OWASP
The Open Web Application Security Project

Securing WebGoat Using ModSecurity

beta

Securing WebGoat using ModSecurity

Summer of Code 2008

OWASP Beta Level

Version 1.0

November 2008

Amendment History				
Change Number	Revision Description	Paragraph Affected	Revision Number	Date
1	Beta at 50% project completion	All	0.1	Sep 2008
2	Near 100% project completion	All	0.2	Oct 2008
3	Project Completion (Beta Level)	All	0.3	Oct 2008
4	Integrate reviewers' comments and corrections	All	0.5	Nov 2008
5	Add new section that contains Concurrent File Access and Lua Security in ModSecurity	4.4 Unfinished Business	0.7	Nov 2008
	Align this Word doc with wiki	All		
6	Complete all project work	All	1.0	Nov 2008

Table of Contents

Securing WebGoat using ModSecurity

1. Introduction

1.1 Background

ModSecurity is an open source web application firewall that can work either embedded in an Apache web server or as a reverse proxy. The new features in version 2.0 and version 2.5 (released in February 2008) allow for a highly configurable capability that can address vulnerabilities (e.g. discovered during black-box penetration testing) on a per-application basis. ModSecurity provides for free a broad set of generic Core Rulesets that cover areas such as protocol compliance, malicious client software detection, XML protection, error detection, and generic attack detection ("Detect application level attacks such as described in the OWASP top 10"). However, the Core Set rule documentation (see README in modsecurity-core-rules_2.5-1.6.0.tar.gz) cautions that since attackers may examine the freely-available core rules to get around them, some core rules should be viewed more as a "nuisance reduction" mechanism instead of a security mechanism.

The lessons in WebGoat 5.2 detail over 30 different types of attacks on the WebGoat application (see the WebGoat v5 User & Install Guide).

1.2 Purpose

The purpose of this project is to create custom ModSecurity rulesets that, in addition to the Core Set, will protect WebGoat 5.2 Standard Release from as many of its vulnerabilities as possible (the goal is 90%) without changing one line of source code. To ensure that it will be a complete 'no touch' on WebGoat and its environment, ModSecurity 2.5.5 will be configured on Apache server as a remote proxy server.

For those vulnerabilities that cannot be prevented (partially or not at all), I will document my efforts in attempting to protect them. Business logic vulnerabilities will be particularly challenging to solve.

The opportunity, challenges, issues or need this project addresses:

- Provides application-level protection for those web applications that cannot be touched
- New custom rulesets can be added as new attack types are discovered
- This solution is programming language and platform agnostic
- With outside help from consultants, this solution can be used by companies that have zero knowledge of software security
- A possible unintended side-effect: introduce software security awareness into an organization, which may lead to software security development lifecycle practices for future projects

- Common types of business logic vulnerabilities (this will be a challenge)

1.3 Tasks and deliverables

- Set up and test the development environment. The initial Reverse Proxy server OS will be Kubuntu 7.10.
- Identify WebGoat vulnerabilities and exploitation methods. Publish to wiki for review, feedback, and modification.
- Develop rulesets for 50% of the vulnerabilities, starting with the low-hanging fruit. Deliver user documentation. Publish progress to wiki as each vulnerability is addressed.
- Peer review, feedback, modification at 50% project completion (Milestone 1).
- Develop rulesets for the 2nd 50% of the vulnerabilities. Publish progress to wiki as each vulnerability is addressed.
- Test final rulesets and Modsecurity Reverse Proxy on two other Linux distros.
- Produce final rulesets and documentation.
- Peer review, feedback, modification, deliver final product (Project completion).

1.4 Project member comments at 100%

The project has a heavy Lua scripting influence simply because it was easier for me with my background being software development; the project's subtitle could almost be "ModSecurity for Software Developer Dummies" and the project probably entered into a realm where no one person has gone before.

One of my security hats is pentesting, and this project has put ModSecurity in my toolbox. Now instead of only "Modify the source code and fix the vulnerability" as a recommendation for particular vulnerabilities, I can also recommend "Or, deploy a WAF and write a custom ruleset to mitigate the vulnerability". This is particularly applicable to PCI-DSS compliance, which for application security contains a triumvirate of penetration testing, code review, and a Web Application Firewall. Now I have all three of those in my arsenal of solutions that I can provide to the customer.

Using ModSecurity's Lua scripting on the back end and its JavaScript injection functionality on the front end lends itself to mitigating almost any type of vulnerability, including business logic flaws. Of course, performance considerations must be taken into account and in high octane scenarios using these features might not be feasible.

Thanks to Ryan Barnett at Breach Security for turning the project almost into a contest; he presented a "traditional" ModSecurity solution to counter many of my Lua lesson solutions. You can compare them and decide which is easier to develop, test, and debug.

A thank you also goes out to Christian Folini, especially for (perhaps unknowingly) giving me moral support when it was needed.

Stephen Evans - 27 October 2008

1.5 Future development and long-term vision

The following are possibilities for future development:

1. Package Lua script functionality:

Several business logic flaws were solved using Lua scripting. Perhaps these examples can be expanded into more robust solutions, or as a project reviewer stated: "In some scenarios it may be advantageous to use Lua scripts to provide "packaged" functionality for some complex issues..."

2. Use ModSecurity to protect Hacme Bank:

A description: Hacme Bank (v2.0 Released 5/19/2006) is designed to teach application developers, programmers, architects and security professionals how to create secure software. Hacme Bank simulates a "real-world" webservices-enabled online banking application, which was built with a number of known and common vulnerabilities. This allows users to attempt real exploits against a web application and thus learn the specifics of the issue and how best to fix it. The web services exposed by Hacme Bank are used by our other testing applications including Hacme Books and Hacme Travel.

Hacme Bank is .NET-based and uses Microsoft IIS so ModSecurity would have to be deployed in a reverse proxy configuration.

3. Implement security event logging as a ModSecurity add-on:

The lack of security logging in an application can make it difficult to trace security breaches and violations of security policy. Get Ken van Wyk's take on this at 12m35s to 15m22s of Cigital's Silver Bullet podcast #30 (http://www.cigital.com/silverbullet/show-030/) with Gary McGraw. To roughly quote: "For incident response after an attack, you have monitoring data from the network components, from the firewalls, even from the web server; but once you start talking application servers and the application itself, the monitoring pretty much goes to zero."

The steps would be:
(1) Identify areas of an application that require security events.
(2) Write traditional ModSecurity rules for these events and write to the audit log file.
(3) Write Lua scripts that, if possible, modify existing log files written by other applications; or, that write to a new, standalone file in a different format that integrates with other log file formats that are used by other ecosystems.

Perhaps the OWASP ESAPI project can be referenced and the relevant portions of it be implemented for this.

4. Modifying the HTTP request:

Perhaps there is merit from a pure research standpoint to be able modify the HTTP request within ModSecurity then send it on to the application.

Work was started on it as part of the solution for Sublesson 12.1: Insecure Login - but not completed and it is not known whether this approach will work, although it should:

- Using a Lua script
  - Extract the HTTP request
  - Modify the HTTP request
  - Write a Perl script to disk with the modified HTTP request
- Execute the Perl script using a ModSecurity SecAction, which sends the new request to the server

The Perl script to date - which does not yet work - is:

```
#!/usr/bin/perl

use LWP::Debug qw(+);

use LWP::UserAgent;  # This will cover all of them!
use URI::URL;

$hdrs = new HTTP::Headers(Accept => 'text/plain',
                          User-Agent => 'MegaBrowser/1.0');

$hdrs = new HTTP::Headers(Host => '192.168.0.5',
User-Agent => 'Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
rv:1.8.1.16)',
Accept => 'text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5',
Accept-Language => 'en-us,en;q=0.5',
Accept-Charset => 'ISO-8859-1,utf-8;q=0.7,*;q=0.7',
Keep-Alive => '300',
Proxy-Connection => 'keep-alive',
Referer => 'http://192.168.0.5/WebGoat/attack?Screen=97&menu=800',
Cookie => 'JSESSIONID=E99C3EBA1803A5EB493C7DB181C4E46E',
Authorization => 'Basic Z3Vlc3Q6Z3Vlc3Q=',
Content-Type => 'application/x-www-form-urlencoded',
Content-Length => '46',
Content => [ QTY1 => '0',
             QTY2 => '0',
             QTY3 => '0',
             QTY4 => '0',
             SUBMIT => 'Update+Cart',
           ]);

$url = new
URI::URL('http://192.168.0.5/WebGoat/attack?Screen=97&menu=800');

$req = new HTTP::Request(POST, $url, $hdrs);
```

```
$ua = new LWP::UserAgent;

$resp = $ua->request($req);

if ($resp->is_success) {
        print $resp->content;}
else {
        print $resp->message;}
```

1.6 Contributors

Project team: Stephen Evans

1st reviewer: Ivan Ristic (Curriculum) & Breach Research Labs (Ryan Barnett, Brian Rectanus)

2nd reviewer: Christian Folini (Curriculum)

Special thanks go out to the reviewers for graciously volunteering their time for this project.

Other contributors:

- Dinis Cruz, who sculpted the original project proposal into something useful
- Paulo Coimbra, who exhibits extreme patience in his dealings with the project team
- Bruce Mayhew of the OWASP WebGoat project

2. WebGoat

2.1 Overview

From the WebGoat home page at OWASP: "WebGoat is a deliberately insecure J2EE web application maintained by OWASP designed to teach web application security lessons. In each lesson, users must demonstrate their understanding of a security issue by exploiting a real vulnerability in the WebGoat application. For example, in one of the lessons the user must use SQL injection to steal fake credit card numbers. The application is a realistic teaching environment, providing users with hints and code to further explain the lesson."

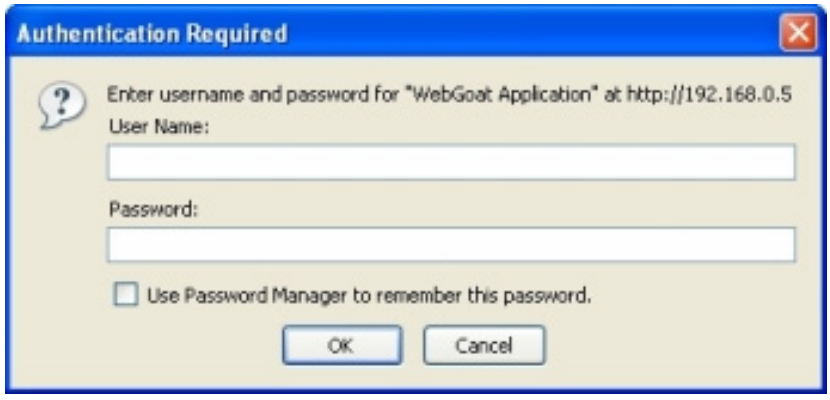
WebGoat can be downloaded from http://code.google.com/p/webgoat/downloads/list. It runs within Tomcat and can either be used stand-alone "out-of-the-box" with a version of Tomcat that is packaged with it, or a WebGoat.war file can be dropped onto an existing Tomcat installation.

Explaining WebGoat and the WebGoat lessons in detail is beyond the scope of this document. Each sublesson has a solution - available from the menu at the top of each lesson - that explains the purpose of the lesson. The motivation of the WebGoat solution (which is exploiting the vulnerability) is often accompanied by screenshots of WebGoat and WebScarab, the web proxy that is used in the solutions. If the reader does not wish to go through the WebGoat lessons themselves, the solutions are the "cheat sheets" for WebGoat, and because of that the solutions are provided in this project's documentation as is. If this is insufficient, installing and using WebGoat may be necessary to gain a complete understanding of the lessons.

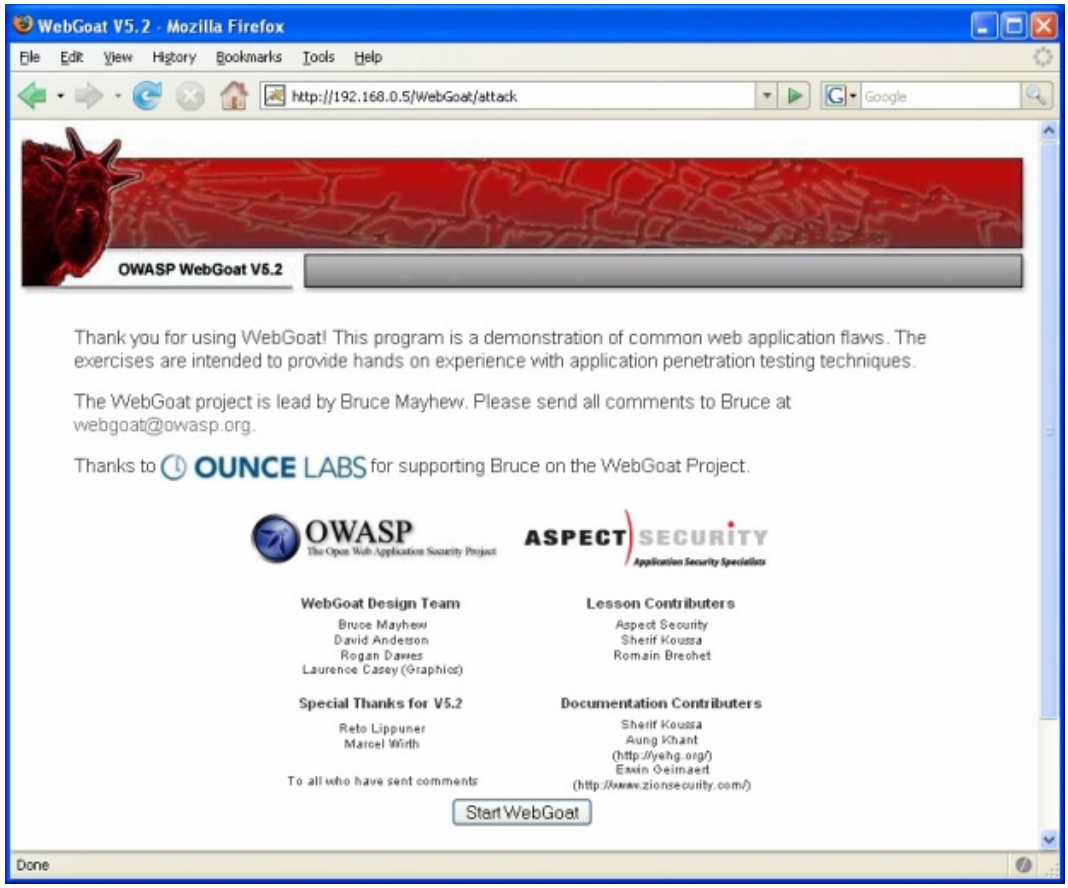
2.2 How it works

WebGoat users are configured in the tomcat-users.xml configuration file.

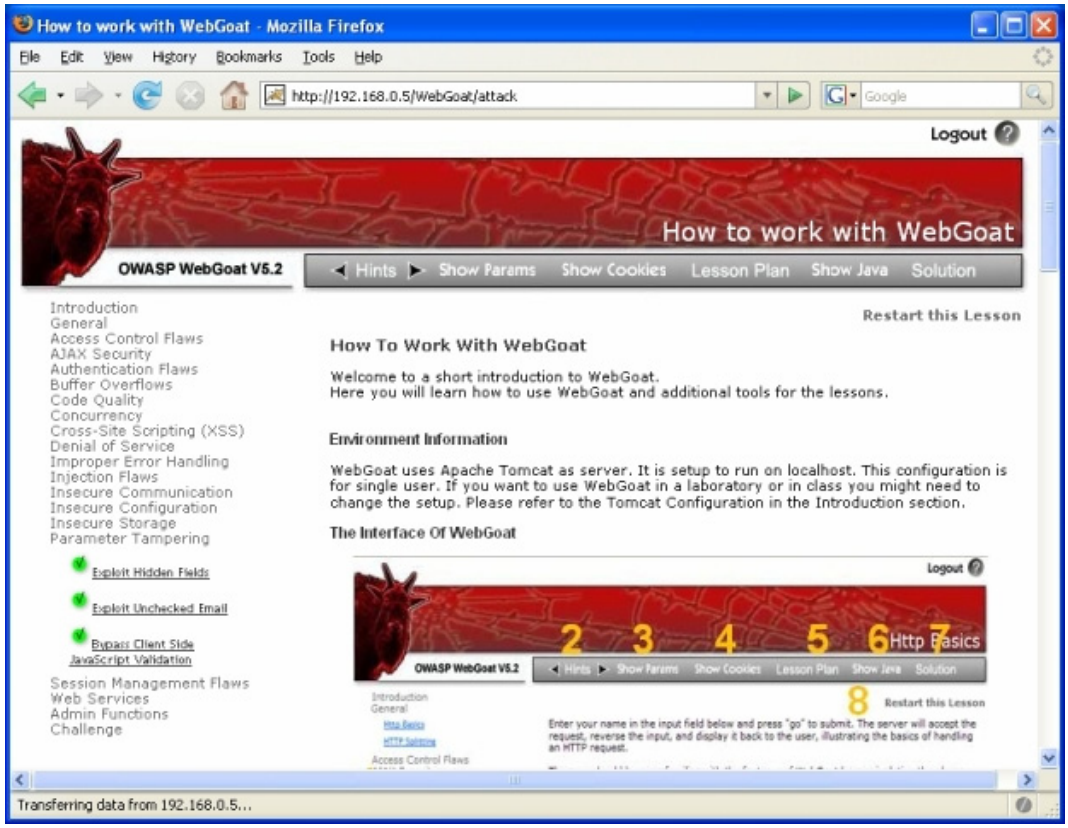
Here is the WebGoat login page, which uses Basic Authentication:



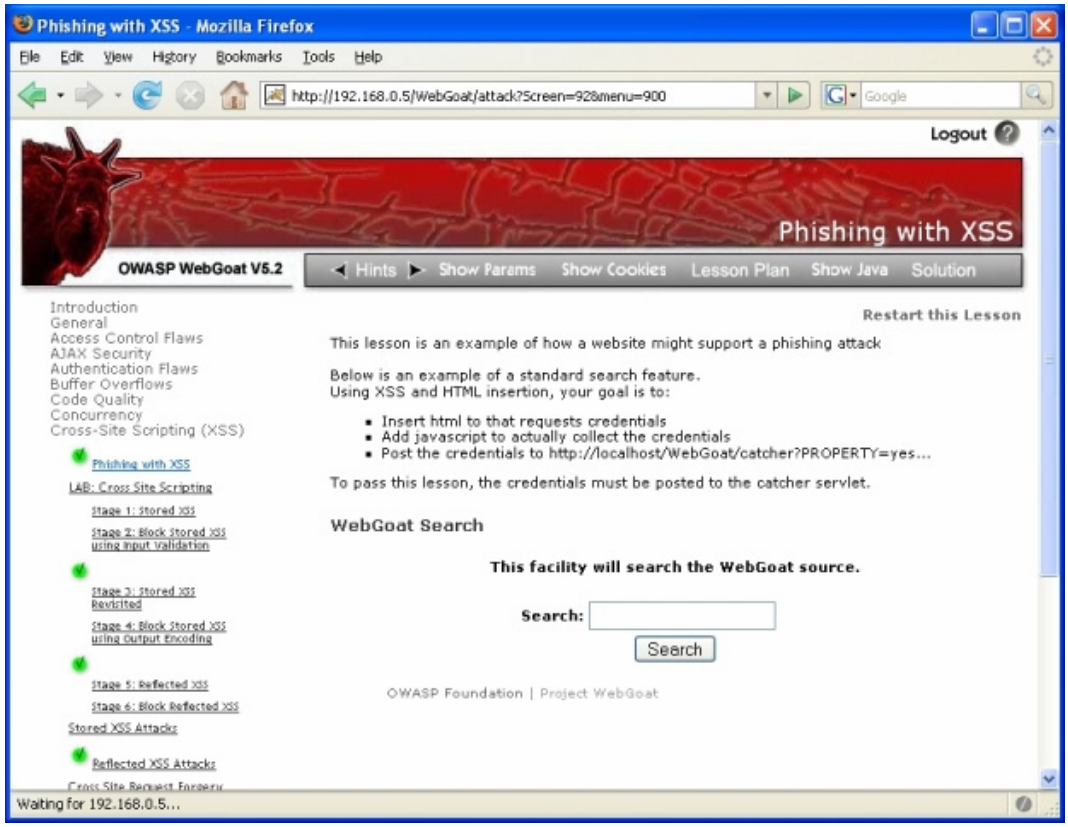
Next is the entrance/start page; a JSESSIONID cookie is set for the session variable:



Here is an overview of the lesson menu on the left side of the page that is used for navigation:

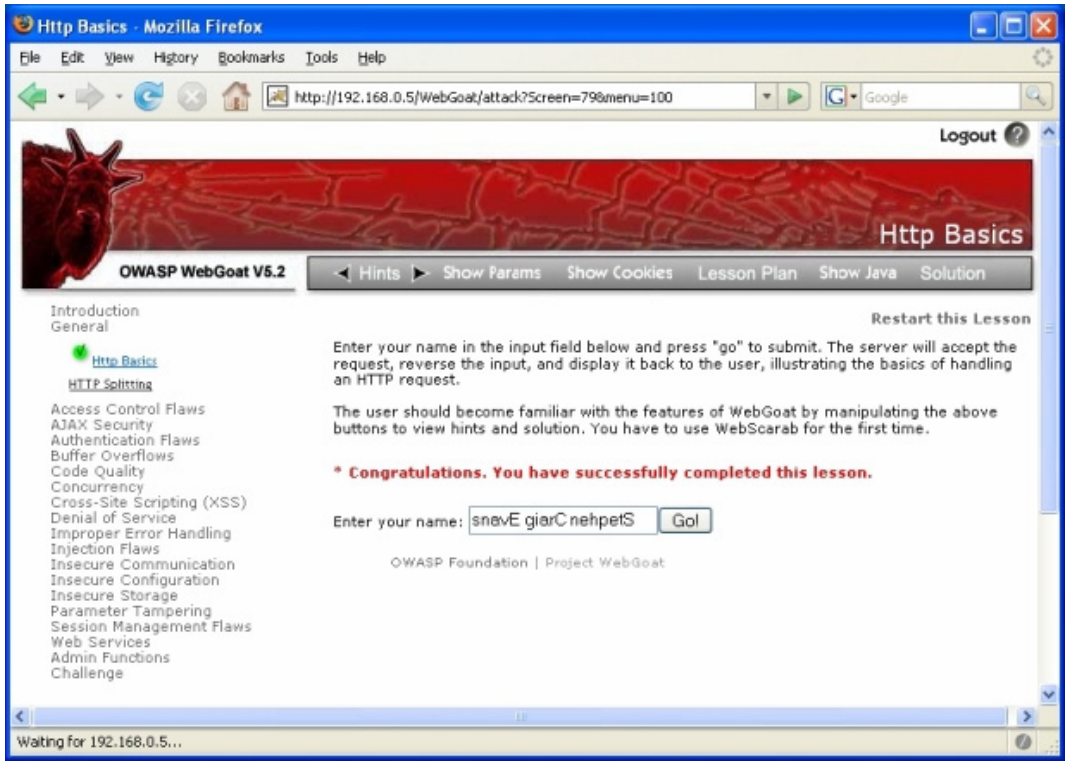


Each WebGoat lesson is designated by the 'menu' parameter in the query string. In the screen shot below, the 'screen' parameter will vary but the 'menu' parameter will always be 900 for this lesson. There is no similar way of easily identifying sublessons within a lesson or stages within a sublesson; ths sublesson 'Stored XSS Attacks' will also have 'menu' value of 900, and each of the stages in the 'LAB: Cross-Site Scripting' sublesson will always have 'menu' value of 900:

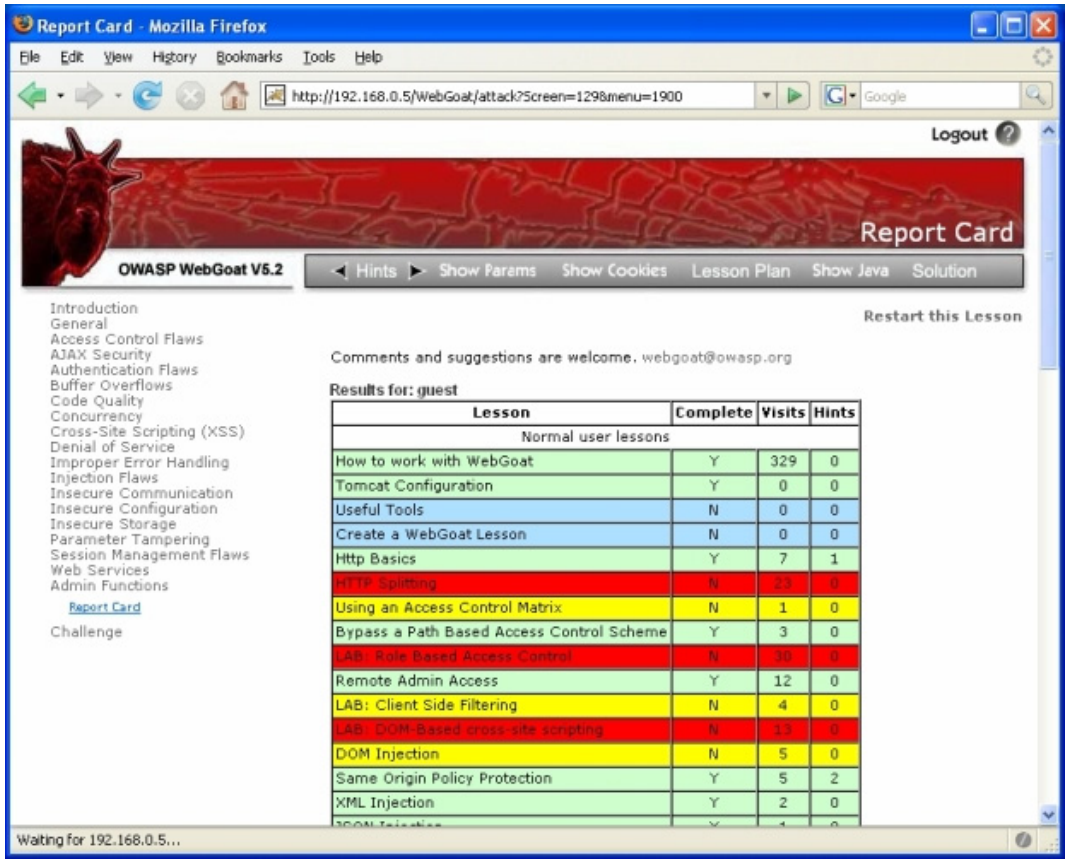


When a WebGoat lesson is completed successfully, there are 3 different areas that denote this:

- A message in red in the body of the HTML page and a green check mark beside the lesson in the menu on the left side:



- A report card in Section 18 of WebGoat:



2.3 Lesson table of contents

The WebGoat lessons are structured as main lessons, which are broken up into sub-lessons, and a sublesson may have several stages. For the purpose of this project, a numbering system is introduced to reference them. The unit of measure will be a sublesson. Lessons are too general and the amount of content for each one varies greatly, while using a stage as the unit of measure is too low-level and would be confusing.

