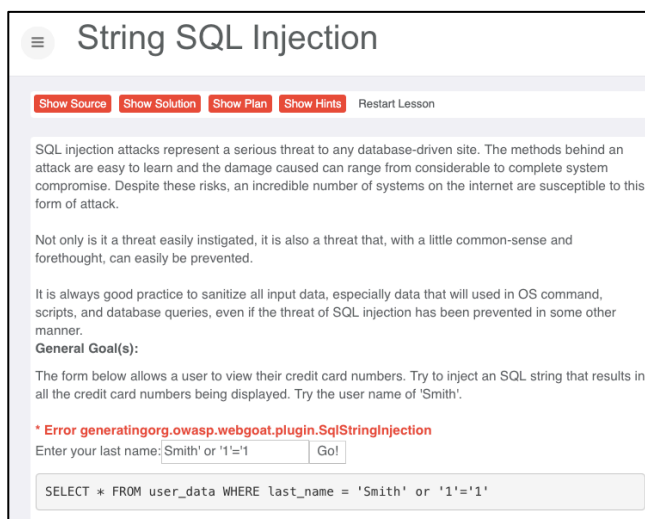
Start gathering app



Find the SQL Injection Rule and set it to **Block** if it isn't already:



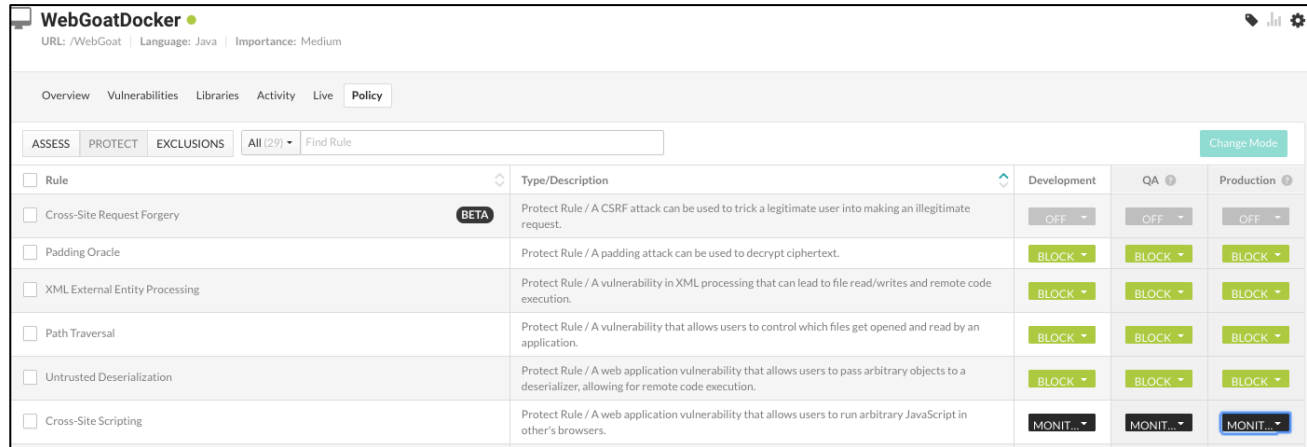
Back in WebGoat's **Injection Flaws** -> **String SQL Injection**, try using just the name **Smith** again. Did this work as expected? Why? *The normal usage works because the SQL grammar did not change.* Now re-start the Webgoat lesson, and attempt the same SQL Injection attack as before. The attack no longer succeeds:



Takeaway: RASP forms a defense in depth layer inside the application which follows potential attacks along routes in the application and uses the collected diagnostics to minimize False Positives. RASP prevents exploits without interfering with non-malicious application usage, and provides code level attack forensics.

Lab 4: RASP Example: Blocking Cross Site Scripting

Cross Site Scripting enables attackers to add arbitrary JavaScript code to other users' browsing sessions, for example to steal cookie data or perform other XHR requests. To understand how Contrast protects against Cross Site Scripting, first set the **Protect** Policy for **Cross-Site Scripting** to **Monitor**:

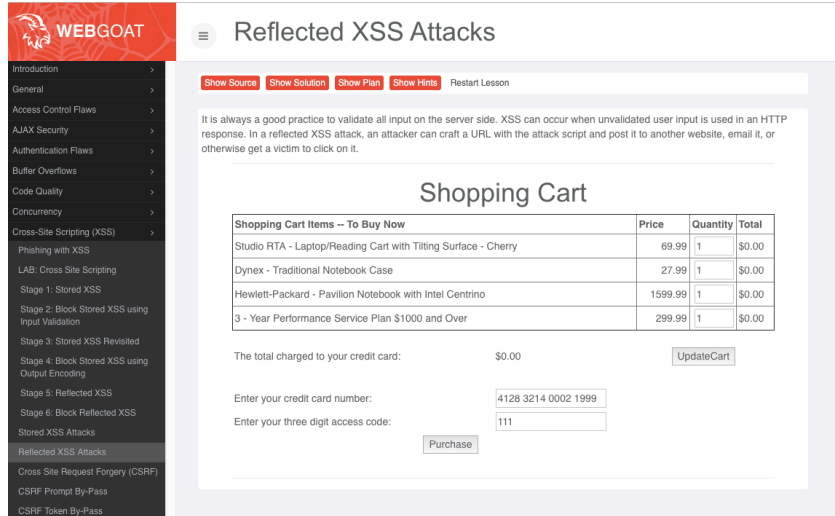


The screenshot shows the 'WebGoatDocker' interface with the 'Policy' tab selected. It displays a table of RASP rules with columns for Rule, Type/Description, Development, QA, and Production. The 'Cross-Site Scripting' rule is highlighted with a blue border, and its policy is set to 'MONIT...' in all three environments.