The Table of Contents for WebGoat 5.2 beta 1 with the numbering system used for this project is:

1. General

- 1.1 Http Basics
- 1.2 HTTP Splitting

2. Access Control Flaws

- 2.1 Using an Access Control Matrix

- 2.2 Bypass a Path Based Access Control Scheme
- 2.3 LAB: Role Based Access Control
    - Stage 1: Bypass Business Layer Access Control
    - Stage 2: Add Business Layer Access Control
    - Stage 3: Bypass Data Layer Access Control
    - Stage 4: Add Data Layer Access Control
- 2.4 Remote Admin Access

3. AJAX Security
    - 3.1 LAB: Client Side Filtering
    - 3.2 LAB: DOM-Based cross-site scripting
    - 3.3 DOM Injection
    - 3.4 Same Origin Policy Protection
    - 3.5 XML Injection
    - 3.6 JSON Injection
    - 3.7 Silent Transactions Attacks
    - 3.8 Insecure Client Storage
    - 3.9 Dangerous Use of Eval
4. Authentication Flaws
    - 4.1 Password Strength
    - 4.2 Forgot Password
    - 4.3 Basic Authentication
    - 4.4 Multi Level Login 1
    - 4.5 Multi Level Login 2
5. Buffer Overflows
6. Code Quality
    - 6.1 Discover Clues in the HTML
7. Concurrency
    - 7.1 Thread Safety Problem
    - 7.2 Shopping Cart Concurrency Flaw

8. Cross-Site Scripting (XSS)

- 8.1 Phishing with XSS
- 8.2 LAB: Cross Site Scripting
    - Stage 1: Stored XSS
    - Stage 2: Block Stored XSS using Input Validation
    - Stage 3: Stored XSS Revisited
    - Stage 4: Block Stored XSS using Output Encoding
    - Stage 5: Reflected XSS
    - Stage 6: Block Reflected XSS
- 8.3 Stored XSS Attacks
- 8.4 Reflected XSS Attacks
- 8.5 Cross Site Request Forgery (CSRF)
- 8.6 HTTPOnly Test
- 8.7 Cross Site Tracing (XST) Attacks

9. Denial of Service

- 9.1 Denial of Service from Multiple Logins

10. Improper Error Handling

- 10.1 Fail Open Authentication Scheme

11. Injection Flaws

- 11.1 Command Injection
- 11.2 Blind SQL Injection
- 11.3 Numeric SQL Injection
- 11.4 Log Spoofing
- 11.5 XPATH Injection
- 11.6 String SQL Injection
- 11.7 LAB: SQL Injection
    - Stage 1: String SQL Injection
    - Stage 2: Parameterized Query #1
    - Stage 3: Numeric SQL Injection

    - Stage 4: Parameterized Query #2
- 11.8 Database Backdoors

12. Insecure Communication
    - 12.1 Insecure Login
13. Insecure Configuration
    - 13.1 Forced Browsing
14. Insecure Storage
    - 14.1 Encoding Basics
15. Parameter Tampering
    - 15.1 Exploit Hidden Fields
    - 15.2 Exploit Unchecked Email
    - 15.3 Bypass Client Side JavaScript Validation
16. Session Management Flaws
    - 16.1 Spoof an Authentication Cookie
    - 16.2 Hijack a Session
    - 16.3 Session Fixation
17. Web Services
    - 17.1 Create a SOAP Request
    - 17.2 WSDL Scanning
    - 17.3 Web Service SAX Injection
    - 17.4 Web Service SQL Injection
18. Admin Functions
    - 18.1 Report Card
19. The Challenge
    - 19.1 The CHALLENGE!

2.4 Overview of lesson results

Following is an overview of the lesson results from May 21 to May 29 using WebGoat 5.2 beta 1.

The intended behavior of a successfully completed lesson is a green check mark next to the lesson; if absent, at the very least, a message in red appears in the web page with a "Successfully completed..." message. In several cases, it appeared the lesson was completed successfully but not recorded as such. Some lessons did not complete because of a bug or that it was not working according to the solution. One lesson was not implemented. A few lessons required developer versions - source code - in Labs so that the code could be altered; those are not applicable since that's our motive via ModSecurity. A few lessons were not lessons that demonstrated a feature or vulnerability; hence, they are not applicable.

Note: It is duly noted that this version of WebGoat is beta 1; additionally, the inability of the project member to complete some of the WebGoat solutions is in no way a reflection on whether they actually can be completed or not.

Summary by category:

(a) Not done because not applicable: 2

18.1, 19.1

(b) Not done because not implemented: 1

5.0

(c) Not done because developer version required: 3

2.3, 8.2, 11.7

(d) Not successful because of bug or not doing correctly with the given solution: 9

3.1, 8.3, 8.5, 11.1, 11.5, 11.8, 16.2, 16.3, 17.4

(e) Successful but no red "successful" message and/or green check mark: 12

1.1, 1.2, 2.2, 3.7, 4.3, 7.1, 7.2, 9.1, 11.2, 12.1, 14.1, 17.1

(f) Successful: red "successful" message and/or green check mark: 29

2.1, 2.4, 3.2, 3.3, 3.4, 3.5, 3.6, 3.8, 3.9, 4.1, 4.2, 4.4, 4.5, 6.1, 8.1, 8.4, 8.6, 8.7, 10.1, 11.3, 11.4, 11.6, 13.1, 15.1, 15.2, 15.3, 16.1, 17.2, 17.3

Conclusion:

Not counting (a) and (b), 53 lessons were attempted and 41 lessons were successfully completed (some guesswork was involved in arriving at these numbers).

3. ModSecurity protecting WebGoat

This section details the strategy and work done in order to reach the goals of the project. When the term 'mitigated' is used throughout this document, it is used in the sense that the WebGoat vulnerability in a lesson has been prevented by ModSecurity from being exploited.

3.1 Project Setup and Environment

Disclaimer: The background of the project team member is software development and not system/network administration, so any suggestions or comments to improve the following configurations are welcome.

3.1.1 Network/hardware/software

The operating system is Kubuntu 7.10 on a Dell Inspiron laptop. Apache 2.2.7 and Tomcat 5.5 from the Kubuntu distribution is used; mod_jk glues Tomcat to Apache. Mod_proxy is used and configured so that Apache has a static IP address, WebGoat is accessible via port 80, and is available to other PCs on the internal network. For security, the NetGear wireless router is configured to block all HTTP & HTTPS traffic to and from the Web server to the outside world.

Firefox 2.0, Internet Explorer 7.0, and Opera 9.26 were used remotely on Windows XP SP2, and occasionally Firefox 2.0 was used on the Web server itself.

WebGoat version 5.2 beta 1 was used. The standard release of WebGoat 5.2 was posted to Google Code on 12 July 2008 and the second half of this project will be based on the standard release. Also, the ModSecurity solutions provided for the first 50% will be re-tested.

ModSecurity 2.5.1 was compiled, installed and used. For the 2nd half of the project, ModSecurity 2.5.5 (or a newer version if available) will be utilized.

3.1.2 Tools used

- WebScarab/Paros Proxy web proxies: The solutions use WebScarab and the project member used both WebScarab and Paros Proxy interchangeably throughout the project.
- Remo - Rule Editor for ModSecurity (http://remo.netnea.com/): Remo was invaluable to get started with an easy-to-use GUI that builds ModSecurity rules. Besides the tutorial video on the Remo site, another very helpful tutorial is at http://www.howtoforge.com/remo_modsecurity_apache.
- Regex tools: The Regex Coach 0.9.2 (http://weitz.de/regex-coach/) and Expresso 3.0 (http://www.ultrapico.com/Expresso.htm) were used for building, testing, and analyzing regular expressions.

- Lua tools: The standalone binaries and all related documentation and artifacts for using the Lua scripting language can be found starting at http://www.lua.org. A big help was a version of Notepad2 with Lua support (http://zigmar.googlepages.com/notepad2withluasupport).
- The ModSecurity debug file: It's simply not possible to go without the ModSecurity debug file set at level 9 for debugging.
- A text editor with line numbers: the ModSecurity debug file makes extensive references to line numbers with rulesets, so having a text editor with line numbers is essential for a debugging session. 'kate' was used on Kubuntu 7.10 and 'Notepad2' was used on WinXP.

3.2 Doing the WebGoat lessons - tips and tricks

Here are some tips and tricks to consider while using the ModSecurity rules from this project with WebGoat:

- WebGoat keeps track of the user's position within WebGoat on the back end out of the reach of ModSecurity. For instance, typing in the URL http://192.168.0.5/WebGoat/attack when the session ID (the JSESSIONID cookie) still exists will take the user directly to the position that they were previously. Since there is no 'menu' parameter in the URL, the ModSecurity rule for that lesson will not be invoked. It is necessary to explicitly choose the sublesson from the left side menu to invoke the ModSecurity rules for that lesson. To get WebGoat to forget the user's current position, erase the session cookie from within the browser and empty the cash (Firefox requires the fewest clicks to do this). In some scenarios, it is also necessary to intercept the HTTP request in a proxy and remove the Basic Authentication line (e.g 'Authorization: Basic Z3Vlc3Q6Z3Vlc3Q='), which will require the user to re-login.
- Within lessons, there is no unique identifier to easily distinguish a sublesson or stage of a sublesson from the others. The best way is to determine uniqueness by parameter names (ARG_NAMES in ModSecurity lingo). But this can get tricky; for example, in one lesson, one key in one sublesson is 'Username' while in another sublesson the key is 'username'. However, it is technically possible to key on the sublesson name in the web page banner, but the trade-off in extra clutter in the rulesets is not deemed a worthy trade-off.
- Different browsers have their pluses and minuses:
    - Internet Explorer 7: The solutions were done with IE7 (or, at least, the screenshots in the solutions are of IE7) so use it if all else fails. But, to toggle IE7 to use a web proxy requires closing the browser and re-opening it.
    - Firefox 2: The fewest mouse clicks are required to remove the WebGoat session cookies and clear the cache. Also, reconfiguring a web proxy is done

on the fly and does not require restarting the browser. One sublesson, 3.8 Insecure Client Storage, uses Firebug - a Firefox extension - but the other browsers have similar browser plug-ins (e.g. Opera has Dragonfly as of version 9.5).

- Opera 9.26: The AJAX Security lessons seemed to work best with Opera.
- Keep in mind that when a browser is going through a web proxy and you want to block requests and/or responses, unwanted crap gets sent from the browser to the outside world (anything Google-related was the biggest culprit, with Yahoo coming in second) and interferes with the Web proxy session. So, you might want to have more than one browser open; either your browser of choice to use with the proxy, or your browser of choice to use for general surfing and research while you are using WebGoat.

3.3 Testing ModSecurity rules – tips and tricks

The following are 2 contributor's suggestions to speed up the process of writing ModSecurity rules (especially RegExs) and testing them:

(1)
"I work with 4 shells on a single screen and I believe I get quite close to what you are looking for:

- 1 (top left): Apache config
- 2 (bottom left): Apache commands (-> alias "apareload")
- 3 (bottom right): curl commands
- 4 (top right): tail on the debug log (this can be filtered with tail -f ... | grep -v (...|...|...)

Alias apareload: That is a command alias to reload apache with the latest timestamp on the command line. This is very helpful when you play around with multiple apaches and configs. If you want to be really nifty, then you can script apareload into a while loop and whenever you press enter in that shell window, it reloads apache.

I do almost all my testing with "curl -v". Sometimes directly, sometimes scripted into an adhoc shell script (like: proceed by entering different strings). Looking at the http status, I can determine whether it matched or not. Sometimes I write multiple rules and have each one return a redirect to a different location. That way I can determine which one triggered. The same information is also in the debug log, but most of the time, the http response is enough for me.

Note the counterclockwise arrangement of the shell windows. Edit Apache, reload Apache, run curl, check the debug log, run curl, check the output or proceed with the debug log on top. Then eyes to the left again to the apache config. Of course, you can also arrange it clockwise or any other way. It's just the way the works for me. Your mileage may vary.

Maybe that's all obvious to experienced ModSec users. But I believe it is useful for newbies to get into a short reconfigure->tryout cycle. With the setup pointed out above, it takes merely a few seconds." - Christian Folini

(2)
"For the mean-time I have a very tiny tool (product of another night of coding) attached to this mail. You can run it by:

```
java -jar RegexpSelector.jar  [/path/to/samples.txt]
```

Image:OWASP Securing WebGoat using ModSecurity RegexpSelector.zip

(Or by double-clicking the jar-Archive on some systems)

It will present you a list of string-values and provide you with a box for entering a regular expression. As soon as you type your regex it will show, which string do match and which do not.

This very closely resembles ModSecurity's regexp matching. If you provide it with a file as an optional attribute, then it loads the sample strings from that file line-by-line." - Christian Bockermann

3.4 Project organization

Following is how the project has been organized. The overall intent is to formulate the rulesets and the mitigating solutions in the most modular and granular fashion possible with the fewest dependencies on each other. Another consideration was to introduce a file-naming and numbering system that is as intuitive as possible. Simplicity is favored over complexity.

3.4.1 ModSecurity rules

The numbering convention is 'rulefile', lesson number, sublesson number (if applicable), and description; e.g. 'rulefile_15_parameter-tampering.conf' is for Lesson 15: Parameter Tampering.

There is an 'initialization' rulefile, rulefile_00_initialize.conf, that contains all of the global configuration parameter needed by ModSecurity (e.g. SecRuleEngine, SecRequestBodyAccess, SecResponseBodyAccess, SecDataDir, SecDebugLog) plus a few rules that make the most sense - for clarity - to put there.

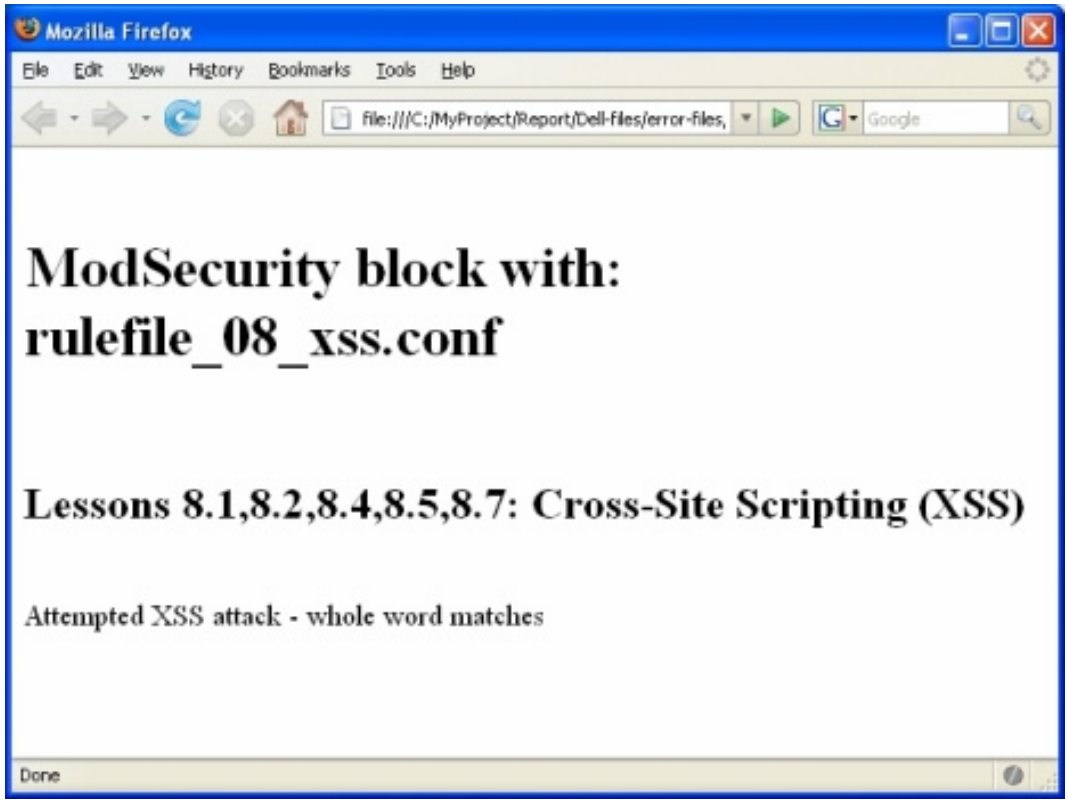
In most cases, each ruleset from a lesson applies to only that lesson; however, the Cross-site Scripting (Lesson 8) and Command Injection rulesets (Lesson 11) can act globally on all lessons. Otherwise, the initialization ruleset plus any other ruleset can act on their own for that particular lesson.

3.4.2 SecDirData directory

The SecDirData for ModSecurity persistence is configured in the initialization file. Since it already has the correct Apache read/write permissions, the same directory is used for Lua persistence and to store the Lua scripts. Also, this directory is hard-coded inside the Lua scripts.

3.4.3 Error pages

In the real world, only generic error messages should be reflected back to the end user. In our situation, this is an education project so each rule in each ruleset has its own HTML error file page that it is redirected to; for example, the error file 'lesson08a.html' for many of the XSS vulnerabilities in Lesson 8 is:



3.4.4 Informational and debug messages

Rules and Lua scripts contain informational and debug messages. It is a good practice to use unique and specific words in them (per rule and per ruleset) to make it easier to search through the ModSecurity debug and audit log files.

4. Mitigating the WebGoat lessons

4.1 Project metrics at 50% project completion

See Section 2 for the WebGoat lesson Table of Contents and an overview of the results from doing the WebGoat lessons. Appendix A contains a zip file which is made up of the lesson plans and solutions - in HTML format - which were taken from WebGoat and can be viewed stand-alone.

Out of 51 possible lessons, the following are teaching lessons, not vulnerabilities, and therefore have no context for ModSecurity rules:

- 1.1 Http Basics
- 4.1 Password Strength
- 15.3 Bypass Client Side JavaScript Validation
- 17.1 Create a SOAP Request

Therefore there are a total number of 47 lessons to do; half is 24 so that was the goal of the first 50% of project completion. The lowest hanging fruit was taken first because considerable effort was put into: (1) setup and configuration of the environment; (2) getting familiar with WebGoat and taking all of the lessons; (3) learning ModSecurity (and Remo); (4) re-learning regular expressions; (5) learning Lua script; and (6) developing an efficient work methodology.

The total number of sublessons mitigated by ModSecurity rules: 25 - thereby achieving the goal of at least 50% of sublessons mitigated.

They are:

- Sublesson 1.2
- Sublesson 2.4
- Sublessons 4.2, 4.4, 4.5
- Sublesson 6.1
- Sublessons 8.1, 8.2, 8.4, 8.5, 8.7
- Sublesson 10.1
- Sublessons 11.1, 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8
- Sublesson 13.1
- Sublessons 15.1, 15.2
- Sublessons 17.3, 17.4

4.2 Project metrics at 100% project completion

Based on Reviewer feedback, three (3) of the 4 sublessons that were originally assessed as not being able to be a ModSecurity solution, actually can be. They are:

- 1.1 Http Basics
- 15.3 Bypass Client Side JavaScript Validation
- 17.1 Create a SOAP Request

Therefore, there are a total possible number of 50 sublessons; 25 of which were done in Phase 1. To reach the goal of 90%, 45 sublessons would need to be solved; 20 in the 2nd phase.

Fifteen (15) sublessons were solved in the 2nd phase which makes a total of 40 sublessons solved.

Five (5) sublessons do not count because ModSecurity solutions cannot be written for them (further explanation below):

3. AJAX Security -> 3.3 Same Origin Policy Protection
5. Buffer Overflows (not implemented)
4. Authentication Flaws -> 4.1 Password Strength
8. Cross-Site Scripting (XSS) -> 8.6 HTTPOnly Test
14. Insecure Storage -> 14.1 Encoding Basics

The remaining 7 sublessons are either unable to be solved or can be solved but were complex solutions and weren't completed because of lack of time. The next section gives further explanation for each sublesson that was not solved.

Therefore, 40 out of 47 WebGoat sublessons were solved, or 85%. The original goal was 90%, or 42 lessons.

See 'Section 5, The Mitigating Solutions' for details on each of the ModSecurity solutions new in Phase 2. In chronological order, these 15 sublessons are:

- Sublesson 12. Insecure Communication -> 12.1 Insecure Login
- Sublesson 7. Concurrency -> 7.1 Thread Safety Problem
- Sublesson 7. Concurrency -> 7.2 Shopping Cart Concurrency Flaw
- Sublesson 2. Access Control Flaws -> 2.2 Bypass a Path Based Access Control Scheme
- Sublesson 8. Cross-Site Scripting (XSS) -> 8.3 Stored XSS Attacks

- Sublesson 9. Denial of Service -> 9.1 Denial of Service from Multiple Logins
- Sublesson 3. AJAX Security -> 3.4 DOM Injection
- Sublesson 3. AJAX Security -> 3.5 XML Injection
- Sublesson 3. AJAX Security -> 3.7 Silent Transactions Attacks
- Sublesson 3. AJAX Security -> 3.1 LAB: DOM-Based cross-site scripting
- Sublesson 3. AJAX Security -> 3.6 JSON Injection
- Sublesson 3. AJAX Security -> 3.8 Dangerous Use of Eval
- Sublesson 1. Introduction -> 1.1 Http Basics
- Sublesson 15. Parameter Tampering -> 15.3 Bypass Client Side JavaScript Validation
- Sublesson 17. Web Services -> 17.1 Create a SOAP Request

Note that as the project progressed, repeating ModSecurity solutions over and over resulted in standard formats and cleaner output for the solution write-ups and their artifacts (ModSecurity *.conf files, Lua scripts, error HTML files, etc.). For example, to get an example of a Lua script being used, find one later in the project because it will be more refined and clearer to understand.

4.3 Sublessons that don't count or were not solved (and why)

Sublessons that do not count:

3. AJAX Security -> 3.3 Same Origin Policy Protection

This WebGoat lesson demonstrates a browser's same origin policy protection and as such has no vulnerability with a ModSecurity solution or any other way to illustrate similar functionality within ModSecurity. Therefore, this sublesson is not part of the total count.

5. Buffer Overflows (not implemented by WebGoat)

4. Authentication Flaws -> 4.1 Password Strength:

This WebGoat lesson calls a 3rd party website and shows password strength (time to crack a user-supplied password that matches the criteria); as such it has no vulnerability with a ModSecurity solution or any other way to illustrate similar functionality within ModSecurity. Therefore, this sublesson is not part of the total count.

8. Cross-Site Scripting (XSS) -> 8.6 HTTPOnly Test:

This WebGoat lesson is a demonstration of the HttpOnly flag which prevents client side script from reading a cookie value if it is set; "Read Cookie" and "Write Cookie" buttons can be selected to view that particular browser's support of the HttpOnly flag.

There really is no ModSecurity solution or any other way to illustrate similar functionality within ModSecurity, especially since this takes place in the browser and ModSecurity cannot modify source code in an HTTP response.

14. Insecure Storage -> 14.1 Encoding Basics:

This WebGoat lesson demonstrates encoding and decoding (e.g. Base64, md5, entity, unicode); as such it has no vulnerability with a ModSecurity solution or any other way to illustrate similar functionality within ModSecurity. Therefore, this sublesson is not part of the total count.

Note: ModSecurity has many built-in encoding/decoding transformation functions - it supports each encoding/decoding function in this lesson demonstration, and more.

Sublessons that were not solved (but almost all have suggestions):

See the Mitigating Solutions section for the link to these sublessons:

2. Access Control Flaws -> 2.3 LAB: Role Based Access Control
3. AJAX Security -> 3.2 LAB: Client Side Filtering
3. AJAX Security -> 3.9 Insecure Client Storage
16. Session Management Flaws -> 16.1 Hijack a Session
16. Session Management Flaws -> 16.2 Spoof an Authentication Cookie
16. Session Management Flaws -> 16.3 Session Fixation
17. Web Services -> 17.2 WSDL Scanning

4.4 Unfinished business

The following topics need to be addressed in order to use the content of this project in a real world situation.

4.4.1 Concurrent file access

In the real world, a file locking mechanism will need to be implemented for any solution that uses Lua persistence and writes data to a file. The implemention should be easy; something like the Unix file.lock mechanism: check for the lock file before opening and writing; if it doesn't exist, then create the lock and then delete when finished. For the rare

occurrence that a requested file is in use, return an error message to the user saying "please try again", or implement a 'sleep' or 'wait' capability in Lua.

See this thread for more information on a Lua 'wait' function:

NEWBIE Question: LUA wait() function?
http://lua-users.org/lists/lua-l/2008-03/msg00209.html

4.4.2 Lua security in ModSecurity

This section was created after project completion in response to 2 reviewer comments:

```
(1) I am pretty sure your Lua code is vulnerable to (Lua)
Injection. For example, in '10. Sublesson 03.5' you do:

     _, _, inner1 = string.find(a, "<reward>(.-)</reward>")

Then:

     outstr = string.format("Entry{\n  reward = \"%s\"\n}\n\n",
inner1)

But you do not escape inner1.

I've already raised this issue, but I wouldn't let a solution that
uses this approach anywhere near my systems, for this very reason.

2) As previously commented, I am terrified with your usage of
dofile(), coupled with dynamically-generated code based on external
data. In fact, there may be a vulnerability or two in your code.
Even if there isn't, the point is that we, security professionals,
should be leading the way by using foolproof techniques. We mustn't
rely on things that are so easy to do wrong.
```

Security using Lua in ModSecurity is presented here as an opening discussion; anybody who wants to add to this section on the project wiki or have any other comments can email me at stephencraig_dot_evans_at_gmail_dot_com.

How this project uses Lua

A Lua script called from a ModSecurity rule returns either nil or non-nil; that result determines what action ModSecurity then takes.

Inside of a Lua script, "Lua persistence" is used to read and write data to ModSecurity's SecDataDir (for this project it is '/etc/modsecurity/data'). Some of the data is for configuration such as the number of password retries. Another case is to record information culled from an HTTP response, then later compare that with user input in an HTTP POST request.

The Lua 'dofile()' function

The Lua 'dofile()' function does not do anything extra that is not already in the Lua language; it is what is called in Lua lingo as 'sugarcoating'.

For example, a project data file might be:

```
Entry{
  username = "global",
  retry = 2,
  interval = 3,
  lockedout = "no",
  current = "no"
}

Entry{
  username = "webgoat",
  retry = 2,
  interval = 1216639104,
  lockedout = "yes",
  current = "no"
}

Entry{
  username = "admin",
  retry = 2,
  interval = 0,
  lockedout = "yes",
  current = "no"
}
```

A function is declared in the Lua script:

```
  function Entry (b)
    local uame = b.username
    local locked = b.lockedout
    local intvl = b.interval
    ...
  end
```

The function is called from the main Lua program:

```
 dofile(<datafile>)
```

This performs the function on each 'Entry' chunk in the data file. There's nothing magical about it: it is a convenience because it implicitly does looping and file I/O; 'dofile' could be implemented by using the existing Lua language functionality.

See Sublesson '4. Authentication Flaws -> 4.2 Forgot Password' for a more complicated use of 'dofile'.

Research sources for this section

Google search and the very-active, offical, Lua mailing list, lua-l, was used for researching this section. The excellent, official Lua mailing list archive, which has averaged around 700 messages per month for the last 5 years, is at: http://lua-users.org/lists/lua-l/

Possible attack vectors

(1) The Lua engine (in ModSecurity, it is a shared object file or dll)

(2) Affecting the return value of the Lua script

(3) Malicious user data

The Lua engine

There have been vulnerabilities in the C code that comprises Lua, e.g. buffer overflows which have caused Denial of Service but no known vulnerabilities that have allowed code execution. Of course, that doesn't mean that there is no possibility of it, plus there is no known "seal of approval" on the Lua engine. However, in the context of Lua in ModSecurity, full compromise would be restricted to the damage that can be done with the permission levels of the Apache process on Linux; on Windows it could be much worse.

Affecting the return value of the Lua script

An attacker that could access the file system of the ModSecurity installation can modify a Lua-persisted file and change its settings to alter the return value of the Lua script, causing either a false positive or false negative because the ModSecurity rule will do the opposite of the expected behaviour.

It must be taken into consideration that for this to occur, an attacker will already have write access to the SecDataDir with the privileges of the Apache process, and perhaps other important directories and files on the file system.

Malicous user data

User input written to a file that is later read from the file cannot be used maliciously to inject into any HTTP stream because ModSecurity does not have the capability to modify the HTTP request or HTTP response; therefore, any kind of malicious HTML or HTTP code is useless (there is one exception: malicious user data that is written by the Lua script to the ModSecurity debug log, then read by a Web browser).

User data stored in a file can later be accessed by a project Lua script to be compared with other values. For example, using 'dofile', a loop will read in each of these values and compare them with an HTTP POST paremeter value:

```
Entry{
  reward = "WebGoat Mug 20 Pts"
}
```

```
Entry{
  reward = "WebGoat t-shirt 50 Pts"
}

Entry{
  reward = "WebGoat Secure Kettle 30 Pts"
}
```

The danger is that a malicious Lua function can be written to Lua-persisted file because it was not sanitized, so instead of:

```
reward = "WebGoat Mug 20 Pts"
```

It can be:

```
reward = "<malicious Lua code>"
```

Next, if that data is accessed in a Lua script - e.g. 'tempvar = reward' - can a malicious function be executed?

The answer: maybe, but it hasn't been tested yet. A Lua function can be executed by using the Lua 'loadstring' function; e.g. loadstring(functionname .. "()"). Perhaps a malicious function can be defined at the beginning of a user input string, then be followed by the 'loadstring' function that will execute it.

Some pseudo code:

```
reward = "malfunc function() \n  <malicious
code>\nend;loadstring(malfunc .. "()")
```

Then, perhaps this statement will trigger the malicious function to execute:

```
tempvar = reward
```

Execution of scripts and programs by Apache

If there is a syntax error in a Lua script called by a ModSecurity rule, then Apache will not start; actually, Apache acts more like a compiler in that it resolves file references and other entities. Is it possible that Apache loads everything it is going to execute at startup? Answer: unknown.

If true, then:

1. Because user data is written to file during Apache runtime, dynamic execution of malicious Lua script in ModSecurity is not possible.

2. Most of this section's discussion on Lua security in ModSecurity is irrelevant.

3. ModSecurity will not execute malicious shell or Perl scripts constructed by a Lua script, then afterwards called by a subsequent ModSecurity rule.

This can be tested by writing a malicious Lua script in a stand-alone environment, then including it in a variable that has been written to file. Then load the variable in a Lua script called by a ModSecurity rule and see if the malicious Lua script executes.

Potential secure Lua coding options

In a nutshell, here are recommendations and/or options for secure Lua coding in:

1. Sanitize (e.g. HTML-encode) user data that is written to the debug log file; the Lua function that does this is: 'm.log(<log message>)'

2. It appears that any malicious Lua code needs left and right parenthesis. If true, write a Lua encode/decode function that converts a '(' or ')' to an obscure, never-used character (in Lua) when writing user data to a file, then reverse the process when reading it from a file.

3. Enforce a maximum string length of user data before writing to file.

4. In every file that reads user data from a file, disable all unnecessary Lua functionality that can be potentially dangerous, as outlined by one of the creators of Lua, Luiz Henrique de Figueiredo:

```
local E=debug.getregistry()
E._LOADED={}
E._LOADLIB=nil
E=nil
arg=nil
debug.debug=nil
debug.getregistry=nil
dofile=nil       -- this might not be needed in the particular Lua script
io={write=io.write}   -- this might not be needed in the particular Lua
script
loadfile=nil
os.execute=nil
os.getenv=nil
os.remove=nil
os.rename=nil
os.tmpname=nil
package=nil
require=nil
loadstring = nil      -- this one added is added for the project
```

5. Deploy other solutions such as sandboxing (see References below).

Conclusion

Presented in this section was an opening discussion on Lua security within the context of ModSecurity.

How would somebody know that Lua is being used within ModSecurity, and that ModSecurity is being used? Let's be honest: Mac, Linux, and mobile malware are virtually insignificant now because those platforms are not yet popular enough and homogeneus enough to spend the time on them. It's the same - and even far less of a risk - for using Lua in ModSecurity.

To look on the bright side - the silver lining of the cloud - if Metasploit starts releasing exploits for Lua script in ModSecurity, then we've made it!

References

Lua script as data model (Lua security)
http://lua-users.org/lists/lua-l/2008-09/msg00399.html

Re: Lua script as data model (Denial of Service)
http://lua-users.org/lists/lua-l/2008-09/msg00408.html

Re: function calls without brackets
http://lua-users.org/lists/lua-l/2008-06/msg00067.html

Re: Execute a function
http://lua-users.org/lists/lua-l/2008-08/msg00291.html
http://lua-users.org/lists/lua-l/2008-08/msg00294.html

Indirect call to function and passing arguments
http://lua-users.org/lists/lua-l/2008-07/msg00267.html

Secure tables in Lua?
http://lua-users.org/lists/lua-l/2008-07/msg00007.html

Secure tables in Lua: Summary
http://lua-users.org/lists/lua-l/2008-07/msg00065.html

Sandboxing techniques
http://lua-users.org/wiki/SandBoxes

Sandboxing user code in Lua
http://lua-users.org/lists/lua-l/2008-07/msg00344.html

Sandbox in Lua 5.1
http://lua-users.org/lists/lua-l/2008-05/msg00812.html

Re: Loose sandbox ***
http://lua-users.org/lists/lua-l/2008-07/msg00406.html

4.5 Overall strategy

The intention of mitigating the vulnerabilities is to demonstrate the largest variety of mitigating solutions and features of ModSecurity as possible:

- Some lessons are solved using the easiest way possible for convenience (and to count towards achieving the goal of 50% complete!)
- Some lessons are solved by using rules from the core rulesets provided by Breach Security
- Some mitigating solutions are meant to be global, meaning being in effect at all times, like the XSS and Command Injection core rules from Breach Security
- One lesson demonstrates the use of a session variable
- Some lessons require persistence beyond what is offered by ModSecurity; Lua scripts are used to achieve this
- Some lessons require a more robust capability than ModSecurity's regular expressions, 'and/or' logical mechanisms, and goto statements (skip and skipAfter); again, Lua scripts are used to achieve this
- One lesson uses the 'append' action to append JavaScript to the end of a response body, which alters the content and behavior of the HTML page

The rulesets can be used all together or, for a specific WebGoat sublesson, the initialization file (rulefile_00_initialize.conf) plus that sublesson's ruleset can be used.

The best way to open the discussion about the overall strategy used is to show a chunk of the initialization file:

```
<LocationMatch "^/WebGoat/attack$">
  # Group 1: the following block pertain to pages that don't have
  #   Screen or menu parameters
1.  SecRule &ARGS:Screen "!@eq 0" chain,skipAfter:200
2.  SecRule &ARGS:menu "!@eq 0" "t:none"

  # Group 2: set session collection if entering WebGoat; POST body
parameter
  #   is "start=Start WebGoat" (Start WebGoat submit button)
3.  SecRule &ARGS_POST:start "@eq 0" "nolog,skip:3"
4.  SecRule ARGS_POST:start "!@streq Start WebGoat" \
      "t:urlDecodeUni,t:htmlEntityDecode,skip:2"
5.  SecRule REQUEST_COOKIES:JSESSIONID "!^$" \
      "chain,log,auditlog,pass,msg:'Setting session collection'"
6.  SecAction setsid:%{REQUEST_COOKIES.JSESSIONID}
7.  SecAction "log,setvar:session.lesson13=0,msg:'setting
session.lesson13=0 \
      initially after setsid from rulefile_00-0-initialize.conf'"

8.  SecAction "t:none,allow,id:'200'"

  # Group 3: here there should be a 'menu' parameter; set a variable for
  #   the menu number that's used if needed in Phase 4
```

```
9.  SecRule ARGS_GET:menu "^(.*)$" "pass,setvar:tx.menu=%{MATCHED_VAR}"
```

In Group 1, the two chained rules denote the major section of WebGoat after the login and start pages. The 'Screen' parameter is arbitrary, while the 'menu' parameter is the lesson key. Basically, if this condition is met, skip past Group 2.

In Group 2, the next 2 rules pertain to the start page, denoted by the 'start=Start WebGoat' POST body parameter. Nothing of importance occurs on this page so we skip to the 'allow' action on line 8.

Line 5 and 6 denote that a successful login has occurred, so the session ID is saved in a session collection for later use throughout the remaining lessons. Line 7 sets a session variable (a toggle switch) for use in 'Lesson 13: Insecure Configuration'.

At Line 9, we have to save the 'menu' value for use in Phase 4.

At this point, we know the 'menu' value: it exists throughout phase 2 as is, and we have saved the value in a variable for Phase 4.

For the rest of the ruleset files, we will filter on the 'menu' parameter and each lesson will have its own ruleset file.

For example, ruleset file 'rulefile_04_authentication-flaws.conf' has 2 sections:

`'SecRule ARGS:menu "!@eq 500" "phase:2,t:none,skip:2"'` says "if we aren't in Lesson 4 in Phase 2, then don't do any further processing here".

Similarly, `'SecRule TX:MENU "!@eq 500" "phase:4,t:none,pass,skip:1"'` says "if we aren't in Lesson 4 in Phase 2, then don't do any further processing here".

4.6 Reviewer comments

For many lesson solutions, a project reviewer has made comments regarding the project's choice of a solution (or lack of one), and often also has given a "classic" ModSecurity solution to those lessons solved by using Lua script.
Reviewer comments are added at the end of each individual lesson solution write-up in its own section; to date, the following sublessons have been updated with reviewer comments:

- 1. General -> 1.2 HTTP Splitting
- 2. Access Control Flaws -> 2.4 Remote Admin Access
- 4. Authentication Flaws -> 4.2 Forgot Password
- 4. Authentication Flaws -> 4.4 Multi Level Login 1
- 4. Authentication Flaws -> 4.5 Multi Level Login 2
- 6. Code Quality -> 6.1 Discover Clues in the HTML
- Lesson 8.0: 8. Cross-Site Scripting (XSS)] Addressing XSS attacks
- 8. Cross-Site Scripting (XSS) -> 8.2 LAB: Cross Site Scripting
- 8. Cross-Site Scripting (XSS) -> 8.4 Reflected XSS Attacks

- 8. Cross-Site Scripting (XSS) -> 8.5 Cross Site Request Forgery (CSRF)
- 8. Cross-Site Scripting (XSS) -> 8.6 HTTPOnly Test
- 8. Cross-Site Scripting (XSS) -> 8.7 Cross Site Tracing (XST) Attacks
- 16. Session Management Flaws -> 16.1 Hijack a Session
- 16. Session Management Flaws -> 16.2 Spoof an Authentication Cookie
- 16. Session Management Flaws -> 16.3 Session Fixation

4.7 Using the Lua scripting language

In some situations, the ModSecurity Rule Language is not flexible enough to prevent complex exploits. Enter Lua script, a tried-and-true programming language used mainly in the gaming industry but also used in other areas such as Nmap and the MySQL Proxy project.

Some advantages are:

1. Enables ModSecurity to address business logic flaws
2. Enhances the capability of using ModSecurity as an egress filter
3. Lua persistence: Super-persistence beyond the existing Session scope and capability (the SecDataDir directory is used for storage)
4. By being able to use a programming language, it is easier to do whitelisting and implement a positive security model.
5. Flexibility for virtual patching (as in, mitigating vulnerabilities discovered in a penetration test and NOT as in porting Snort or scanner signatures).
6. Ease-of-use: The Lua scripts can be developed and tested offline as standalone programs before being used by ModSecurity.
7. A programming language that is built to be rugged; e.g. handling multi-megabyte memory buffers and files.

In Phase 1, Lua scripts are used in sublessons:

- 4.2: Forgot Password (OWASP_ModSecurity_Securing_WebGoat_Section4_Sublesson_04.2)
- 4.4: Multi Level Login 1, 4.5: Multi Level Login 2 (OWASP_ModSecurity_Securing_WebGoat_Section4_Sublesson_04.4_04.5)
- 8.2 LAB: Cross Site Scripting (OWASP_ModSecurity_Securing_WebGoat_Section4_Sublesson_08.2_08.4_08.5_08.7)
- 15.1 Exploit Hidden Fields, 15.2 Exploit Unchecked Email (OWASP_ModSecurity_Securing_WebGoat_Section4_Sublesson_15.1_15.2)

In Phase 2, Lua scripts are used in sublessons:

- 3. AJAX Security -> 3.4 DOM Injection
- 3. AJAX Security -> 3.5 XML Injection
- 3. AJAX Security -> 3.6 JSON Injection
- 7. Concurrency -> 7.1 Thread Safety Problem

  Shows the process of developing a standalone Lua script on Windows, then integrating it with ModSecurity on Linux.
- 7. Concurrency -> 7.2 Shopping Cart Concurrency Flaw

  Shows how Lua can be used to handle application state; in this case, a purchase that utilizes a shopping cart.
- 8. Cross-Site Scripting (XSS) -> 8.3 Stored XSS Attacks

  Shows rudimentary output sanitization using Lua.
- 9. Denial of Service -> 9.1 Denial of Service from Multiple Logins
- 12. Insecure Communication -> 12.1 Insecure Login

  Shows: (1) using JavaScript injection that adds an MD5 library on the front end in order to hash a password before being sent; (2) rebuilding the Lua engine and including a 3rd party MD5 library; and (3) hashing the expected password within a Lua script and comparing it with the client side password.

The Implementation section of each sublesson explains in detail how Lua script is used.

4.8 Using JavaScript ‘prepend’ and ‘append’

The JavaScript 'prepend' and 'append' ModSecurity actions and their applications are shown in sublessons:

- 3. AJAX Security -> 3.1 LAB: DOM-Based cross-site scripting
- 3. AJAX Security -> 3.7 Silent Transactions Attacks
- 12. Insecure Communication -> 12.1 Insecure Login

4.9 Structure of mitigating a lesson

The following outlines the overall structure of each lesson/sublesson to be used in the next section that contains the details of each mitigating solution:

Lesson overview: The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution: Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy (including challenges): What approach was taken to solve the lesson and why; what were the challenges? (if any)

Implementation: Details how the sublesson was mitigated.

Note that the project's solution files are formatted using a Linux editor with a 2-space tab. For viewing the files from Windows, it's preferable to use Wordpad instead of Notepad because Notepad might add CR/LF combinations. Still, using Notepad and the files in DOS format on Linux should pose no problems but using Wordpad is recommended.

Comment(s): Any comments on the solution.

4.10 The mitigating solutions

Following are links to the mitigating solution pages. Some may link to the same page if their sublesson is solved by the same rule or by a group of rules in the same *.conf file.

Two asterisks (**) next to a sublesson denotes that it has not been solved but there are suggestions to solve it. Four asterisks (****) next to a sublesson indicates that there is no solution and the reason.

1. Sublesson 1.1: HTTP Basics
2. Sublesson 1.2: HTTP Splitting
3. Sublesson 2.2: Bypass a Path Based Access Control Scheme
4. Sublesson 2.3: LAB: Role Based Access Control **
5. Sublesson 2.4: Remote Admin Access
6. Sublesson 3.1: LAB: DOM-Based cross-site scripting
7. Sublesson 3.2: LAB: Client Side Filtering ****
8. Sublesson 3.4: DOM Injection
9. Sublesson 3.5: XML Injection
10. Sublesson 3.6: JSON Injection
11. Sublesson 3.7: Silent Transactions Attacks
12. Sublesson 3.8: Dangerous Use of Eval
13. Sublesson 3.9: Insecure Client Storage **
14. Sublesson 4.2: Forgot Password
15. Sublesson 4.4: Multi Level Login 1
16. Sublesson 4.5: Multi Level Login 2

17. Sublesson 6.1: Discover Clues in the HTML
18. Sublesson 7.1: Thread Safety Problem
19. Sublesson 7.2: Shopping Cart Concurrency Flaw
20. Lesson 8: Cross-Site Scripting (XSS) (Addressing XSS attacks - Reviewer comments)
21. Sublesson 8.1: Phishing with XSS
22. Sublesson 8.2: LAB: Cross Site Scripting
23. Sublesson 8.3: Stored XSS Attacks
24. Sublesson 8.4: Reflected XSS Attacks
25. Sublesson 8.5: Cross Site Request Forgery (CSRF)
26. Sublesson 8.7: Cross Site Tracing (XST) Attacks
27. Sublesson 8.6: HTTPOnly Test **
28. Sublesson 9.1: Denial of Service from Multiple Logins
29. Sublesson 10.1: Fail Open Authentication Scheme
30. Sublesson 11.1: Command Injection
31. Sublesson 11.2: Blind SQL Injection
32. Sublesson 11.3: Numeric SQL Injection
33. Sublesson 11.4: Log Spoofing
34. Sublesson 11.5: XPATH Injection
35. Sublesson 11.6: String SQL Injection
36. Sublesson 11.7: LAB: SQL Injection
37. Sublesson 11.8: Database Backdoors
38. Sublesson 12.1: Insecure Login
39. Sublesson 13.1: Forced Browsing
40. Sublesson 15.1: Exploit Hidden Fields
41. Sublesson 15.2: Exploit Unchecked Email
42. Sublesson 15.3: Bypass Client Side JavaScript Validation
43. Sublesson 16.1: Hijack a Session **
44. Sublesson 16.2: Spoof an Authentication Cookie **

45. Sublesson 16.3: Session Fixation **
46. Sublesson 17.1: Create a SOAP Request
47. Sublesson 17.2: WSDL Scanning **
48. Sublesson 17.3: Web Service SAX Injection
49. Sublesson 17.4: Web Service SQL Injection

5. The Mitigating Solutions

5.1 Sublesson 1.1: HTTP Basics

1. Introduction -> 1.1 Http Basics

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This lesson shows the basics of HTTP and the use of a web proxy to examine HTTP traffic.