Rule	Type/Description	Development	QA	Production
<input type="checkbox"/> Rule				
<input type="checkbox"/> Cross-Site Request Forgery BETA	Protect Rule / A CSRF attack can be used to trick a legitimate user into making an illegitimate request.	OFF	OFF	OFF
<input type="checkbox"/> Padding Oracle	Protect Rule / A padding attack can be used to decrypt ciphertext.	BLOCK	BLOCK	BLOCK
<input type="checkbox"/> XML External Entity Processing	Protect Rule / A vulnerability in XML processing that can lead to file read/writes and remote code execution.	BLOCK	BLOCK	BLOCK
<input type="checkbox"/> Path Traversal	Protect Rule / A vulnerability that allows users to control which files get opened and read by an application.	BLOCK	BLOCK	BLOCK
<input type="checkbox"/> Untrusted Deserialization	Protect Rule / A web application vulnerability that allows users to pass arbitrary objects to a deserializer, allowing for remote code execution.	BLOCK	BLOCK	BLOCK
<input type="checkbox"/> Cross-Site Scripting	Protect Rule / A web application vulnerability that allows users to run arbitrary JavaScript in other's browsers.	MONIT...	MONIT...	MONIT...

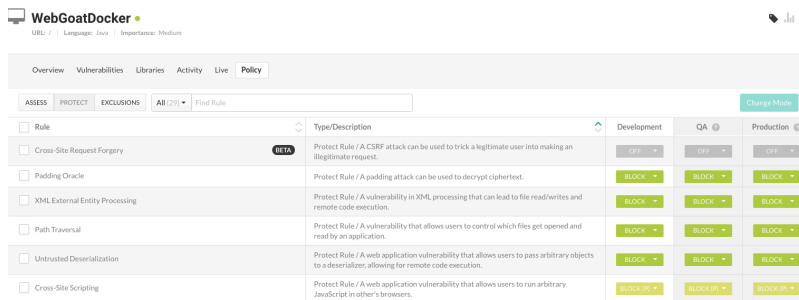
This will allow attacks to proceed but would of course alert you through any notification systems you've connected to Contrast (e.g. email, Slack, MS Teams, Jira, PagerDuty, ServiceNow, etc). Now

open the **Cross-Site Scripting** section in WebGoat then click on **Reflected XSS Attacks**:



In the **three digit access code** field, add an XSS payload such as: `<script>alert("XSS Test")</script>`. Click **Purchase** and the browser will display the alert in a popup. Try running anything else between the script tags.

In the Contrast UI, set the **Protect** Policy for Cross-Site Scripting to **Block**.



Try running the attack payload again. What happens?

Contrast also provides a **Block at Perimeter** setting. While **Block** prevents runtime exploits by generating exceptions in your application's code, **Block at Perimeter** will prevent XSS attacks before they enter the application. This will block reflected and persistent XSS, where the payload may pass to another area that technically is not exploitable but will exploit users later (an example of this is saving the XSS attack to a persistent database).

Intelligent Reduction of False Positives

Contrast operates on something like a Hippocratic oath: first, do no harm. Developers may often store non-harmful HTML code in areas that do not perform attacks. Contrast does a grammatical analysis of content, letting format tags through while blocking script functions. For example, the following non-harmful attack will pass through the credit card field as normal:

Blah

Therefore it may be possible for attackers to mess up an HTML display but not in a harmful way.

Contrast Protect's RASP Modes

Protect provides different operation modes that enable you to handle attacks in different ways. Very often, users will start with **Protect** in **Monitor** mode until they become comfortable with its attack detection capabilities. The operation modes for **Protect** are:

- Off – do nothing – do not monitor attacks or attempt to block them
- Monitor – watch for attacks and send notifications, without any blocking
- Block – Block an attack using the Contrast Agent, just before an exploit against the application's vulnerable data sink would have taken place, by raising an exception at runtime.
- Block at Perimeter – Block when a type of data is first detected coming in, regardless of whether it is headed for a vulnerable sink. This is most common for XSS type issues so that they do not pass through a sink that is not vulnerable to XSS and reflect back to the user.

Teams can monitor for vulnerabilities through the **Vulnerabilities** tab or watch attack attempts take place through the **Attacks** tab. Contrast attempts to avoid noisy wolf-crying, alerting only when elements are worth looking at.

Lab 5: Analyzing OSS Risk

Click on **Libraries** at the top of the Contrast UI. Click **Show Library Stats** at the top right. Which libraries would you update first?

Want More?

Integrations

In the Contrast UI dropdown under your name, click **Organization Settings**, then in the left nav area, click **Integrations**. Also check this URL: <https://contrast-security-oss.github.io/>

Product Documentation: <https://docs.contrastsecurity.com>

Blogs: <https://www.contrastsecurity.com/security-influencers>