For a ModSecurity solution to this lesson, relevant ModSecurity configuration basics will be presented along with debugging tips and tricks. Performing this simple WebGoat lesson, one can experiment with the configuration directives, examine the debug log file, and be better prepared to implement the ModSecurity solutions in this project.

```
SecRuleEngine On
SecRequestBodyAccess On
SecResponseBodyAccess On
```

The SecRuleEngine and SecRequestBodyAccess are no-brainers but many of this project's solutions require "SecResponseBodyAccess On" in order to use the HTTP request body.

```
SecDataDir /etc/modsecurity/data
```

The directory and file permissions on this directory on non-Windows systems is very important; otherwise several of the project solutions will not work properly. The directory must be writable and a file's owner and group must be the same as the Apache process, which is defined in the Apache configuration file.

```
SecContentInjection On
```

This directive is essential for the solutions that use JavaScript injection ('prepend' or 'append') into the response body.

```
SecDefaultAction "log,deny,phase:2,status:501"
```

This is an interesting directive. Most of the processing is done in Phase 2 of the Apache cycle (see http://www.apachetutor.org/dev/request and http://stein.cshl.org/~lstein/talks/perl_conference/apache_api/lifecycle.html for more information). It is not necessary to put 'phase:2' in SecRules or SecActions, but at times in this project's solution configuration files it is explicitly stated when Phase 4 processing is also done. One of the biggest beginner mistakes is not to include 'phase:4' in a SecRule or SecAction directive when the Response body is to be processed.
"status:501" is the "Internal Server" error. When you see it while using the project's ModSecurity solution files, it is a bug somewhere; often it is seen when a variable in a Lua script has a nil value instead of an expected string. Check the debug log file when this message appears.

```
SecAuditEngine
SecAuditLog
SecAuditLog2
SecAuditLogParts
SecAuditLogRelevantStatus
SecAuditLogStorageDir
SecAuditLogType
```

In the real world use of ModSecurity, these audit log directives are very important but for this project the audit log file was not looked at once. The debug log cranked up to level 9 is where the action is.

The audit log configuration used for this project is:

```
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus ^5
SecAuditLogParts ABIEFHZ
SecAuditLogType Serial
SecAuditLog /etc/modsecurity/logs/modsec_audit.log
```

Debugging

```
SecDebugLog /etc/modsecurity/logs/modsec_debug.log
SecDebugLogLevel 9
```

Every single bug or unwanted behavior can be discovered by setting the debug log file at Level 9. Instead of guessing what the problem is and messing around with other stuff, set this to 9 and reproduce the problem. If you spend any significant amount of time using ModSecurity, it is best to learn how to read the debug as quickly as possible. The debug file can get large quickly, so go to the step immediately before the problem, then do this from '/etc/modsecurity/logs' (an apology to Windows users):

```
echo > modsec_debug.log
/etc/init.d/apache2 restart   # if you made changes to any ModSecurity
files loaded at Apache startup
```

Use the 'history' command to make this faster: 'history -c' to clear before typing in those commands. Then type in the above commands in order. Next time, one only has to type in something like:

```
!11
!12
```

over and over and over.

On the first attempt to debug, try command line search like 'grep' or 'findstr' on key words like 'rulefile_' or 'lua' to isolate the problem. There's valuable information not logged from debug messages written by the ModSecurity rule or Lua script creator, so if that fails to find the problem, load the debug file in a text editor (line numbering is an essential feature) and search from there.

As mentioned in several other places in this project, the more unique and consistent user-created debug messages are, the easier it is to pinpoint problems.

Some tips, learned from bad experiences and long hours

ModSecurity session collections and variables: A confusing aspect of ModSecurity persistence is when to use 'initcol' to load session collections and stored variables. It seems that it is not necessary when in the same phase or within the same ruleset file (with more than one phase), but necessary at all other times [Note: This observation by the project member is presumably wrong; a reviewer stated that "'initcol' has to be executed at least once, anywhere prior to the place where the collection is being used. Phases and rule location do not matter."]. The debug log explicitly states when a session collection is loaded; a missing 'initcol' will result in a message like 'Variable does not exist'.

Use of 'skip': When there are many ruleset files, using 'skipAfter' can potentially cause a problem if there are duplicate SecActions with the same 'id' number so in some cases it is preferable to use 'skip'. But, during development and debugging, or using 'chain', the 'skip' number can get out of alignment and jump over directives that should be processed. Again, going straight to the debug log set at level 9 and stepping through the expected ModSecurity directives one at a time will save a lot of time.

Use of 'phase:4': Forgetting to use 'phase:4' in a directive when processing the HTTP response body will produce strange results because they will be processed in the default phase which is normally phase 2 the request phase.

Use of 'allow': Mistakenly using 'allow' instead of 'allow:request' when it is intended to end processing of the request phase results can take eons to discover if one doesn't go straight to the debug log file; very clearly it will state something like "Skipping processing response phase"; the "Starting phase RESPONSE_BODY" message and the expected ModSecurity directives will not appear.

The processing order of ruleset files: Apache processes *.conf files in alphabetical order which is why the initialization file needed by all project lesson solutions is named 'rulefile_00_initialize.conf' and the lesson solutions follow the naming convention of 'rulefile_xx_*.conf' where 'xx' is the lesson number and '*' is a description of the sublesson [Reviewer note: "Explicitly listing the rules will determine the order that they will be processed."]. Using multiple solution files at the same time may cause unintended results.

The Apache 'LocationMatch' directive: When processing HTTP requests and WebGoat accesses files and directories outside of the normal '/WebGoat/attack' directory, be careful about the ordering of 'LocationMatch' when multiple ModSecurity configuration files are used. The symptom is that the intended configuration file will not be processed and it will not appear in the debug log file.

Lua Debugging

A convention has been developed over the time of this project to make debugging Lua scripts as easy as possible.

```
  msg0 = "Luascript (activate-request_03-4.lua): "
  msg1 = string.format("Post parameter: name = %s; value = %s;",
parmname, parmvalue)
  msg2 = msg0 .. msg1
  m.log(9, msg2)
```

Note the flexibility that Lua has over ModSecurity when writing error messages to the debug log file.

5.2 Sublesson 1.2: HTTP Splitting

1. General -> 1.2 HTTP Splitting

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

The solution is to prevent carriage returns and line feeds from passing through. ModSecurity ruleset 'modsecurity_crs_40_generic_attacks.conf' already has some rules for an HTTP response splitting attack so they were used. In the second rule, note the transformations urlDecodeUni and htmlEntityDecode.

Implementation

The lesson is mitigated by the ruleset 'rulefile_01_general_http-splitting.conf':

```
# The first rule is not necessary to solve the WebGoat lesson
SecRule REQUEST_URI|REQUEST_HEADERS|REQUEST_HEADERS_NAMES "%0[ad]" \
       "t:lowercase,capture,log,auditlog,deny,severity:3, \
msg:'HTTP Response Splitting Attack via URI/Header',logdata:'%{TX.0}', \
tag:'HTTP_SPLITTING',redirect:/_error_pages_/lesson01a.html"

SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES|XML:/* \
        "(?:\bhttp\/(?:0\.9|1\.[01])|<(?:html|meta)\b)" \
       "t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,capture,log,auditl
og,deny, \
severity:3,msg:'HTTP Response Splitting Attack via args/file name', \
logdata:'%{TX.0}',tag:'HTTP_SPLITTING',redirect:/_error_pages_/lesson01b
.html"
```

Reviewer comments

The Strategy/Implementation details look fine as you are using our existing Core Rule IDs 950910/950911. The only issue that I would raise, and it is not specific to this particular issue, is that of anti-evasion considerations when choosing the ModSecurity transformation functions to use. As you are all aware, the bad guys are constantly trying to bypass these types of filters by messing with encodings, whitespace formatting, etc… With this in mind, we may need to rethink and retest the applied trans functions. Ivan has been doing some research in this area and can provide more details and recommendations.

One item that I would suggest that you test is that of adding the ModSecurity multiMatch action - http://www.modsecurity.org/documentation/modsecurity-apache/2.5.5/modsecurity2-apache-reference.html#N11554.

This may not be the best option from a performance perspective, but it will help to identify attacks as we would be inspecting the data both before and after each trans function is applied vs. the normal process of only inspecting it after all of them are completed.

I have also seen web apps that allow clients to submit html. Oftentimes they use %0A and %0D to actually format html and not necessarily for HTTP Response Splitting attacks. We would need a different parser or something a bit more granular for this (Anti-Samy-ish).

5.3 Sublesson 2.2: Bypass a Path Based Access Control Scheme

2. Access Control Flaws -> 2.2 Bypass a Path Based Access Control Scheme

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson demonstrates access control bypass: a file from a dropdown list is chosen and sent, but intercepted in the web proxy and substituted with '../../../conf/tomcat-users.xml', which reveals tomcat users and passwords.

The solution to this WebGoat lesson is to prevent directory traversal.

Implementation

The lesson is mitigated in the ruleset 'rulefile_02_access-control-flaws.conf':

```
  # Lesson 2.2; directory traversal in 'File' parameter of POST request
  SecRule &ARGS:File "!@eq 0" "chain,log,auditlog,deny,msg:'Path
Traversal Attack', \
tag:'PATH_TRAVERSAL',redirect:/_error_pages_/lesson02-2.html"
  SecRule ARGS:File "\.\.[/\x5c]" "t:urlDecodeUni"
```

5.4 Sublesson 2.3: LAB: Role Based Access Control

2 Access Control Flaws -> 2.3 LAB: Role Based Access Control

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

In this WebGoat lab, each role has permission to certain resources (A-F). Each user is assigned one or more roles. Only the user with the 'Admin' role should have access to the 'F' resources but there is a vulnerability where the user doesn't have the 'Admin' role can access resource; this is exploited by intercepting a request in a web proxy and altering the Employee ID number.

The WebGoat solution is to modify the back end source code and add an 'isAuthorized' method.

A ModSecurity solution would be to Lua-persist the Role Based Access Control configuration to 2 *.data files.

Implementation

One *.data file would contain a list of roles with the resources that they have access to, something like this:

```
Entry{
  role = "Admin",
  resources = "A,B,D,F"
}

Entry{
  role = "User",
  resources = "A,B,C"
}
...
```

The second *.data file would contain each user and their roles:

```
Entry{
  employee_id = 101,
  roles = "User, Manager"
}
...
```

ModSecurity would intercept the request and call a Lua script which would determine if the request to the resource was valid or not.

Comments

- In the real world, a web interface would serve as a front end to the *.data files.

5.5 Sublesson 2.4: Remote Admin Access

2. Access Control Flaws -> 2.4 Remote Admin Access

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

The solution is to prevent 'admin=true' from appearing in the query string.

Implementation

The lesson is mitigated in the ruleset 'rulefile_02_access-control-flaws.conf':

```
  # Lesson 2.4: Remote Admin Access; don't allow 'admin=true' in the
querystring
  SecRule &ARGS:admin "!@eq 0" "chain,log,auditlog,deny,\
    msg:'Admin Function Attack',tag:'ADMIN_FUNCTION', \
    redirect:/_error_pages_/lesson02-4.html"
  SecRule ARGS:admin "true" "t:lowercase"
```

5.6 Sublesson 3.1: LAB: DOM-Based cross-site scripting

3. AJAX Security -> 3.1. LAB: DOM-Based cross-site scripting

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson demonstrates DOM-Based cross-site scripting. The vulnerability in this WebGoat lesson is to enter malicious code in the "Enter your name" text box like:

```
<img src=x onerror=;;alert('XSS') />
```

The source code snippet containing the vulnerability is:

```
<script src='javascript/DOMXSS.js'
language='JavaScript'></script><script
 src='javascript/escape.js' language='JavaScript'></script>
 <h1 id='greeting'></h1>Enter your name:
 <input value='' onKeyUp='displayGreeting(person.value)' name='person'
 type='TEXT'><br><br><input name='SUBMIT' type='SUBMIT'
 value='Submit Solution'>
```

The 'displayGreeting' function is in the file 'DOMXSS.js'.

The solution for the lesson is to modify WebGoat's source code and HTML-escape 'person.value' with a function 'escapeHTML' function which is in the file 'escape.js'.

Implementation

Since this vulnerability is in the JavaScript code, the attack payload stays on the client side and ModSecurity will not see it since it will not be sent to the server.

Instead, our solution will be to replace the 'displayGreeting' function with our own by appending it to the end of the HTML page and calling the 'escapeHTML' function in 'escape.js'. While not practical (an attacker could bypass the new function), replacing an existing JavaScript function with a new one will be demonstrated.

The JavaScript function:

```
<script language="JavaScript1.2" type="text/javascript">
function displayGreeting(name) {
  if (name != ''){
    document.getElementById("greeting").innerHTML="Hello, " +
escapeHTML(name) + "!";
  }
}
</script>
```

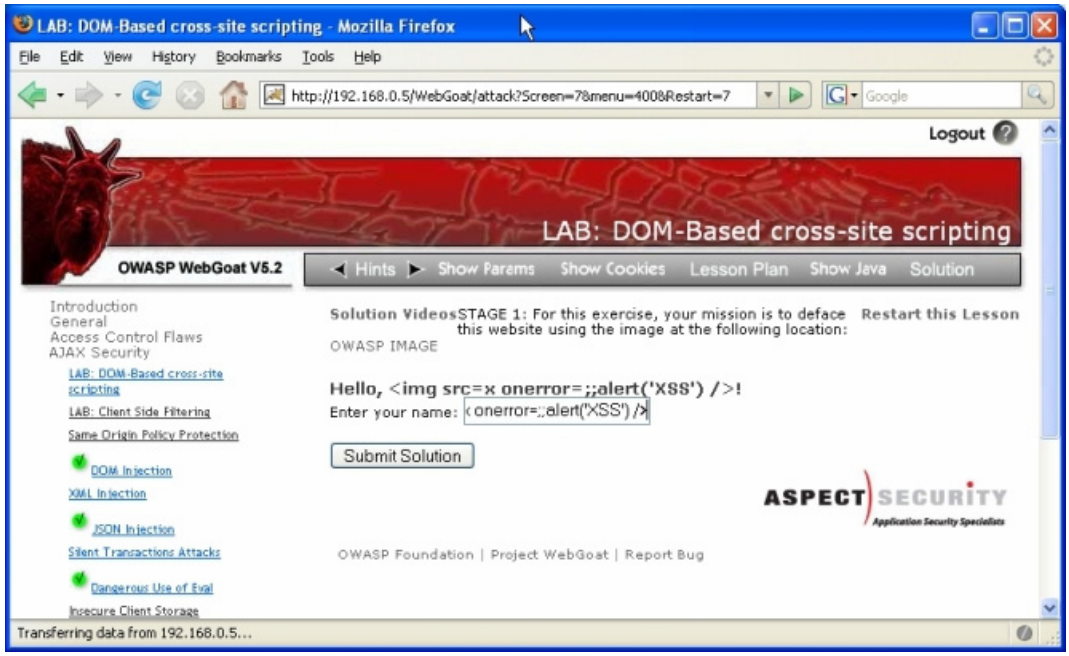
Before placing the function into a ModSecurity rule, test it by intercepting the HTTP response in a web proxy, copy and paste the code after the </html>, then forward the response to the browser.

A phase 2 rule is not necessary; the function can be placed in every HTML page for this lesson.

The phase 4 rule in 'rulefile_03-1_DOM-based-XSS.conf' is (the JavaScript should be packed on one line):

```
  SecRule RESPONSE_CONTENT_TYPE "!^text/html"
"phase:4,t:none,allow,nolog"

  # Here check if session variable set; if so, then subst.
'displayGreeting' function
  SecRule SESSION:LESSON031 "@eq 1" "phase:4,t:none,log,auditlog,pass,
    msg:'appending javascript in rulefile_03-1_DOM-based-XSS.conf',
    append:'<script language=\"JavaScript1.2\" type=\"text/javascript\">
function displayGreeting(name) {if (name != \"\")
{document.getElementById(\"greeting\").innerHTML=\"Hello, \"
+ escapeHTML(name) + \"!\";}}</script>'"
```

When malicious input is entered, it is properly escaped and displayed on the HTML page:



Comments

- This lesson demonstrates replacing an existing JavaScript function with a new one.

5.7 Sublesson 3.2: LAB: Client Side Filtering

3. AJAX Security -> 3.2 LAB: Client Side Filtering

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

In this WebGoat lab, an XPath query returns name, salary, and other information about users in XML in a hidden table. The vulnerability is that user information is sent for users not within the scope of the requesting user and client side filtering is relied upon to show only the necessary users.

The WebGoat solution is to modify the source code and return only the users and their information that is allowed according to the user requesting the information.

This is essentially an information leakage problem that does not need to be exploited (the extra information is always there).

Since ModSecurity cannot alter an HTTP response (intercept, then remove the unnecessary data), there seems to be no ModSecurity solution to this vulnerability.

Implementation

None

5.8 Sublesson 3.4: DOM Injection

3. AJAX Security -> 3.4 DOM Injection

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson modifies an AJAX response with JavaScript that enables the 'Activate!' button for a license key by replacing the response body (about 28k bytes) with:
`document.forms[0].SUBMIT.disabled = false;`

Implementation

When the button is disabled, it comes from the server as:

```
<input disabled value='Activate!' name='SUBMIT' type='SUBMIT'>
```

The ModSecurity solution will be to set a variable in 'lesson03-4.data' called 'isdisabled' to "yes" if the 'Activate!' button is disabled; "no" otherwise. For the subsequent request, if the 'Activate!' button is submitted such as:

```
POST
http://192.168.0.5/WebGoat/attack?Screen=131&menu=400key=aaaa&SUBMIT=Act
ivate%21
```

Then we check against the persisted variable to see if it is enabled or disabled, then block the request if the button was originally disabled.

'lesson03-4.data' will have the simple format of:

```
Entry{
  isdisabled = "yes"
}
```

First, we start with the response body because if the 'Activate!' button is sent in a request, for sure it had to be sent with its state in the previous response body.

The phase 4 response portion of the configuration file is:

```
  SecRule TX:MENU "!@eq 400" "phase:4,t:none,pass,skip:1"

  SecRuleScript "/etc/modsecurity/data/activate-response_03-4.lua" \
"phase:4,t:none,log,auditlog,allow,msg:'Luascript: AJAX Security -> \
3.4 DOM Injection: in RESPONSE; checking if Activate button is
disabled'"
```

Refer to the Lua script 'activate-response_03-4.lua'. The steps are:

- put the response body into a buffer
- search and get the state of the Activate button
- write the state to the data file

The phase 2 request portion of the configuration file is:

```
  SecRule ARGS:menu "!@eq 400" "phase:2,t:none,skip:4"
  SecRule &ARGS_POST:SUBMIT "@eq 0" "nolog,skip:3"
```

```
  SecRule &ARGS_POST:key "@eq 0" "nolog,skip:2"

  # action is triggered if script returns non-nil value
  SecRuleScript "/etc/modsecurity/data/activate-request_03-4.lua" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'Luascript: \
AJAX Security -> 3.4 DOM Injection: request is pending', \
tag:'INJECTION_ATTACK',redirect:/_error_pages_/lesson03-4.html"
  SecAction "phase:2,allow:request,t:none,log,auditlog,msg:'Luascript: \
AJAX Security -> 3.4 DOM Injection: acceptable state for Activate
button'"
```

Refer to the Lua script 'activate-request_03-4.lua'. The steps are:

- get the button state from the data file
- if it is enabled, return OK (nil) to ModSecurity; if it is disabled, return an error (non-nil)

Comments

- AJAX for this lesson only worked using Opera 9.26

5.9 Sublesson 3.5: XML Injection

3. AJAX Security -> 3.5 XML Injection

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson adds more rewards to the allowed set of rewards by intercepting an AJAX response and appending these 2 entries to the XML list:

```
<reward>WebGoat Core Duo Laptop 2000 Pts</reward>
<reward>WebGoat Hawaii Cruise 3000 Pts</reward>
```

However, the lesson is broke on the back end.

When rewards are selected, a POST is sent, for example:

```
accountID=836239&check1001=on&check1002=on&check1003=on&SUBMIT=Submit
```

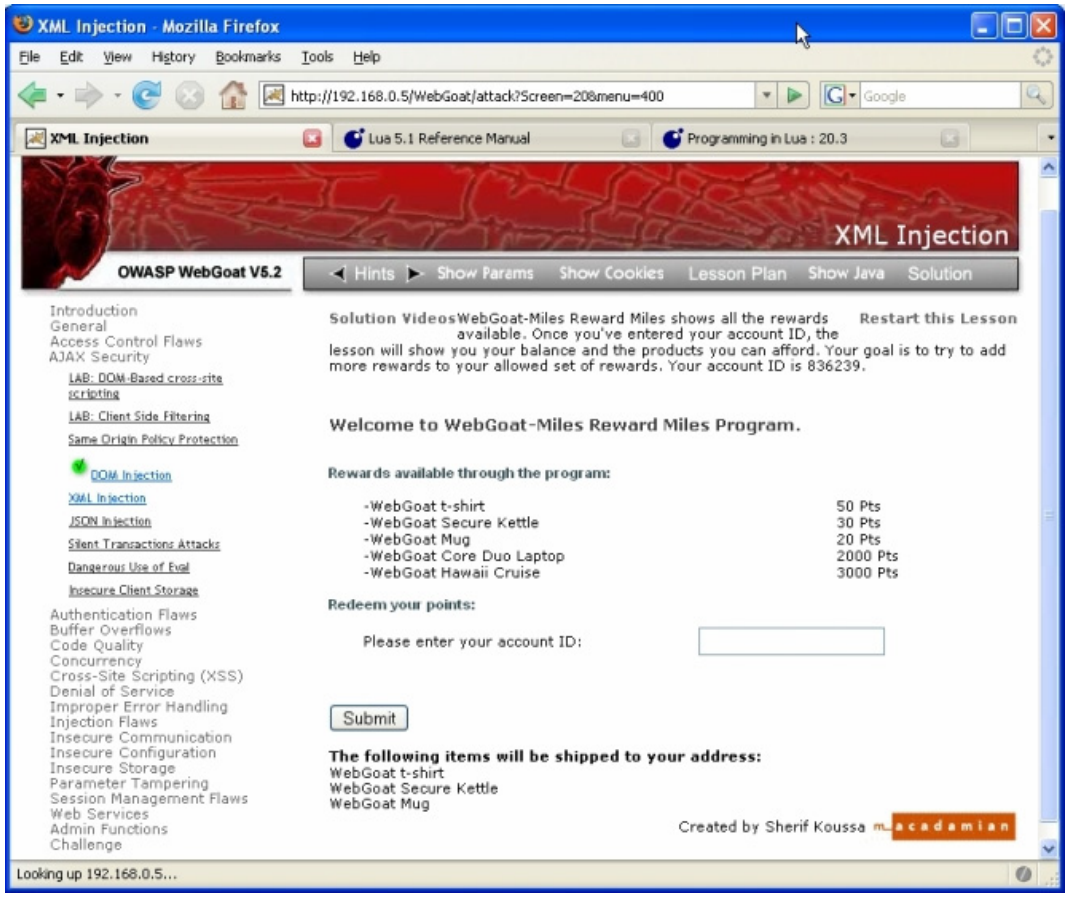
The problem is that there is no association between a checked entry, e.g. 'check1001' and a reward. This is because in the callback routine of the Ajax request, numbers are assigned irrespective of the reward:

```
  for(var i=0; i< rewards.length; i++){
    strHTML = strHTML + '<tr><td><input name="check' + (i+1001) +'"
type="checkbox">
```

To prove it, I did not add the 2 high-priced point rewards, I substituted them for the 't-shirt 50 Pts' and 'Secure Kettle 30 pts'; the return message should be:

```
The following items will be shipped to your address:
WebGoat Core Duo Laptop
WebGoat Hawaii Cruise
WebGoat Mug
```

But the return message was erroneous:



Therefore, since the rewards cannot be distinguished from each other, the only choice is to count the number of rewards sent, and if that doesn't match the number of rewards in

the original HTTP response - before manipulation in the web proxy - then an error is thrown.

Implementation

The ModSecurity solution will be to count the number of rewards sent to the user in the response body, then compare that with the number of rewards that the user subsequently sends in the request.

Start the lesson with an empty data file 'lesson03-5.data' with the proper file permissions.

First, process the response body and persist the rewards list to the data file.

ModSecurity has to be configured to process XML; add 'text/xml' to the 'SecResponseBodyMimeType' directive in 'rulefile_00_phase2-initialize.conf':

```
SecResponseBodyMimeType text/plain text/html text/xml
```

The phase 4 response portion of the configuration file is:

```
  SecRuleScript "/etc/modsecurity/data/rewards-response_03-5.lua" \
"phase:4,t:none,log,auditlog,allow,msg:'Luascript: \
AJAX Security -> 3.5 XML Injection: in RESPONSE; writing rewards to
file'"
```

Refer to the Lua script 'rewards-response_03-5.lua'. The steps are:

- read the response body into a buffer
- extract each reward using the pattern '<reward>(.-)</reward>'
- write each reward to the data file

An example will look like:

```
Entry{
  reward = "WebGoat t-shirt 20 Pts"
}

Entry{
  reward = "WebGoat Secure Kettle 50 Pts"
}

Entry{
  reward = "WebGoat Mug 30 Pts"
}
```

For phase 2, the POST request parameters have the form:

```
accountID=836239&check1001=on&check1002=on&check1003=on&SUBMIT=Submit
```

The phase 2 request portion of the configuration file is:

```
  SecRule &ARGS_POST:SUBMIT "@eq 0" "nolog,skip:3"
  SecRule &ARGS_POST:accountID "@eq 0" "nolog,skip:2"
```

```
  # action is triggered if script returns non-nil value
  SecRuleScript "/etc/modsecurity/data/rewards-request_03-5.lua" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'Luascript: \
AJAX Security -> 3.5 XML Injection: request is pending', \
tag:'INJECTION_ATTACK',redirect:/_error_pages_/lesson03-5.html"
  SecAction "phase:2,allow:request,t:none,log,auditlog,msg:'Luascript: \
AJAX Security -> 3.5 XML Injection: no illegal attempts made to add
rewards'"
```

Refer to the Lua script 'rewards-request_03-5.lua'. The steps are:

- get the POST request parameters
- loop through them and for each 'check*' argument name, increment the number of rewards by 1
- count the number of rewards in the data file
- compare the total and return an error if not equal

Comments

- This lesson shows how to use a 'do' loop in Lua and retrieve POST parameter names and values

5.10 Sublesson 3.6: JSON Injection

3. AJAX Security -> 3.6 JSON Injection

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson, a schedule and airfare for a roundtrip flight from Boston (BOS) to Seattle (SEA) is requested; the AJAX HTTP response is intercepted, the fare of the higher priced flight in the JSON array is lowered from $600 to $100, and the flight is bought at the lower price.

Implementation

The ModSecurity solution will be to persist the actual prices coming from the AJAX request and compare the price of the flight chosen by the user; if they are not the same, the request is blocked.

A snippet of the source code will look like:

```
"flights": [
{"stops": "0", "transit" : "N/A", "price": "$600"},
{"stops": "2", "transit" : "Newark,Chicago", "price": "$300"}
]
```

Start the lesson with an empty 'lesson03-6.data' file; once populated it will have the format of:

```
Entry{
  radioindex = 0,
  price = 600
}

Entry{
  radioindex = 1,
  price = 300
}
```

First, we start with the response body because we persist the flights here.

The phase 4 response portion of the configuration file 'rulefile_03-6_json-injection.conf' is:

```
   SecRuleScript "/etc/modsecurity/data/flights-response_03-6.lua" \
"phase:4,t:none,log,auditlog,allow,msg:'Luascript: AJAX Security -> \
3.6 JSON Injection: in RESPONSE; writing flight prices to file'"
```

Refer to the Lua script 'flights-response_03-6.lua'. The steps are:

- read the response body into a buffer
- extract the information from each flight from the buffer (array index and price) and write to the data file

After the price is manipulated and when the purchase is made, the POST parameters are:

```
travelFrom=BOS&travelTo=SEA&radio0=on&SUBMIT=Submit&price2Submit=%24100
```

In this example, zero from 'radio0' has to be extracted to get the correct index in the array; then obtain the price from 'price2Submit'.

The phase 2 request portion of the configuration file 'rulefile_03-6_json-injection.conf' is:

```
  SecRule ARGS:menu "!@eq 400" "phase:2,t:none,skip:4"
  SecRule &ARGS_POST:SUBMIT "@eq 0" "nolog,skip:3"
  SecRule &ARGS_POST:price2Submit "@eq 0" "nolog,skip:2"

  # action is triggered if script returns non-nil value
  SecRuleScript "/etc/modsecurity/data/flights-request_03-6.lua" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'Luascript: AJAX
Security \
-> 3.6 JSON Injection: An illegal attempt was made to alter the flight
price',\
```

```
tag:'INJECTION_ATTACK',redirect:/_error_pages_/lesson03-6.html"
  SecAction "phase:2,allow:request,t:none,log,auditlog,msg:'Luascript:
AJAX Security \
-> 3.6 JSON Injection: no illegal attempts made to alter the flight
price'"
```

Refer to the Lua script 'flights-request_03-6.lua'. The steps are:

- retrieve the POST parameters
- loop through each argument; extract the radio parameter index (e.g. zero from 'radio0') that is on plus the price parameter value
- loop through the data file until arriving at the correct index
- compare the prices and return an error message if they are not equal

Comments

- This lesson shows how to use a 'do' loop in Lua and retrieve POST parameter names and values

5.11 Sublesson 3.7: Silent Transactions Attacks

3. AJAX Security -> 3.7 Silent Transactions Attacks

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson shows a money transfer page with the user's balance, the recipient's account ID and the amount of money to transfer. The application uses AJAX to submit the transaction. The vulnerability is that malicious code injected into the page can call the AJAX code directly - bypassing client side validation - which results in a silent transaction without the user's authorization.

The attack is simulated by typing 'javascript:submitData(1234556,11000);' into the browser's address bar. The HTTP GET Ajax request looks exactly the same whether it comes from the JavaScript functions as intended or is called directly in the browser.

Normally, the the HTML code calls the 'processData()' function which does authorization and validation, then calls the 'submitData' function. The attack bypasses 'processData()' and calls 'submitData' directly.

The goal of the ModSecurity solution is to notify the user when a silent transaction commences.

Implementation

The ModSecurity solution is to append JavaScript to the lesson's start page HTTP response body which extends the the 'submitData' function by displaying a confirmation message box to the user. If they push 'Cancel', the transaction will not go through.

Here is the JavaScript code:

```
<script language="JavaScript1.2" type="text/javascript">
var oldsubmitData = submitData;

submitData = function(accountNo, balance)
{
  if (confirm("You sure you want to make this transaction?"))
  {
    if (oldsubmitData != null)
    {
      oldsubmitData(accountNo, balance);
    }
  }
  else
  {
    alert("Transaction canceled at your request");
  }
}
</script>
```

ModSecurity appends JavaScript at the very end of the response body after the '</html>' tag. Even though the resulting format does not adhere to the HTML 4.01 and HTML 5 specifications, current Web browsers will accept this format.

Before incorporating the solution into ModSecurity, test it by intercepting the HTTP response in a web proxy, copy and paste the code after the '</html>', then forward the response to the browser.

To build the ModSecurity rule, it is necessary to know the format of the GET request. The AJAX request is of the form:

```
GET
http://192.168.0.5/WebGoat/attack?Screen=40&menu=400&from=ajax&newAccoun
t=1234556&amount=11000
```

Because of the way WebGoat it is implemented, for this ModSecurity solution to work the lesson must explicitly be restarted by clicking on the link "Restart this Lesson".

The ModSecurity phase 4 rule that appends the JavaScript to the response body can only be invoked when it comes from the GET request as shown above. To implement this

behavior, it is necessary to set a flag - a session variable - in the request. If a GET request is not the lesson restart, the flag is turned off as a safeguard.

The phase 2 request portion of the configuration file 'rulefile_03-7_silent-transaction.conf' is:

```
  SecRule &ARGS_GET:Restart "@eq 0" "nolog,skip:4"
  SecRule &ARGS_GET:amount "!@eq 0" "nolog,skip:3"
  SecRule &ARGS_GET:newAccount "!@eq 0" "nolog,skip:2"

  SecRule REQUEST_COOKIES:JSESSIONID "!^$" \
"chain,log,auditlog,pass,msg:'rulefile_03-7_silent-transaction.conf: \
Setting session collection'"
  SecAction setsid:%{REQUEST_COOKIES.JSESSIONID}
  SecAction "log,setvar:session.lesson037=1,msg:'setting
session.lesson037=1 \
initially after setsid from rulefile_03-7_silent-
transaction.conf',skipAfter:370"

  # if not the GET request that we want, as a safeguard turn the flag
off
  SecAction "log,setvar:session.lesson037=0,msg:'setting
session.lesson037=0 \
initially after setsid from rulefile_03-7_silent-transaction.conf'"

  SecAction "t:none,allow:request,id:'370'"
```

The first 3 rules filters out the AJAX request and allows only the lesson restart. The next 2 rules creates the session collection, and the following rule sets the flag for that session.

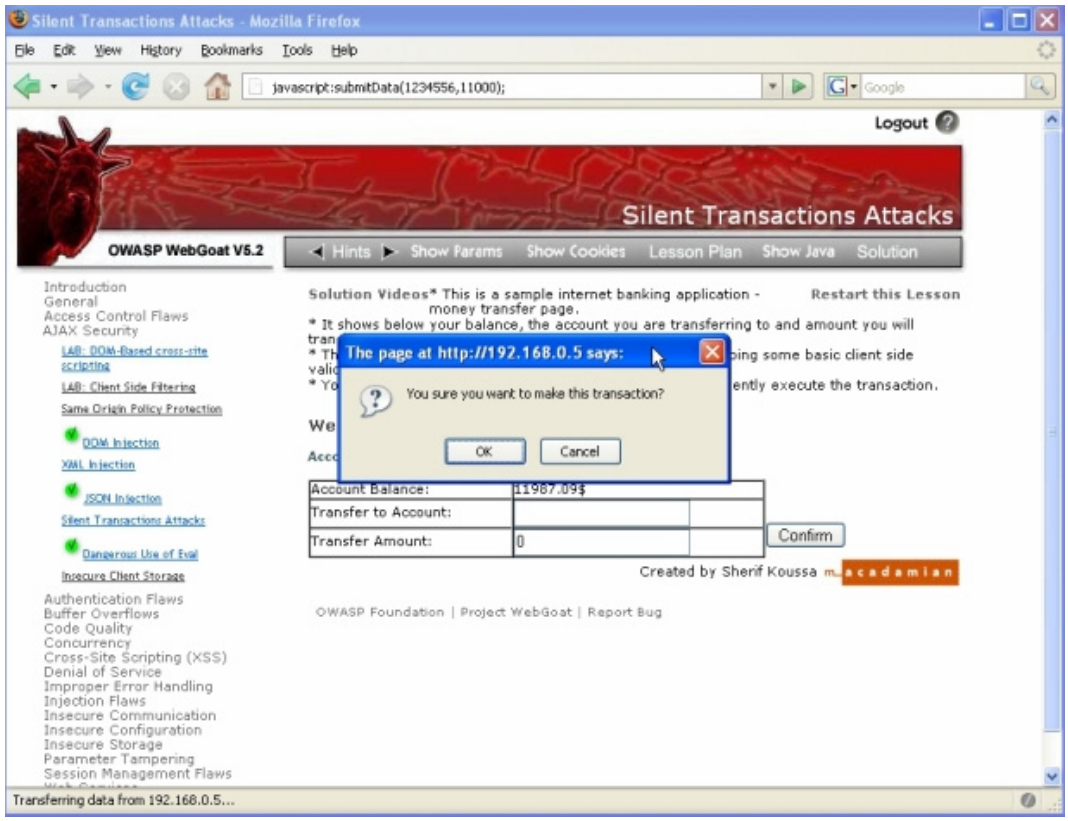
The phase 4 response portion of the configuration file 'rulefile_03-7_silent-transaction.conf' is (the JavaScript should be packed on one line):

```
  # finished if not 'text/html'
  SecRule RESPONSE_CONTENT_TYPE "!^text/html"
"phase:4,t:none,allow,nolog"

  SecAction "phase:4,log,initcol:session=%{SESSIONID}, \
msg:'rulefile_03-7_silent-transaction.conf: getting session.lesson037'"

  # Here check if session variable set;
  #   if so, then override 'submitData' function so that user is
prompted
  SecRule SESSION:LESSON037 "@eq 1" "phase:4,t:none,log,auditlog,pass, \
msg:'appending javascript in rulefile_03-7_silent-transaction.conf', \
append:'<script language=\"JavaScript1.2\" type=\"text/javascript\"> \
var oldsubmitData = submitData; submitData = function(accountNo,
balance)\
 {if (confirm(\"You sure you want to make this transaction?\")) \
{if (oldsubmitData != null) {oldsubmitData(accountNo, balance);}} \
else{alert(\"Transaction canceled at your request\");}}</script>'"
```

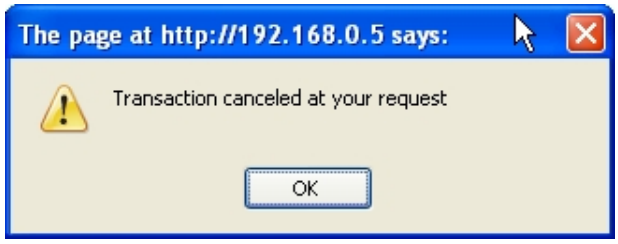
Since the session collection is accessed in a different phase than it was created, the 'initcol' action must be called to load it.

Also, note that the double quotes in the packed JavaScript function must be escaped with a back slash.

If a silent transaction is attempted, a confirmation message box appears:



And after canceling the transaction:



Comments

- This lesson demonstrates the use of ModSecurity's session collections.

- This lesson demonstrates ModSecurity's JavaScript injection function, 'append'. It is useful when the attacker is not the user and acts as a 3rd party and would place malicious code in a page (otherwise, if the attacker was the user, it would simply bypass the new, injected JavaScript).

- 'initcol' has to be used to load and access session collection variables under two scenarios: (1) When the collection is created in an earlier phase; and (2) When the collection is created in a separate ModSecurity configuration file, even if the phase is the same.

- Unfortunately, the way WebGoat is set up, there is no way to distinguish this Sublesson 3.7 from the other Lesson 3's - all are 'menu=400', so this solution cannot be used with any of the other solutions in Lesson 3. There's one caveat: if somehow it can be the last *.conf file in the group and control properly filters down to it, then it can be used with the other Lesson 3 sublessons.

5.12 Sublesson 3.8: Dangerous Use of Eval

3. AJAX Security -> 3.8 Dangerous Use of Eval

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson demonstrates the use of the JavaScript eval() function. The vulnerability is an XSS attack in the 3 digit credit card access code field such as:

```
123');alert(document.cookie);('
```

Note that the '<script>' tag is not necessary because the data is placed in the JavaScript eval() function.

Implementation

The credit card field is also vulnerable.

The ModSecurity solution is to whitelist both fields.

The regex for the 3-digit access code ('field1') is: ^\d\d\d$

The regex for the credit card number ('field2') is: ^[\d]{16,20}$

Since the solution cannot distinguish between a malicious attack and an errant entry for credit card number or the access code, a JavaScript alert box is shown to the user that closely resembles the application's error message box. But appending JavaScript to the response body is not an option because then the request would go through to the application, which we don't want. Luckily, it was discovered that since an 'eval' function calls the file 'eval.jsp', upon an error condition the response body was simply:

```
alert('Whoops: You entered an incorrect access code of "123xxx"');
```

So, the ModSecurity rule blocks the request upon an error condition and replicates the application's error message box as closely as possible.

For the solution, a new 'rulefile_00_initialize.conf' initialization file had to be made because it had junk in it for Lesson 13 and the line:

```
<LocationMatch "^/.*$">
```

throws off this lesson because it's evaluated first. The new file to use is 'rulefile_00_phase2-initialize.conf'.

The POST URI is 'http://192.168.0.5/WebGoat/lessons/Ajax/eval.jsp' with sample POST parameters:

```
field1=123');alert(document.cookie);('&field2=4128 3214 0002 1999
```

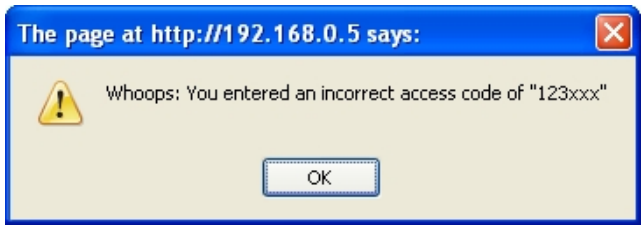
The ModSecurity configuration file 'rulefile_03-8_dangerous-eval.conf' contains only a phase 2 request section:

```
  SecRule REQUEST_URI "!^/webgoat/lessons/ajax/eval.jsp"
"t:lowercase,skipAfter:380"

  # 3 digit acess code
  SecRule &ARGS_POST:field1 "@eq 0" "nolog,skip:1"
  SecRule ARGS_POST:field1 "!^\d\d\d$" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'AJAX Security -> \
3.8 Dangerous Use of Eval: A malicious attempt may have been made to
inject \
XSS script into the 3 digit access code field',tag:'INJECTION_ATTACK', \
redirect:/_error_pages_/lesson03-8a.html"

  # credit card number
  SecRule &ARGS_POST:field2 "@eq 0" "nolog,skip:1"
  SecRule ARGS_POST:field2 "!^[ \d]{16,20}$" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'AJAX Security -> \
3.8 Dangerous Use of Eval: A malicious attempt may have been made to
inject \
XSS script into the credit card number field',tag:'INJECTION_ATTACK', \
redirect:/_error_pages_/lesson03-8b.html"

  SecAction "t:none,nolog,id:'380'"
```

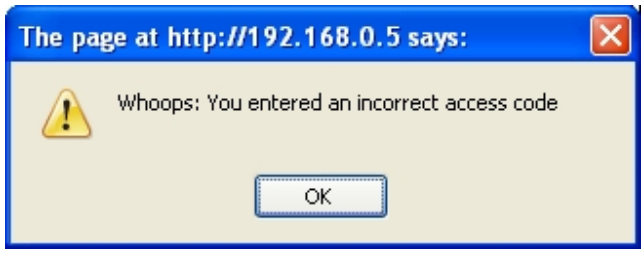
Note the uniqueness of the response from eval.jsp on an error and how the application displays it. Most of the ModSecurity solutions that block on an error condition redirects to a bland error HTML file that does not have the look and feel of WebGoat (this can be remedied, of course, with a little bit of work). In this lesson, 'lesson03-8a.html' consists simply of one line:

```
alert("Whoops: You entered an incorrect access code");'
```

Here is a screen shot of the application's error message box returned to the user for an invalid access code:



Here is the screen shot of the ModSecurity solution's error message box:



Comments

- This lesson demonstrates the use of the JavaScript 'eval' function and an interesting way that an error message can be displayed to the user.
- Unfortunately, there is no way to pass incorrect user input to the error message box at run time because the redirected error pages are loaded when Apache is started.
- ModSecurity supplies an operator 'verifyCC' to check if a string is a potential credit card number; see the reference manual for more details.

5.13 Sublesson 3.9: Insecure Client Storage

3. AJAX Security -> 3.9 Insecure Client Storage

(This sublesson was not formally solved by the project)

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson consists of 2 stages:

- Stage 1: A decrypted coupon is used to get a discount but the decryption is done in a JavaScript routine; stepping through it with FireBug or IEWatch will give away the decrypted coupon to the user who can then use it to get a discount. ModSecurity cannot provide a solution for this vulnerability; the decrypt function would have to be removed from the source code in order to begin a solution, but ModSecurity cannot alter HTTP response source code.

- Stage 2: A shopping cart is displayed: the quantity of each item is editable, but the unit price, the total price of that item, and the grand total is not editable. The exploit is to remove the HttpOnly attribute from the grand total field, then change the price to 0 and make the purchase. In real life, this is not very practical because the back end would calculate the grand total based on unit price and the number of each item.

Implementation

Intercept the POST request, which looks something like this (lines split for readability):

```
PRC1=69.99&QTY1=4&TOT1=279.95&PRC2=27.99&QTY2=0&TOT2=0&PRC3=1599.99&QTY3
=2&
TOT3=3199.98&PRC4=299.99&QTY4=0&TOT4=0&SUBTOT=3479.93&GRANDTOT=3479.93&
field2=4128+3214+0002+1999&field1=&SUBMIT=Purchase
```

Then call a Lua script that pulls all of the relevant POST parameters, tallies up from the individual item total what the grand total should be, and then compares it with the grand total that was submitted.

5.14 Sublesson 4.2: Forgot Password

4. Authentication Flaws -> 4.2 Forgot Password

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This lesson illustrates using a weak secret password, favorite color, which can be guessed eventually. The project member chose to take a broader view in mitigating this vulnerability and using a couple of features of ModSecurity in ways not done in other lessons.

A configurable account lockout mechanism (maximum retry count) is implemented using Lua persistence. The per-user configurable parameters are: (1) the number of retries allowable before the account is locked out for further use; (2) the interval (in minutes) that time has elapsed after being locked out that the account is reactivated.

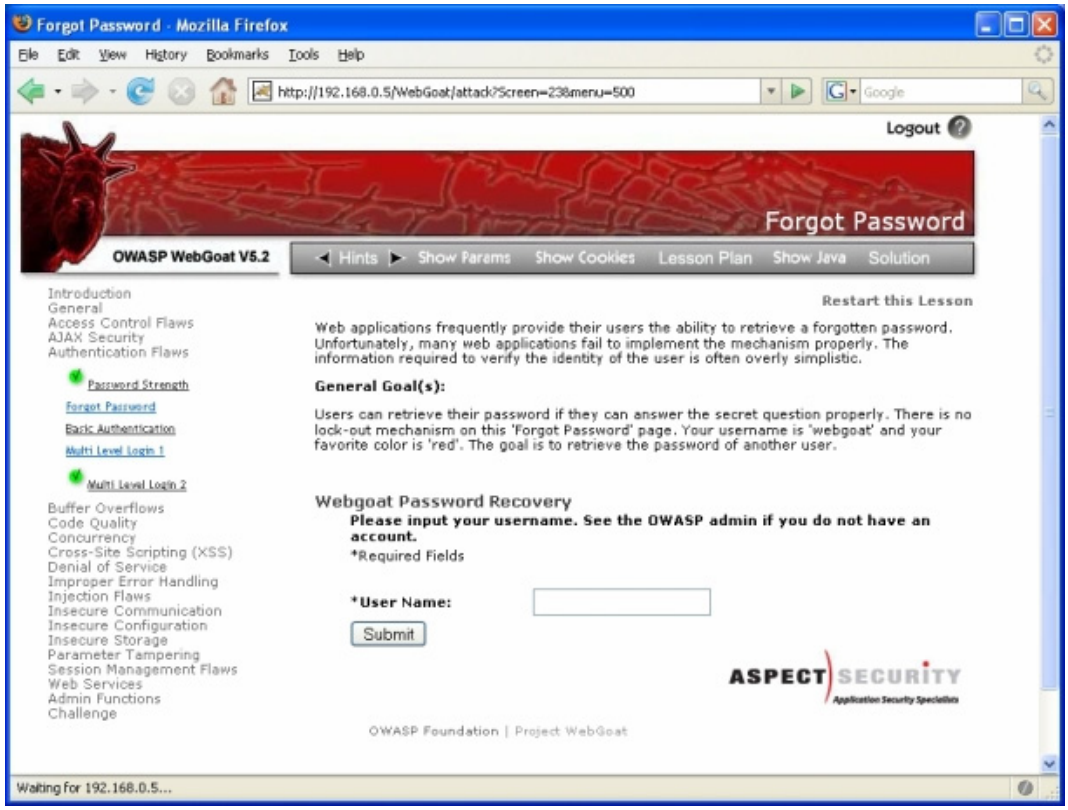
Appending JavaScript to the end of the response body - a very unsung useful feature of ModSecurity - is also used to alter the response back to the user.

We will show the implementation, then after discuss how this solution can be extended and used as a temporary authentication mechanism.

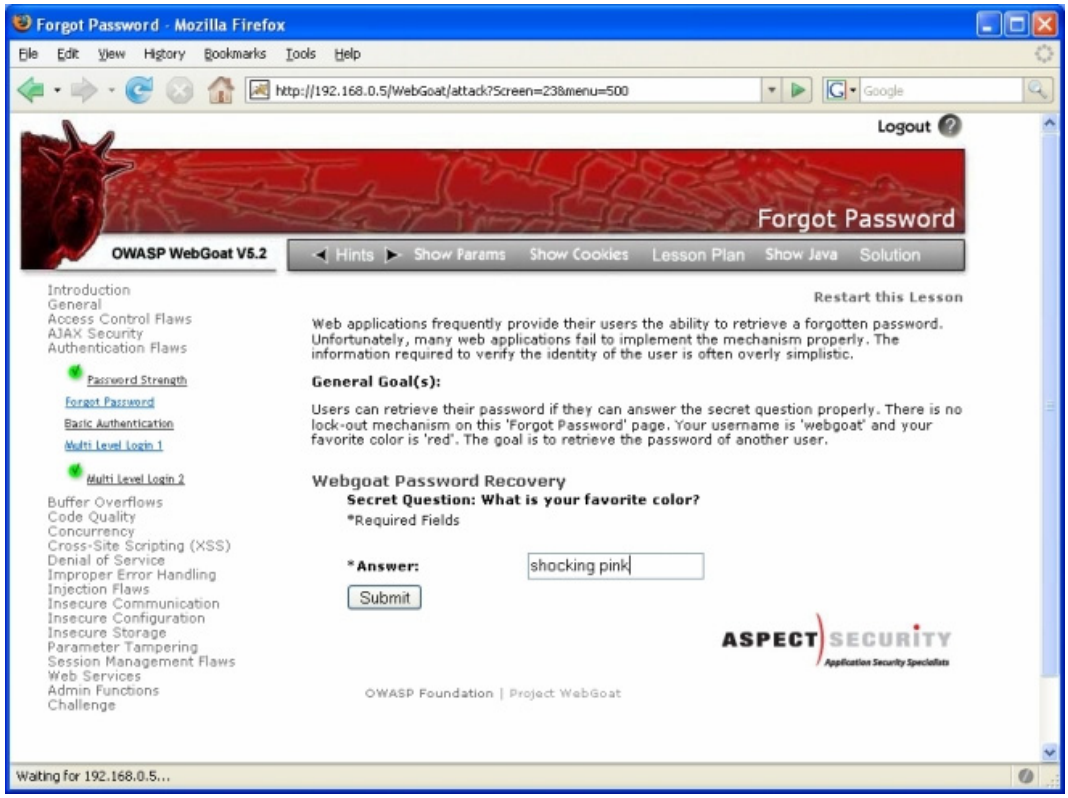
Implementation

First, let's go through the steps.

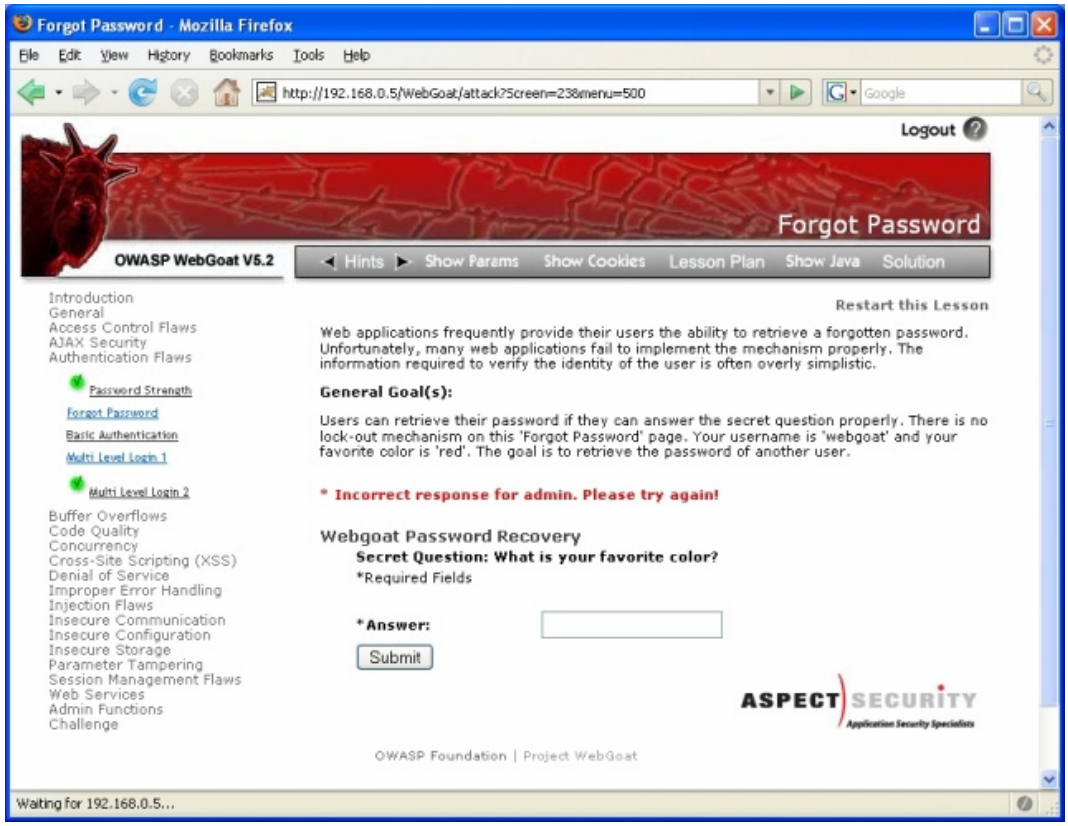
At the beginning of the lesson, the user is asked for their user name:



The user is asked to provide an answer to the secret question:



If the answer is wrong, the user receives an error message and is prompted again:



There is no limit on the number of retries, a scenario which exists in many web applications. A hint is given that the administrator may have an account name of 'admin' and therefore eventually the admin's secret question can be guessed.

The first step of the solution is a little tricky: we have to keep track of the user name from the request/response cycle of the first page to the 2nd page that guesses the answer to the secret question. In the real world, to take into account concurrency, we would set a cookie for this scenario, but for our purpose we we to keep it simple and show the main points of the solution.

The user name is displayed when an incorrect response; e.g. '* Incorrect response for admin. Please try again!'. So at this point we have the user name and know that they have guessed the secret question; for our purpose, the retry count is technically still 0.

We are using Lua scripts and Lua persistence.

The Lua code to extract the user name from this string is:

```
_, _, currentuser = string.find(tbuff, "%* Incorrect response for
(%w+)%.")
```

Let's look at the data file format:

```
Entry{
  username = "global",
  retry = 2,
  interval = 15,
  lockedout = "no",
  current = "no"
}

Entry{
  username = "webgoat",
  retry = 0,
  interval = 0,
  lockedout = "no",
  current = "no"
}

Entry{
  username = "admin",
  retry = 0,
  interval = 0,
  lockedout = "no",
  current = "no"
}
```

The 'global' entry contains the configurable values for the number of retries that are going to be allowed and the interval (in minutes) that needs to elapse before the user's account is reactivated. Here, 2 retries are allowed, and if the user's account is locked and the time interval between retry attempts is 15 minutes, the account will be reactivated.

Where the Lua scripts have to make changes to the data file, a read/write operation is performed. In one case, only a read operation is necessary. Keep in mind that the data file can be edited at any time (if using *nix, be sure to restore the original owner and group name) to observe and experiment with the values and results.

The Lua script responsible for handling the HTTP responses for this lesson is 'secret-question-result_04-2.lua'.

After obtaining the user name, we loop through all of the users in the data file:

```
'dofile(datafile)'
```

This calls the function 'Entry'; first that particular user's data is saved to local variables:

```
  function Entry (b)
    local username = b.username
    retry = b.retry
    interval = b.interval
    local lockedout = b.lockedout
    local current = b.current
    local currentuser = currentuser
```

Next:

```
  if username == currentuser then
    current = "yes"
```

```
    retry = retry + 1
    if lockedout == "yes" then
      str1 = string.format("Luascript: current user '%s' account is
locked!", username)
      m.log(9, str1)
      retstr = str1
    end
  else
    current = "no"
  end
```

The user is set to current or not. If it is the current user, we increase the retry count and if the account is locked out, we set the script's return screen to non-nil, which will trigger the SecRuleAction in phase 4 of this lesson's ModSecurity configuration file; we will go over this later in this lesson - view either 'rulefile_04-2a_forgot-password.conf' or 'rulefile_04-2b_forgot-password.conf'.

The last step of the loop (the function 'Entry') is to add the user's updated information to a string buffer:

```
 -- build buffer and write back to disk
outstr = string.format("Entry{\n  username = \"%s\",\n  retry = %d,\n  \
    interval = %d,\n  lockedout = \"%s\",\n  current = \"%s\"\n}\n\n", \
    username, retry, interval, lockedout, current)
  tbuff = tbuff .. outstr
```

When the function is over, we write the file:

```
  local fh2 = io.open(datafile, "w+")
  if fh2 == nil then    -- don't fail open
    str1 = string.format("Luascript: Error in creating file: %s;\n",
datafile)
    m.log(9, str1)
    retstr = str1
  else
    fh2:write(tbuff)
    fh2:flush()
    fh2:close()
  end
```

Finally, the return value is returned to SecRuleAction:

```
  if retstr == "" then
    retstr = nil
  end
  return retstr
```

A nil value means nothing bad happened.

Now, let's take a look how we process the HTTP request. The Lua script, which is heavily commented, is 'secret-question-guess_04-2.lua'.

It is necessary to distinguish between 2 different types of requests that are received.

The 1st request is when the user enters the user name. The POST body parameters '`Username=admin&SUBMIT=Submit`' are retrieved:

```
  loginuser = m.getvar("ARGS_POST.Username", "none")
  local submitvalue = m.getvar("ARGS_POST.SUBMIT", "none")
```

The function for processing this request, Entry0, is called if the proper conditions are met:

```
  if loginuser ~= nil and submitvalue ~= nil and submitvalue == "Submit"
then
    str1 = string.format("Luascript: Entering Entry0 with user
name: %s\n", loginuser)
    m.log(9, str1)
    Entry = Entry0
    dofile(datafile)
    return retstr
  end
```

The 2nd request is when the user is making a guess for the secret question. If no other sublessons of the lesson are being used - only 'rulefile_00_initialize.conf' and either 'rulefile_04-2a_forgot-password.conf' or 'rulefile_04-2b_forgot-password.conf' - then we don't need to filter to call the 2nd function. If it is desirable to filter, add the conditional statement as done previously with the POST parameters as '`Color=green&SUBMIT=Submit`'.

```
  Entry = Entry1
  dofile(datafile)
```

Let's examine what each function does.

'Entry0' loops and processes each user in the data file. First, each entry's values are copied to local variables as previously shown.

Next, we get the configurable value from the user 'global' that we need, 'interval'; note that 'global' always has to be the first entry in the data file.

```
    if username == "global" then
      g_interval = interval
    end
```

The next section of the function is:

```
    if username == loginuser and lockedout == "yes" then
      num1 = os.time()
       if num1 >= (interval + g_interval*60) then
         str1 = string.format("Luascript: User %s is locked out but
global \
             interval has been exceeded\n;", username)
         m.log(9, str1)
         retstr = nil
       else
```

```
         str1 = string.format("Luascript: Entry0 - User %s is locked
out!\n;", username)
         m.log(9, str1)
         retstr = str1
       end
    end
```

It is better to only read the data file if possible, which is done here. If the user is locked out but the interval between lock out time and now has been exceeded, we will let the user through and in the 2nd function deal with this condition so nil is returned in the function and the main body (to return nil to the SecRuleAction). If the user is locked out, we return non-nil (the string message is recorded in the audit log and the debug file, irrespective of the explicit log entry '`m.log(9, str1)`' to the debug file), the SecRuleAction will trigger, and the user will receive a message that they have been locked out (which will be shown later).

For the 2nd function, 'Entry1', remember that we are on the page that is guessing the color. In this function, though, we will make changes to the data file according to the results.

First, retrieve the 'global' user's configuration parameters but this time we need the retry maximum count also:

```
  if username == "global" then
    g_retry = retry
    g_interval = interval
  end
```

Next:

```
  if current == "yes" then
    if lockedout == "yes" then
      -- check if time interval exceeded; if so, unlock account
      num1 = os.time()
      if num1 >= (interval + g_interval*60) then
        lockedout = "no"
        retry = 0
        interval = 0
      else
        -- question: do we want to set a new interval to the current
time? IMO, yes
        interval = os.time()
        retstr = "Luascript: Entry1 - account already locked!"
      end
    end

    if lockedout == "no" then
      -- have to be real careful here that both are integers;
      -- if not, then function will return suddenly with nil
      if retry >= g_retry then
        lockedout = "yes"
        interval = os.time()  -- record the second that user was locked
        retstr = "Luascript: account is now locked!"
```

```
      end
    end
  end
```

If the user is not the current user, nothing changes.

If the user is current, check if their account is currently locked.

If yes, check if the max interval time has been exceeded; if yes, then set the user lockout to "no", and reset the retry and interval values to 0; if no, reset the interval time to the current time (optional) and set 'retstr' so that it is denoted that the user is locked out and the SecRuleAction will be triggered.

If the account is not locked out, check if the retry count has been reached; if so, mark the account as locked out and reset the interval.

Note, that a conditional '`if lockedout == "no", then, else`' was not used. If the account was originally locked out but changed to not locked out, that user is processed in the '`lockedout == "no"`' section as a safety check (and if one wants to add other logic here).

Also, note that the return value to SecRuleAction as currently implemented only returns boolean: either nil (nothing bad happened) or non-nil (changing ModSecurity to accept nil or a string as a return value should be possible because the return string is written to the log files). In our case, the bottom line is the same: the user is locked out, but the causes are different.

Finally, at end of the function each user's values appended to a buffer, then after exiting the function the buffer is written back for a new data file (the code for this has been previously shown).

We jump up now to the ModSecurity rules. Two alternatives have been given, 'rulefile_04-2a_forgot-password.conf' and 'rulefile_04-2a_forgot-password.conf'; they differ as to how to notify the user that they have been locked out.

First, 'rulefile_04-2a_forgot-password.conf':

Recall that the HTTP request triggers all of the processing when a user has made an incorrect guess; the error message '`* Incorrect response for admin. Please try again!`' (see Figure 3 at the beginning of this page) is our cue that a certain user has failed at making a correct guess.

Here is the rule:

```
  SecRule TX:MENU "!@eq 500" "phase:4,t:none,pass,skip:1"

  # parse response body and process Lua script
  SecRuleScript "/etc/modsecurity/data/secret-question-result_04-2.lua"
\
    "phase:4,t:none,log,auditlog,allow,msg:'Luascript: Writing RESPONSE
\
    BODY extracting user name from error message in secret-question-
result_4-2.lua'"
```

We don't care about the result of the script (nil or non-nil) because we do the heavy lifting in the HTTP request Lua script. The rules for this are:

```
  SecRule ARGS:menu "!@eq 500" "phase:2,t:none,skip:2"

  # action is triggered if script returns non-nil value
  SecRuleScript "/etc/modsecurity/data/secret-question-guess_04-2.lua" \
    "phase:2,t:none,log,auditlog,deny,severity:3,msg:'Authentication
Flaws \
    - 4.2 Forgot Password',tag:'PASSWORD_RETRY', \
    redirect:/_error_pages_/lesson04-2.html"

  SecAction "phase:2,allow:request,t:none,log,auditlog, \
    msg:'Luascript: user account not locked out'"
```

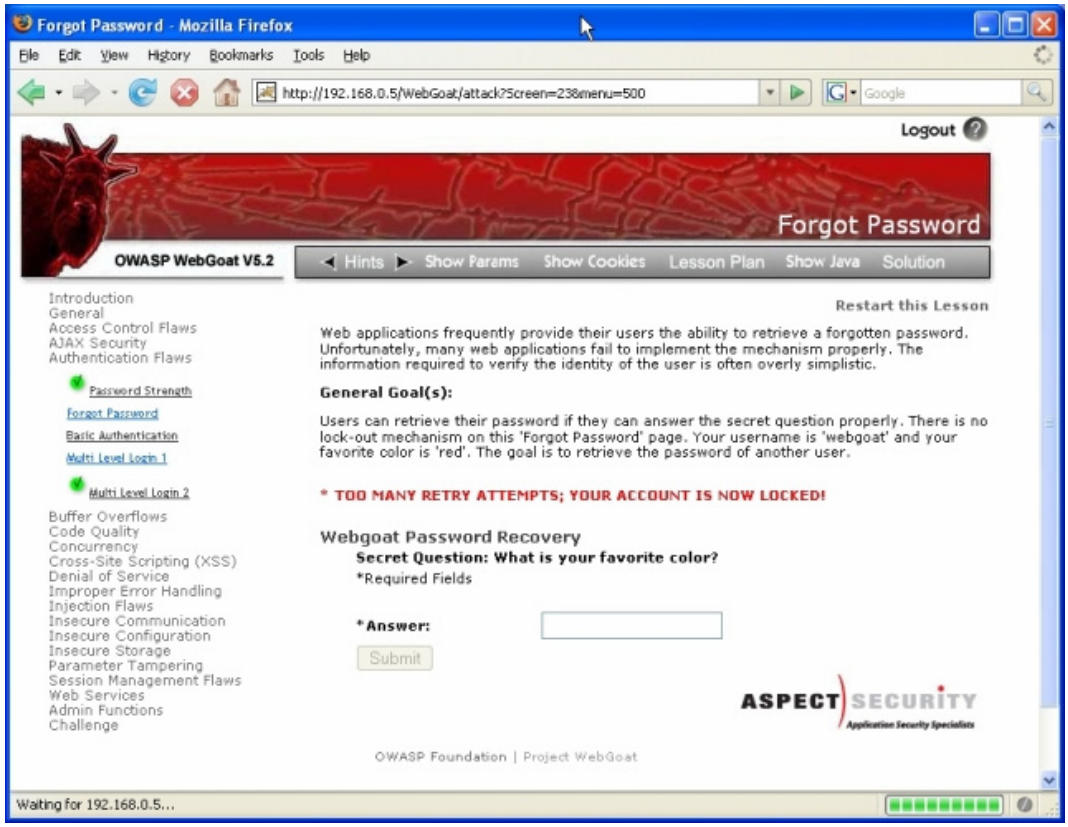
(Note: previously, it was explained why the lessons are filtered by menu number and the reasons for using '`ARGS:menu`' in Phase 2 and '`TX:MENU`' in Phase 4)

Very simply, if the user's account is locked out, they get redirected to an error page.

The 2nd ruleset, rulefile_04-2b_forgot-password.conf, offers a more elegant solution. Inside the Lua script, a non-nil value is returned when the user's account is locked out. Instead of passing through the response body and processing the result in the request body in 'rulefile_04-2a_forgot-password.conf', we do the opposite:

```
  SecRule TX:MENU "!@eq 500" "phase:4,t:none,pass,skip:1"

  # parse response body and process Lua script
  SecRuleScript "/etc/modsecurity/data/secret-question-result_04-2.lua"
\
    "phase:4,chain,t:none,log,auditlog,allow,msg:'Luascript: Writing
RESPONSE \
    BODY and appending javascript in secret-question-result_4-2.lua'"

  SecRule RESPONSE_CONTENT_TYPE "^text/html" \
    "append:'<script language=\"JavaScript1.2\"
type=\"text/javascript\"> \
    var idvar = document.getElementById(\"message\"); idvar.innerHTML =
\"\"; \
    idvar.innerHTML = \"<BR> * TOO MANY RETRY ATTEMPTS; YOUR ACCOUNT IS
NOW \
    LOCKED!\"; document.form.SUBMIT.disabled = true;</script>'"
```

The JavaScript appended to the response body, instead of redirecting to an error page, looks like this:



Compare this with Figure 3. The original error message, in bright red color, has been replaced with '`* TOO MANY RETRY ATTEMPTS; YOUR ACCOUNT IS NOW LOCKED!`' and the Submit button has been disabled. The advantages of this approach are: (1) the user stays in "the same place" in the web site; (2) the look and feel of the web site is retained; the look and feel would have to be incorporated in a new error HTML page.

Comments

In the real world, a file locking mechanism would need to be placed on the data file which could be implemented easily by creating or checking for an empty file like 'file.lock'. There doesn't seem to be an elegant 'sleep' or 'wait' capability in Lua script, so the rare occurrence that a request file lock is already used, an error message would have to be returned to the user saying "please try again".

What has been shown in this sublesson?

1. The use of Lua persistence
2. A configurable account lock out (maximum retry count) mechanism
3. Appending JavaScript to a response body to alter the HTML page's content and behavior.
4. A business logic flaw has been mitigated by a Web Application Firewall

The basic concepts demonstrated here can be extended and be used in real world examples, such as:

1. A web site's authentication has no maximum retry count or account lock out mechanism.
2. A web site (or areas of a web site), previous open to the public, requires protection via an authentication mechanism.

The Lua scripts can be modified to tie into other authentication credential storage mechanisms (if the capability is supported in the language).

5.15 Sublesson 4.4: Multi Level Login 1

5.16 Sublesson 4.5: Multi Level Login 2

4. Authentication Flaws -> 4.4 Multi Level Login 1

4. Authentication Flaws -> 4.5 Multi Level Login 2

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

In both lessons, the attacker alters a value of a hidden field; for the first one it is: '`<input name='hidden_tan' type='HIDDEN' value='2'>`'

In order to solve this, the hidden field values have to be saved in the response body and then, when the request is sent, compared to see if it has been altered. It might be technically possible to use the session collection functionality of ModSecurity, parse the response body with a regular expression and store the hidden values, then do the same in the following request and make the comparison. Instead, Lua scripts are used; see Section 4.3 'Using the Lua scripting language' for details on how Lua is used to detect if input values - in this case hidden ones - have been altered on the client side.

Implementation

The lesson is mitigated in the ruleset 'rulefile_04_authentication-flaws.conf':

```
SecRule ARGS:menu "!@eq 500" "phase:2,t:none,skip:2"
```

```
  # following takes care of 4.3 & 4.4 - Multi Level Login
  # action is triggered if script returns non-nil value
  SecRuleScript "/etc/modsecurity/data/read-hidden-values_04.lua" \
    "phase:2,t:none,log,auditlog,deny,severity:3, \
    msg:'Parameter Tampering; Hidden field', \
    tag:'PARAMETER_TAMPERING',redirect:/_error_pages_/lesson04-4.html"
  SecAction "phase:2,allow:request,t:none,log,auditlog, \
    msg:'Luascript: hidden field not altered or does not exist'"

############################################################

  SecRule TX:MENU "!@eq 500" "phase:4,t:none,pass,skip:1"

  # parse response body and write hidden values to file
  SecRuleScript "/etc/modsecurity/data/write-hidden-values1.lua" \
    "phase:4,t:none,log,auditlog,allow,msg:'Writing RESPONSE BODY \
    & parsed input fields to file using luascript'"
```

Both rules are only processed only when in this lesson (menu=500).

The 2nd rule uses the lua script 'write-hidden-values1.lua' in Phase 4 to write HTML input element values to a data file from every response request, in this format:

```
Entry{
  name = "hidden_tan",
  type = "HIDDEN",
  value = "2"
}

Entry{
  name = "tan",
  type = "TEXT",
  value = ""
}

Entry{
  name = "Submit",
  type = "SUBMIT",
  value = "Submit"
}
```

For every request, the lua file 'read-hidden-values_04.lua' gets the hidden field value, '`hidden_tan`', compares it with the value stored in the data file, and matches the SecRuleScript if the field has been altered.

The same solution is used for Multi Level Login 2, only in this case the hidden field name is '`hidden_user`'.

5.17 Sublesson 6.1: Discover Clues in the HTML

6. Code Quality -> 6.1 Discover Clues in the HTML

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

The solution to this lesson is not to allow any admin or login credentials that have been placed in HTML comments to reach the user.

The guilty code is:

```
<!--
        FIXME admin:adminpw
  -->
```

Implementation

The lesson is mitigated by the ruleset 'rulefile_06_code-quality.conf':

```
  SecRule TX:MENU "!@eq 700" "phase:4,t:none,pass,skip:2"

  SecRule RESPONSE_BODY \
    "<!--[ \r\n\t]*?(.*)?(?i:adm(in)?|pwd|passw(or)?d)(.*)?[ \r\n\t]*?--
>" \
    "phase:4,t:none,log,auditlog,deny,severity:3,msg:'Authentication
Credentials \
    in HTML
comment',id:'61',tag:'LEAKAGE',redirect:/_error_pages_/lesson06-1.html"

  SecAction "phase:4,allow,t:none, \
    msg:'Returning; nothing bad on this page (rulefile_06-1).'"
```

Notice that the '`TX:MENU`' variable, which is set in the rulefile_00_initialize.conf, is used because using '`ARGS:menu`' will not be accurate as it goes out of scope when leaving Phase 2.

Comments

The regex used for this solution can give false positives. It is okay for this lesson, but in 'Lesson 4.2 Authentication Flaws -> Forgot Password', this string of text also matches:

```
<!-- Start Instructions -->
...
Users can retrieve their password if they can answer the secret question
properly.
```

```
...
<!-- Stop Instructions -->
```

The regex works as intended both in The Regex Coach and in Expresso. Instead of going through a laborious process each attempt just to see if the regex works as intended, it would be nice to have a utility that calls the exact same PCRE API that ModSecurity currently calls (see the source code file 'msc_util.c') so that the regex can be tested completely outside of ModSecurity.

Reviewer comments

Information leakage through html comments is an interesting challenge from a WAF perspective. Here are a few of them:

1. No False Negatives: We don't want to miss any comments so we need to have an accurate html parser in order to deal with all possible formats, etc… Something like mod_html_proxy or mod_publisher would help in this area.

2. All Comments != Bad: HTML comments, in and of themselves, are not bad. It is only the ones that are leaking sensitive info. The question is what is sensitive? In the context of this WebGoat lesson, the sensitive info is the admin credentials. Unfortunately, if we try to address the underlying issue and not just for this lesson, we have no idea of the exact format of the data being presented.

3. Blocking vs. Scrubbing: Assuming we can reliably detect comments but not necessarily any sensitive data leaks, is this category of issue severe enough that you would want to block the entire response? Scrubbing inbound data for attacks and sending it onto the application is more risky than scrubbing outbound data for info leakages and then sending it onto clients. Once again, a module such as mod_proxy_html even has a remove comments directive that will handle this issue.

The best option that I see is to actually remove all comments from outbound data, however seeing as ModSecurity cannot currently do this then the next best thing is to alert and/or block. In order to make our RegEx more generic and to look for other comment formats we should account for C-style, JavaScript and one-liner comments:

C-style:

```
/* This is a comment */

/* C-style comments can span
as many lines as you like,
as shown in this example */
```

JavaScript/C++ style:

```
// This is a one-line comment
```

Also a one-liner with just the beginning:

```
<!-- This is treated as a one-line JS comment
```

The updated Mod rule would look like this:

```
SecRule RESPONSE_BODY "(<!--
[ \r\n\t]*?(.*)?(?i:adm(in)?|pwd|passw(or)?d) \
(.*)?[ \r\n\t]*?(-->)?|\/\*.*\*\/|\/\/)" \
    "phase:4,t:none,log,auditlog,deny,severity:3, \
msg:'Authentication Credentials in HTML comment',id:'61',tag:'LEAKAGE',
\
redirect:/_error_pages_/lesson06-1.html"
```

It was tested a bit and seems to work but the RegEx could probably be tweaked a bit more.

5.18 Sublesson 7.1: Thread Safety Problem

7. Concurrency -> 7.1 Thread Safety Problem

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson demonstrates a thread safety problem: when 2 users login almost simultaneously, the first user receives the second user's data.

To solve this lesson, at first an attempt to use ModSecurity rules was made, but figuring out how global persistence works and debugging got too time-consuming for an inexperienced ModSec user, so Lua was chosen.

When the first user logs in, a lock is set and the 2nd user is not allowed to log in until the first user's request returns.

Implementation

For this lesson solution, we will walk through the development process from a standalone Lua script on Windows to the ModSecurity solution on Linux.

Retrieved from the web proxy are the POST body parameters:

```
username=jeff&SUBMIT=Submit
```

A class and method is used for standalone development and testing that closely resembles the ModSecurity implementation of writing to the debug log file:

```
-- BEGIN: CUT HERE WHEN IN MODSEC
-- for standalone testing, simulate ModSec functions
-- have to replace ':' with '.'; e.g. 'm:log' with 'm.log'

ModSec = {}

function ModSec:new (o)
        o = o or {}
  setmetatable(o, self)
  self.__index = self
  return o
end

function ModSec:log (loglevel, msg)
  print(msg)
end

m = ModSec:new()
-- END: CUT HERE WHEN IN MODSEC
```

So, 'm:log(9, msg2)' in the standalone version, which prints the debug messages to the command line, needs to be edited to 'm.log(9, msg2)' for writing to the ModSecurity debug log.
The project also standardized on debug log messaging:

```
  msg0 = "Luascript (request-on_07-1.lua): "
  msg2 = ""
```

A debug log message would be assembled like this:

```
  msg1 = string.format("Request is already pending; user name is '%s'",
username)
  msg2 = msg0 .. msg1
  m:log(9, msg2)
```

This ensures a consistent format and it makes searching through a debug log easier.

The solution is started using a standalone Lua script and simulating an HTTP request which is simple because we only have to run the script to increment the number of users logging in.

The format of the data file 'lesson07-1.data' is:

```
Entry{
  requestpending = 0
}
```

The source code to process a request (minus the debug messages) from 'request-on_07-1.lua' is:

```
  local username = "jsnow"

  function Entry (b)
    local ecount = b.requestpending

    if ecount >= 1 then
      msg1 = string.format("Request is already pending; user name is
'%s'", username)
      retval = msg1
    end

    -- increment count and build string to write back to file
    ecount = ecount + 1
    outstr = string.format("Entry{\n  requestpending = %d\n}\n\n",
ecount)
  end

  dofile(datafile)
```

The 'dofile' function only makes one pass because there is only one entry in the data file.

Test the standalone program (in this case on Windows) by running from the command line:

```
C:\lua\bin> lua request-on_07-1.lua
```

Each time the program is run, the number of logged in users should increment by 1 in the data file until the maximum allowed is reached.

If the number of users logging in is greater or equal to 1, the function returns with a non-nil value, which causes a match for the ModSecurity rule. When the count is incremented, the data file with the new count is written back to disk:

```
  local fh2 = io.open(datafile, "w+")
  fh2:write(outstr)
  fh2:flush()
  fh2:close()
```

Once the program works as intended, modify it and integrate it into ModSecurity on Linux.

To convert to ModSecurity, we only have to: 1. Cut out the log class and method 2. Change m:log to m.log 3. replace the hard-coded 'username' with:

```
  local username = m.getvar("ARGS_POST.username", "none")
```

4. Remove the 'main()' call at the end of the file.

It's that simple!

The Modsecurity rules are in the file 'rulefile_07-1_thread-safety.conf'.

For the request:

```
  SecRule ARGS:menu "!@eq 800" "phase:2,t:none,skip:4"
  SecRule &ARGS_POST:SUBMIT "@eq 0" "nolog,skip:3"
  SecRule &ARGS_POST:username "@eq 0" "nolog,skip:2"

  # action is triggered if script returns non-nil value
  SecRuleScript "/etc/modsecurity/data/request-on_07-1.lua"
"phase:2,t:none,log,auditlog, \
deny,severity:3,msg:'Luascript: Concurrency -> Thread Safety: request is
pending', \
tag:'CONCURRENCY',redirect:/_error_pages_/lesson07-1.html"
  SecAction "phase:2,allow:request,t:none,log,auditlog, \
msg:'Luascript: Concurrency -> Thread Safety: request is NOT pending'"
```

For the HTTP response, the user count is decremented each time so running another Lua script, 'request-off_07-1.lua', simulates that in standalone mode. You can view the source - it's exactly the same as above except it decrements the count instead of incrementing the user count.

Once the Lua program is working in standalone mode, convert it and move it over to ModSecurity.

The ModSecurity rules are:

```
  SecRule TX:MENU "!@eq 800" "phase:4,t:none,pass,skip:1"

  # this will decrement the request-in-progress for any request in
Lesson 7
  #   but will not go less than zero
  SecRuleScript "/etc/modsecurity/data/request-off_07-1.lua" \
"phase:4,t:none,log,auditlog,allow,msg:'Luascript: Concurrency -> \
Thread Safety: in RESPONSE; decrementing requests by 1'"
```

The best way to test the Lua scripts when integrating into ModSecurity is to first comment out the response rules, then when the request rules are working, do the opposite; then enable both sets once the request set of rules are working properly.

When enabled and doing the WebGoat sublesson, Dave logs in first:

Quickly followed by Jeff, who gets blocked and receives this message:

Comments

- This solution shows the process of developing a standalone Lua script on Windows and then integrating it into ModSecurity on Linux.

5.19 Sublesson 7.2: Shopping Cart Concurrency Flaw

7. Concurrency -> 7.2 Shopping Cart Concurrency Flaw

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson demonstrates a shopping cart concurrency flaw: there is a window between when the purchase is started and the confirmation, which allows the shopping cart to be updated and get more expensive items for the same price. The user chooses an item, clicks on 'Purchase', opens another window, chooses more expensive items, clicks 'Update Cart', goes back to the original window and clicks 'Confirm', and the user has bought more items but based on the original price.

To solve this lesson, place a lock on the transaction once the purchase is initiated until it is confirmed or canceled. When a buyer has clicked on Purchase, no other transactions (Purchase or Update Cart) are allowed until Confirm or Cancel is pressed.

Implementation

The relevant data gathered from the web proxy is:

```
menu=800
QTY1=0&QTY2=0&QTY3=0&QTY4=0&SUBMIT=Purchase
QTY1=0&QTY2=0&QTY3=1&QTY4=0&SUBMIT=Update+Cart
CC=5321+1337+8888+2007&PAC=111&SUBMIT=Confirm
CC=5321+1337+8888+2007&PAC=111&SUBMIT=Cancel
```

The data file, lesson07-2.data, looks like this with initial values:

```
Entry{
  state = "none",
  pending = "no",
  time = 0
}
```

'State' changes when a button is pushed. The original intention was to implement a timeout also, but time did not permit this and, besides, a timeout should already be implemented in the application.

The following is what should happen during different states:

- When the Purchase button is pushed and pending is "no", set the datafile parameters as such: state = "purchase"; pending = "yes"; time = 0.
- If the Update Cart button is pushed while pending, return a non-nil error message.
- If Cancel or Confirm is pushed, set pending to "no".
- If there is a page request with no POST parameters, reset pending to "no".

Only the HTTP request is being used and the relevant parts of the Lua script, shopping-cart_07-2.lua, is as follows:

```
datafile = "/etc/modsecurity/data/lesson07-2.data"
retval = nil

function main()
  local submit_type = m.getvar("ARGS_POST.SUBMIT", "none")

  function Entry (b)
    local state = b.state
```

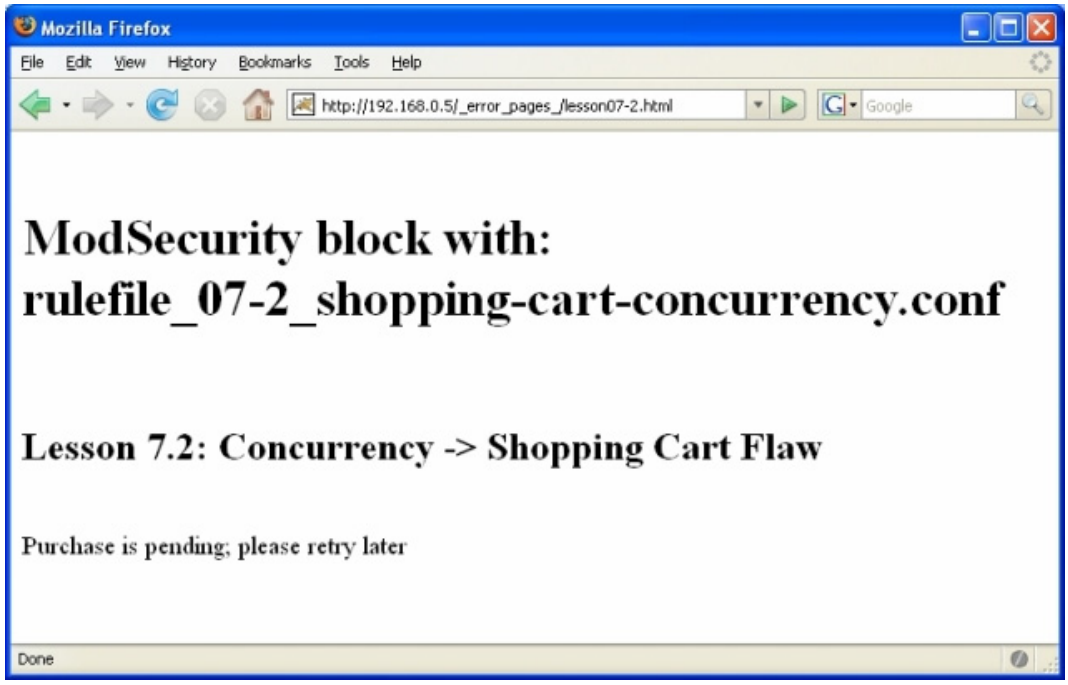
```
    local pending = b.pending
    local time = b.time

    if b.state == "none" then
      if submit_type == "Purchase" then
        state = "Purchase"
        pending = "yes"
        time = 0
      end
    elseif submit_type == "Purchase" then
      state = "Purchase"
      pending = "yes"
      time = 0
    elseif submit_type == "Update Cart" then
      state = "Update Cart"
      time = 0
      if b.pending == "yes" then  -- error; block this action
        str1 = string.format("Error: attempt to update cart when
purchase is pending;\n")
        retval = str1
      end
    elseif submit_type == "Cancel" or submit_type == "Confirm" then
      state = submit_type
      pending = "no"
      time = 0
    end

    -- build the data file - this should all be on one line
    outstr = string.format("Entry{\n  state = \"%s\",\n  pending =
\"%s\",
      \n  time = %d\n}\n\n", state, pending, time)
  end

  dofile(datafile)

   -- write back to file
  local fh2 = io.open(datafile, "w+")
  fh2:write(outstr)
  fh2:flush()
  fh2:close()

  return retval
end
```

Here is the ModSecurity rule file, rulefile_07-2_shopping-cart-concurrency.conf:

```
  SecRule ARGS:menu "!@eq 800" "phase:2,t:none,skip:2"
  SecRule &ARGS_POST:SUBMIT "@eq 0" "nolog,skip:1"

  # action is triggered if script returns non-nil value
  SecRuleScript "/etc/modsecurity/data/shopping-cart_07-2.lua" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'Luascript: Concurrency
-> \
Shopping Cart: purchase is pending',tag:'CONCURRENCY', \
redirect:/_error_pages_/lesson07-2.html"
```

```
  SecAction "phase:2,allow:request,t:none,log,auditlog,msg:'Luascript: \
Concurrency -> Shopping Cart: purchase is NOT pending'"
```

When a transaction is pending, the user will get this message:



Comments

- This lesson solution shows how Lua can be used to handle application state; in this case, a purchase that utilizes a shopping cart.

5.20 Lesson 8: Cross-Site Scripting (XSS)

Addressing XSS attacks and vulnerabilities

XSS attacks and vulnerabilities can be addressed externally in 3 ways:

1. Whitelist validation: Use a standard input validation mechanism to validate all input data for length, type, syntax, and business rules before accepting the data to be displayed or stored. Use an "accept known good" validation strategy. Reject invalid input rather than attempting to sanitize potentially hostile data. Example: the “item_number” parameter should only accept digits between 1 and 5 characters in length however it is vulnerable to XSS:

```
http://www.example.com/cgi-
bin/product_search_query.php?item_number=<script>alert('XSS')</script>
```

The following custom ModSecurity 2.5 rule can provide proper positive security input validation for this parameter:

```
<Location /cgi-bin/product_search_query.php>

SecRule &ARGS_POST_NAMES "!@eq 0" "phase:2,t:none,deny,log,auditlog, \
msg:'Input Validation Alert - Arguments in Post
Payload',logdata:'%{MATCHED_VAR}'"

SecRule &ARGS_GET_NAMES:item_number "!@eq 1"
"phase:2,t:none,deny,log,auditlog, \
msg:'Input Validation Alert - Multiple item_number
parameters.',logdata:'%{MATCHED_VAR}'"

SecRule ARGS_GET:item_number "!^\d{1,5}$"
"phase:2,t:none,deny,log,auditlog, \
msg:'Input Validation Alert - Data not in the correct
format.',logdata:'%{MATCHED_VAR}'"

</Location>
```

This rule set will help to prevent evasion attempts by ensuring that there is only 1 argument called "item_number", that it is only present within the QUERY_STRING data and that it has the proper format and length. Keep in mind that this type of input validation should also be incorporated within the application itself. The main reasons for implementing this type of positive security filter at the web application firewall layer are for general security-in-depth and also for those web applications where updating the code is either not possible or will take a very long time.

2. Blacklist validation: The Core Rule set, which is available for free from the ModSecurity website, includes a robust negative security rule set for XSS detection. The current version of the rule set uses complex logic combining two different operators; @pm set-based pattern matching used for fast pre-qualification of data to identify the existence of key XSS strings, and then @rx regular expression rules to apply advanced checks to both confirm XSS logic and exclude false positives.

```
SecRule
REQUEST_FILENAME|ARGS|ARGS_NAMES|REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS
:Referer \
"@pm jscript onsubmit copyparentfolder javascript meta onmove onkeydown
onchange onkeyup \
activexobject expression onmouseup ecmascript onmouseover vbscript: \
<![cdata[ http: settimeout onabort shell: .innerhtml onmousedown
onkeypress asfunction: \
onclick .fromcharcode background-image: .cookie ondragdrop onblur x-
javascript mocha: \
onfocus javascript: getparentfolder lowsrc onresize @import alert
onselect script onmouseout \
onmousemove background application .execscript livescript:
getspecialfolder vbscript iframe \
.addimport onunload createtextrange onload <input" \
```

```
"t:urlDecodeUni,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase,pass
,nolog,skip:1"

SecAction pass,nolog,skipAfter:959004

SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES \
"(?:\b(?:(?:type\b\W*?\b(?:text\b\W*?\b(?:j(?:ava)?|ecma|vb)|application
\b\W*?\bx-(?:java|vb)) \
script|c(?:opyparentfolder|reatetextrange)|get(?:special|parent)folder|i
frame\b.{0,100}?\bsrc)\b| \
on(?:(?:mo(?:use(?:o(?:ver|ut)|down|move|up)|ve)|key(?:press|down|up)|c(
?:hange|lick)| \
s(?:elec|ubmi)t|(?:un)?load|dragdrop|resize|focus|blur)\b\W*?=|abort\b)|
(?:l(?:owsrc\b\W*?\b \
(?:(?:java|vb)script|shell|http)|ivescript)|(?:href|url)\b\W*?\b(?:(?:ja
va|vb)script|shell)| \
background-
image|mocha):|s(?:(?:tyle\b\W*=.*\bexpression\b\W*|ettimeout\b\W*?)\(|rc
\b\W*?\b(?:(?:java|vb) \
script|shell|http):)|a(?:ctivexobject\b|lert\b\W*?\(|sfunction:))| \
<(?:(?:body\b.*?\b(?:backgroun|onloa)d|input\b.*?\btype\b\W*?\bimage)\b|
 ?(?:(?:script|meta)\b|iframe)| \
!\[cdata\[)|(?:\.(?:(?:execscrip|addimpor)t|(?:fromcharcod|cooki)e|inner
html)|\@import)\b)" \

"capture,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase,ctl:auditLo
gParts=+E,log, \
auditlog,msg:'Cross-site Scripting (XSS)
Attack',id:'950004',tag:'WEB_ATTACK/XSS', \
logdata:'%{TX.0}',severity:'2'"


SecRule REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS:Referer \
"(?:\b(?:(?:type\b\W*?\b(?:text\b\W*?\b(?:j(?:ava)?|ecma|vb)|application
\b\W*?\bx-(?:java|vb)) \
script|c(?:opyparentfolder|reatetextrange)|get(?:special|parent)folder|i
frame\b.{0,100}?\bsrc)\b| \
on(?:(?:mo(?:use(?:o(?:ver|ut)|down|move|up)|ve)|key(?:press|down|up)|c(
?:hange|lick)| \
s(?:elec|ubmi)t|(?:un)?load|dragdrop|resize|focus|blur)\b\W*?=|abort\b)|
\
(?:l(?:owsrc\b\W*?\b(?:(?:java|vb)script|shell|http)|ivescript)| \
(?:href|url)\b\W*?\b(?:(?:java|vb)script|shell)|background-
image|mocha):| \
s(?:(?:tyle\b\W*=.*\bexpression\b\W*|ettimeout\b\W*?)\(|rc\b\W*?\b(?:(?:
java|vb) \
script|shell|http):)|a(?:ctivexobject\b|lert\b\W*?\(|sfunction:))| \
<(?:(?:body\b.*?\b(?:backgroun|onloa)d|input\b.*?\btype\b\W*?\bimage)\b|
\
?(?:(?:script|meta)\b|iframe)|!\[cdata\[)|(?:\.(?:(?:execscrip|addimpor)
t| \
(?:fromcharcod|cooki)e|innerhtml)|\@import)\b)" \

"capture,t:urlDecodeUni,t:htmlEntityDecode,t:compressWhiteSpace,t:lowerc
ase, \
```

```
ctl:auditLogParts=+E,log,auditlog,msg:'Cross-site Scripting (XSS)
Attack',id:'959004', \
tag:'WEB_ATTACK/XSS',logdata:'%{TX.0}',severity:'2'"
```

While these generic XSS attack detection rules are extremely effective, they are still employ the negative security model and thus are subject to evasion issues. This is why utilizing a positive security model for input validation is the preferred method.

3. Identifying Poor/Missing Output Encoding: Ensure that all user-supplied data is HTML entity encoded before rendering in HTML, taking the approach to encode all characters other than a very limited subset. ModSecurity does not currently manipulate inbound or outbound data so it can not, by itself, be used to entity encode user data that is returned in output. While this is true, ModSecurity can be utilized to identify when web applications are failing to properly html entity encode user data in output.

The following ModSecurity rule set will generically identify both Stored and Reflected XSS attacks where the inbound XSS payloads are not properly output encoded. For Reflected XSS attacks, the rules will identify inbound user supplied data that contains dangerous meta-characters, then store this data as a custom variable in the current transaction collection and inspect the resulting outbound RESPONSE_BODY data to see if it contains the exact same inbound data. If proper outbound entity encoding of meta-characters is not utilized by the web application then the user supplied data in the response will exactly match the captured inbound data. This is effective at catching XSS attacks that utilize the "`<script>alert('XSS')</script>`" type of checks typically sent during web assessments.

For Stored XSS attacks, instead of the looking at the response body returned for the current transaction, we need to be able to identify if this user supplied data shows up in other parts of the web application. These rules address this issue by capturing the same inbound data and then storing it in a persistent global collection. On subsequent requests by any client, the response body payload is inspected to see if it contains any of the XSS strings captured in the global collection.

```
SecAction "phase:1,nolog,pass,initcol:global=xss_list"

SecRule &ARGS "@gt 0" "chain,phase:4,t:none,log,auditlog,deny,status:403,
\
msg:'Potentially Malicious Meta-Characters in User Data Not Properly
Output Encoded.', \
logdata:'%{tx.inbound_meta-characters}'"

SecRule ARGS "([\'\"\(\)\;<>/])" \
"chain,t:none,capture,setvar:global.xss_list_%{time_epoch}=%{matched_var
}, \
setvar:tx.inbound_meta-characters=%{matched_var}"

SecRule RESPONSE_BODY "@contains %{tx.inbound_meta-characters}"
"ctl:auditLogParts=+E"

SecRule GLOBAL:'/XSS_LIST_.*/' "@within %{response_body}"  \
```

```
"phase:4,t:none,log,auditlog,pass, msg:'Potentially Malicious Meta-
Characters in User \
Data Not Properly Output Encoded',tag:'WEB_ATTACK/XSS'"
```

5.21 Sublesson 8.3: Stored XSS Attacks

8. Cross-Site Scripting (XSS) -> 8.3 Stored XSS Attacks

5.22 Sublesson 8.1: Phishing with XSS

5.23 Sublesson 8.2: LAB: Cross Site Scripting

5.24 Sublesson 8.4: Reflected XSS Attacks

5.25 Sublesson 8.5: Cross Site Request Forgery (CSRF)

5.26 Sublesson 8.7: Cross Site Tracing (XST) Attacks

8. Cross-Site Scripting (XSS) -> 8.1 Phishing with XSS
8. Cross-Site Scripting (XSS) -> 8.2 LAB: Cross Site Scripting
8. Cross-Site Scripting (XSS) -> 8.4 Reflected XSS Attacks
8. Cross-Site Scripting (XSS) -> 8.5 Cross Site Request Forgery (CSRF)
8. Cross-Site Scripting (XSS) -> 8.7 Cross Site Tracing (XST) Attacks

The lessons are combined because the strategy and the ruleset to solve them are the same.

Lesson overviews

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solutions

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

Probably due to the constraints of WebGoat, it is not feasible to fully illustrate sophisticated CSRF and XST attacks so they can be mitigated with the core ruleset rules from 'modsecurity_crs_40_generic_attacks.conf' for generic XSS attacks.

However, Stage 3 of the Lab proved challenging. The employee Bruce has malicious XSS code stored in his profile and it is triggered when David views it. The malicious code is contained in an address field that is displayed in the HTML body:

```
8899 FreeBSD Drive<script>alert(document.cookie)</script>
```

The strategy to mitigate this vulnerability is to use ModSecurity as an egress filter on the response body. It is observed that WebGoat only uses JavaScript in the Head section of HTML files and that JavaScript is used only from *.js files such as:

```
<script language="JavaScript1.2" src="javascript/javascript.js" \
  type="text/javascript"></script>
```

Because we know this, we can consider any other JavaScript code as malicious and, for example, require any JavaScript to start with:

```
<script language="JavaScript1.2" src="javascript/
```

Implementation

The generic XSS attacks are mitigated from rules in the file 'rulefile_08_xss.conf':

```
  # False positives removed from following list: 'application'
  SecRule
REQUEST_FILENAME|ARGS|ARGS_NAMES|REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS
:Referer \
    "@pm jscript onsubmit copyparentfolder javascript meta onmove
onkeydown onchange \
    onkeyup activexobject expression onmouseup ecmascript onmouseover
vbscript: \
    <![cdata[ http: settimeout onabort shell: .innerhtml onmousedown
onkeypress \
    asfunction: onclick .fromcharcode background-image: .cookie
ondragdrop onblur \
    x-javascript mocha: onfocus javascript: getparentfolder lowsrc
onresize @import \
    alert onselect script onmouseout onmousemove background .execscript
livescript: \
    getspecialfolder vbscript iframe .addimport onunload createtextrange
onload <input" \

"t:urlDecodeUni,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase,deny
,log,auditlog, \
    msg:'Cross-site Scripting (XSS) Attack - whole word
matches',tag:'WEB_ATTACK/XSS', \

logdata:'%{TX.0}',redirect:/_error_pages_/lesson08a.html,severity:'3'"

  # Enable the next line if want to change to 'pass' for detect-only
mode
  # SecAction pass,nolog,skipAfter:959004

  SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES \
    "(?:\b(?:(?:type\b\W*?\b(?:text\b\W*?\b(?:j(?:ava)?|ecma|vb)| \
    application\b\W*?\bx-
(?:java|vb))script|c(?:opyparentfolder|reatetextrange) \
    |get(?:special|parent)folder|iframe\b.{0,100}?\bsrc)\b| \
```

```
on(?:(?:mo(?:use(?:o(?:ver|ut)|down|move|up)|ve)|key(?:press|down|up)| \

c(?:hange|lick)|s(?:elec|ubmi)t|(?:un)?load|dragdrop|resize|focus|blur)\
b\W* \
    ?=|abort\b)|(?:l(?:owsrc\b\W*?\b(?:(?:java|vb)script|shell|http)|ive
script)| \
    (?:href|url)\b\W*?\b(?:(?:java|vb)script|shell)|background-
image|mocha) \
    :|s(?:(?:tyle\b\W*=.*\bexpression\b\W*|ettimeout\b\W*?)\(|rc\b\W*?\b
\

(?:(?:java|vb)script|shell|http):)|a(?:ctivexobject\b|lert\b\W*?\(|sfunc
tion:))| \

<(?:(?:body\b.*?\b(?:backgroun|onloa)d|input\b.*?\btype\b\W*?\bimage)\b|
\
    ?(?:(?:script|meta)\b|iframe)|!\[cdata\[)|(?:\.(?:(?:execscrip|addim
por)t| \
    (?:fromcharcod|cooki)e|innerhtml)|\@import)\b)" \

"capture,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase,log,auditlo
g,deny, \
    msg:'Cross-site Scripting (XSS) Attack - script
tags',tag:'WEB_ATTACK/XSS', \

logdata:'%{TX.0}',redirect:/_error_pages_/lesson08b.html,severity:'3'"

  SecRule REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS:Referer \
     "(?:\b(?:(?:type\b\W*?\b(?:text\b\W*?\b(?:j(?:ava)?|ecma|vb)| \
    application\b\W*?\bx-
(?:java|vb))script|c(?:opyparentfolder|reatetextrange) \
    |get(?:special|parent)folder|iframe\b.{0,100}?\bsrc)\b| \

on(?:(?:mo(?:use(?:o(?:ver|ut)|down|move|up)|ve)|key(?:press|down|up)| \

c(?:hange|lick)|s(?:elec|ubmi)t|(?:un)?load|dragdrop|resize|focus|blur)\

b\W*?=|abort\b)|(?:l(?:owsrc\b\W*?\b(?:(?:java|vb)script|shell|http)|ive
script)| \
    (?:href|url)\b\W*?\b(?:(?:java|vb)script|shell)|background-
image|mocha):| \
    s(?:(?:tyle\b\W*=.*\bexpression\b\W*|ettimeout\b\W*?)\(|rc\b\W*?\b \

(?:(?:java|vb)script|shell|http):)|a(?:ctivexobject\b|lert\b\W*?\(|sfunc
tion:))| \

<(?:(?:body\b.*?\b(?:backgroun|onloa)d|input\b.*?\btype\b\W*?\bimage)\b|
\
    ?(?:(?:script|meta)\b|iframe)|!\[cdata\[)|(?:\.(?:(?:execscrip|addim
por)t| \
    (?:fromcharcod|cooki)e|innerhtml)|\@import)\b)" \

"capture,t:urlDecodeUni,t:htmlEntityDecode,t:compressWhiteSpace,t:lowerc
ase,log, \
```

```
    auditlog,deny,msg:'Cross-site Scripting (XSS) Attack - in request
headers or XML', \
    id:'959004',tag:'WEB_ATTACK/XSS',logdata:'%{TX.0}', \
    redirect:/_error_pages_/lesson08c.html,severity:'3'"
```

For stopping the malicious stored XSS script, this rule is used:

```
  SecRule TX:MENU "!@eq 900" "phase:4,t:none,pass,skip:1"

  # parse response body and write hidden values to file
  SecRuleScript "/etc/modsecurity/data/jscript_08.lua"
"phase:4,t:none,log, \
    auditlog,deny,msg:'Lesson 8 XSS Lab Stage 3, found nonconformant
javascript \
    tags using luascript',redirect:/_error_pages_/lesson08-3.html"
```

Since we are whitelisting the pattern that is accepting for using JavaScript, a Lua script is used. The source code is shown here in its entirety, replete with debug code for Level 9 debugging:

```
function main()
  m.log(9, "Starting luascript file jscript_08.lua")
  print ("Executing luascript jscript_08.lua")

  local tbuff = m.getvar("RESPONSE_BODY", "none")
  local str1

  for a in string.gmatch(tbuff, "<script.->") do
    str1 = string.format("\nLuascript: Trying to match: %s: ", a)
    m.log(9, str1)

    if string.match(a, \
      "^<script%s+language=\"JavaScript1.2\"%s+src=\"javascript/") ==
nil then
      str1 = string.format("Luascript: from RESPONSE_BODY - \
        string not matching: %s; exiting", a)
      m.log(9, str1)
      return str1
    end
  end

  m.log(9, "Exiting luascript file jscript_08.lua")
  return nil
end
```

Note that Lua is implemented in ModSecurity so that returning a value of 'nil' means there is no match in the SecRuleScript; you can think of 'nil' as a "good" return code meaning that nothing bad has happened. Anything else returned will cause a match and invoke the actions in the SecRuleScript.

Reviewer comments

- 8.1 Phishing with XSS
  - Can be mitigated by a combination of positive/negative security rules and the missing output encoding rule.
- 8.2 LAB: Cross Site Scripting
  - Stage 1: Stored XSS
  - Stage 3: Stored XSS Revisited
  - Stage 5: Reflected XSS
  - Can be mitigated by a combination of positive/negative security rules and the missing output encoding rule.
- 8.4 Reflected XSS Attacks
  - Can be mitigated by a combination of positive/negative security rules and the missing output encoding rule.
- 8.5 Cross Site Request Forgery (CSRF)
  - The scenario for this lesson is that the user forum where the attacker is injecting the CSRF code is on the same site/domain as the CSRF targeted webapp. So this means that initially blocking the CSRF injection would be feasible with the XSS rules and/or positive security for the newsgroup form submission page.
  - What is more challenging would be to assume that this newsgroup could possibly be hosted on a totally different website (possibly even hacker controlled). The attacker is hoping that the victim would happen to be currently logged into the target website at the same time they viewed the CSRF code page. Obviously, the likelihood of this increases significantly if the CSRF vector is hosted on the same site as the target app.
  - Assuming that the CSRF code is hosted on a separate site, the challenge from the web application's (and ModSecurity's) point of view is that we may *only* see the final request:

```
GET /WebGoat/attack?Screen=9&menu=900&transferFunds=5000
HTTP/1.1

Host: www.webgoat.net
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
rv:1.9.0.3) Gecko/2008092417 Firefox/3.0.3
Accept: image/png,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-us
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer:
http://www.webgoat.net/WebGoat/attack?Screen=145&menu=900&Num=8
Cookie: JSESSIONID=E98191536998CC789A59CD32EF865FB3
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
```

Without the implementation of Anti-CSRF tokens, it is very challenging to identify this request as malicious. One anomaly-based approach that we can see however is that when CSRF code is used within html "image" type tags, some browsers will tell you that it is expecting an image based on its Accept header. We could possibly write a ModSec rule to compare the URL file extension and Accept header to look for this type of mismatch.

- 8.7 Cross Site Tracing (XST) Attacks
    - The scenario for this lesson is that the form where the attacker is injecting the XST code is on the same site/domain as the targeted webapp. So this means that initially blocking the XST injection would be feasible with the XSS rules and/or positive security for the newsgroup form submission page.
    - Additionally, you can use ModSecurity rules to deny TRACE request methods.

5.27 Sublesson 8.6: HTTPOnly Test

8. Cross-Site Scripting (XSS) -> 8.6 HTTPOnly Test

(This sublesson was not formally solved by the project)

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson is a demonstration of the HttpOnly flag which prevents client side script from reading a cookie value if it is set; "Read Cookie" and "Write Cookie" buttons can be selected to view that particular browser's support of the HttpOnly flag.

There really is no ModSecurity solution or any other way to illustrate similar functionality within ModSecurity, especially since this takes place in the browser and ModSecurity cannot modify source code in an HTTP response.

Reviewer comments

For this lesson, ModSecurity rules can be implemented to inspect Response Headers and look for Set-Cookie data. When we see it, we can create an Application Defect alert if it is missing the HttpOnly flag:

```
SecRule RESPONSE_HEADERS:/Set-Cookie2?/ "!(?:\;? ?httponly;?)" \
"phase:3,t:none,t:lowercase,pass,log,auditlog, \
msg:'AppDefect: Missing HttpOnly Cookie Flag.'"
```

5.28 Sublesson 9.1: Denial of Service from Multiple Logins

9. Denial of Service -> 9.1 Denial of Service from Multiple Logins

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson has a database connection pool of 2 and provides 3 user names and passwords: jsnow:passwd1, jdoe:passwd2, and jplane:passwd3. Multiple logins by the same user is also allowed, so there are different variations of having a 3rd user logging in and resulting in a Denial of Service.

The ModSecurity solution to this lesson is to keep a list of logged in users and do not allow a user to log in more than once. Also, a 3rd user is not allowed to log in, which of course is a Denial of Service in itself but at least it is restricted to that user and does not affect the first 2 logged in users (which could occur in a real-world situation).

There is one small problem with our solution at this point: When we intercept the request, we don't know if the login at the server is successful or not. Therefore, we have to assume that the login is successful upon the request, write to the *.data file as such if the conditions are not violated, then in the response check if the login was successful or not. If the login failed then we need to remove the entry; we know it is the last entry in the data file, so we don't have to go through the headache of retaining the user name from the request to the response phase within Lua and then afterwards have to do any matching on the user name in the data file.

Implementation

A sample of the POST parameters is:

```
Username=jsnow&Password=passwd1&SUBMIT=Login'
```

We will persist a logged-in user to a file 'lesson09-1.data'; the user name is the only bit of information that is needed. At the beginning of the lesson, it has to be blank (use 'touch' on Linux) with the correct file permissions. The data file after 2 distinct users have logged in would consist of:

```
Entry{
  username = "jsnow"
}

Entry{
  username = "jdoe"
}
```

The phase 2 request portion of the configuration file is:

```
  SecRule ARGS:menu "!@eq 1000" "phase:2,t:none,skip:3"
  SecRule &ARGS_POST:SUBMIT "@eq 0" "nolog,skip:2"
  SecRule &ARGS_POST:Username "@eq 0" "nolog,skip:1"

  # action is triggered if script returns non-nil value
  SecRuleScript "/etc/modsecurity/data/denial-of-service-request_09-
1.lua" \
"phase:2,t:none,log,auditlog,deny,severity:3, \
msg:'Luascript: Denial of Service -> 9.1 Denial of Service from Multiple
Logins: \
request is
pending',tag:'DENIAL_OF_SERVICE',redirect:/_error_pages_/lesson09-
1.html"
  SecAction "phase:2,allow:request,t:none,log,auditlog,msg:'Luascript:
Denial of \
Service -> 9.1 Denial of Service from Multiple Logins: request is NOT
pending'"
```

The relevant section of the Lua script 'denial-of-service-request_09-1.lua' is:

```
  datafile = "/etc/modsecurity/data/lesson09-1.data"
  local username = m.getvar("ARGS_POST.Username", "none")

    function Entry (b)
      local ename = b.username

      usercount = usercount + 1

      if usercount >= 2 or username == ename then
        adduser = "no"
        msg1 = string.format("User '%s' is already logged in
           or the maximum number of users has been reached", username)
        msg2 = msg0 .. msg1
        m.log(9, msg2)
        retval = msg2
      end

      outstr = string.format("Entry{\n  username = \"%s\"\n}\n\n",
ename)
      tbuff = tbuff .. outstr
    end

    dofile(datafile)

    if adduser == "yes" then
      outstr = string.format("Entry{\n  username = \"%s\"\n}\n\n",
username)
      tbuff = tbuff .. outstr
    end

   -- write back to data file and return 'retval'...
```

What the program does:

- 'dofile(datafile)' invokes the function 'Entry', which loops and processes each entry in the data file
- If the conditions will be violated - either login count is more than 2 or the same user will be logged in - then the return value will be set to non-nil which will match the ModSecurity SecRuleScript directive and cause the request to be blocked
- While each datafile entry is being processed, it is being written back to a buffer which will be used to replace the existing data file.
- After the data file has been processed, if there is a new user then add it to the list of current users

The Phase 4 response portion of the configuration file is:

```
  SecRule TX:MENU "!@eq 1000" "phase:4,t:none,pass,skip:1"

  SecRuleScript "/etc/modsecurity/data/denial-of-service_response_09-
1.lua" \
"phase:4,t:none,log,auditlog,allow,msg:'Luascript: Denial of Service ->
\
9.1 Denial of Service from Multiple Logins; checking for failed login'"
```

In the response body, indicators of a failed or successful login attempt are:

```
  "Login Failed"
       "Successfull login count: 0"
```

And:

```
  "Login Succeeded: Total login count: 1"
```

Refer to the Lua script 'denial-of-service-response_09-1.lua'. The steps are:

- Determine whether login succeeded or failed
- If it succeeded, do nothing and return nil. If it failed, then:
- Remove the last user from the data file by:
    - Read the entries into an array
    - Write back the entries to the data file except for the last entry

Comments

- This lesson shows the use of an array in Lua.
- It would be nice if this WebGoat lesson had a logout mechanism which would make it easier to experiment with the total login count and the ModSecurity solution.

5.29 Sublesson 10.1: Fail Open Authentication Scheme

10. Improper Error Handling -> 10.1 Fail Open Authentication Scheme

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This lesson demonstrates a fail open vulnerability in that when the 'Password' parameter in the POST body is removed entirely, the user is authenticated. To mitigate this, a straightforward pinpoint strategy was used.

Implementation

The lesson is mitigated by the ruleset 'rulefile_10_improper-error-handling.conf':

```
  SecRule ARGS:menu "!@eq 1100" "t:none,pass,skip:2"
  SecRule &ARGS:Password "@eq 0" "t:none,deny,severity:3, \
    msg:'Post parameter Password is mandatory, but it is not present in
request.', \
    tag:'AUTHENTICATION',redirect:/_error_pages_/lesson10-1.html"
  SecAction "allow:request,t:none, \
    msg:'Returning; nothing bad on this page (rulefile_10-1).'"
```

Note that it is perfectly acceptable - and probably preferable - to use '`&ARGS_POST:Password`' instead of '`&ARGS:Password`'.

5.30 Sublesson 11.1: Command Injection

5.31 Sublesson 11.2: Blind SQL Injection

5.32 Sublesson 11.3: Numeric SQL Injection

5.33 Sublesson 11.4: Log Spoofing

5.34 Sublesson 11.5: XPATH Injection

5.35 Sublesson 11.6: String SQL Injection

5.36 Sublesson 11.7: LAB: SQL Injection

5.37 Sublesson 11.8: Database Backdoors

11. Injection Flaws -> 11.1 Command Injection
11. Injection Flaws -> 11.2 Blind SQL Injection
11. Injection Flaws -> 11.3 Numeric SQL Injection
11. Injection Flaws -> 11.4 Log Spoofing

11. Injection Flaws -> 11.5 XPATH Injection
11. Injection Flaws -> 11.6 String SQL Injection
11. Injection Flaws -> 11.7 LAB: SQL Injection
11. Injection Flaws -> 11.8 Database Backdoors

The lessons are combined because the strategy and the ruleset to solve them are the same.

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

Some lessons were mitigated using straightforward (probably too straightforward) pinpoint whitelisting rules.

Some lessons were mitigated using rules from the core ruleset. Note that these blacklist rules apply to all of the sublessons within this lesson and that some false positives had to be weeded out. Also note the rare use of 'skipAfter'.

The best way to explain the mitigating solutions is to present the ruleset in its entirety and let the comments speak for themselves.

Implementation

The lessons are mitigated in the ruleset 'rulefile_11_injection-flaws.conf':

5.38 Sublesson 12.1: Insecure Login

12. Insecure Communication -> 12.1 Insecure Login

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

The WebGoat lesson shows a password being sent in clear text vs. encrypted over SSL and observed using Wireshark. It is actually more of a demo than a vulnerability lesson, but this is chosen as a project solution because it introduces some new ModSecurity functionality and capabilities.

The ModSecurity solution is: whatever is typed in for the password on the client side without SSL is MD5-hashed on the client and and then sent to the server; a Lua script on the back end hashes the user name and compares it to the hashed password. If the user name and typed-in password are the same, there is a match (the WebGoat lesson structure is being followed here and not a real world scenario).

One should be able to see how this solution can be modified for real-world uses.

What this lesson will demonstrate:

- The powerful concept of being able to program on both ends - client and server - from within ModSecurity
- Encrypting the HTTP-sent plain-text password with an MD5 hash by including a JavaScript MD5 hash library and some extra JavaScript code
- Using a Lua script on the back end to MD5-hash a message (in our case, it will be the user name)
- The core Lua script shared object library that ModSecurity uses can be rebuilt to include a 3rd party crypto library (MD5 and DES56), which also shows that Lua functionality in ModSecurity is fully extensible
- A fully compatible Lua standalone engine with the new functionality is also built for the server, meaning Lua scripts can be fully tested as standalone before integration into Modsecurity. Also, these test scripts can be used as unit tests; for example, input data - HTTP request parameters or an HTTP response body - can be saved into a file off-line and the scripts can be tested outside of ModSecurity.

The steps for this solution:

(1) Identify the problem: The first part of the WebGoat lesson shows a password being sent over HTTP in clear text without SSL.

(2) Theorize a solution: Encrypt the password on the client side using JavaScript content injection, then send it over the wire and compare it with a message string that has been MD5-hashed in a Lua script on the back end; the result is then returned to the ModSecurity rule.

(3) Implement the solution.

Implementation

Step 1: There is a minor issue with the way that the WebGoat lesson is configured; it uses the 'readonly' attribute in the username and password fields which restricts our flexibility to test the project solution (and also it will not occur in a real-world scenario). If the 'readonly' attribute cannot be worked around, the importance of the mitigating solution

diminishes. There are 2 ways to work around this: (1) modify the WebGoat source code; or (2) use a web proxy, intercept and modify the user name and password values for experimentation, and observe the results there. The latter work-around is the obvious choice.

Step 2: Ensure that the Lua extension that includes the crypto library is working on your ModSecurity/Apache platform, which includes building Lua from source code and running a test program using the standalone Lua engine. For more information, see "Rebuilding the Lua library and standalone engine" towards the end of this solution. Linux using gcc version 4 is supported; Kubuntu 7.10 is the target platform but other Linux distros should work with just a little tweaking. Other platforms such as Windows will require more effort such as knowledge of the compiler, linker, and build process is necessary in order to modify the make file.

Step 3: Ensure that MD5 hashing is working inside of ModSecurity. Modify the Lua script to hash a hard-coded string:

```
function main()
  m.log(9, "Starting luascript file md5_12.lua")
  print ("Executing luascript md5_12.lua")

  local username = "sniffy"
  binstr1 = md5.sum(username)

       -- this converts binary string to lowercase hex string of length
32
  str2 = string.gsub(binstr1, ".", function (c)
           return string.format("%02x", string.byte(c))
         end)

  str1 = string.format("\nLuascript: Hashed user name is %s;" str2)
  m.log(9, str1)

  m.log(9, "Ending luascript file md5_12.lua")
end
```

The ModSecurity rule is:

```
  SecRuleScript "/etc/modsecurity/data/md5_12.lua" \
"phase:2,t:none,log,auditlog,allow:request,msg:'Luascript in \
rulefile_12-1-initialize.conf: In request (md5_12.lua)'"
```

Turn DebugLevel to 9, submit the WebGoat lesson page, then search the ModSecurity debug level for the string 'luascript' and check that it worked. As with testing all Lua scripts in ModSecurity, make sure the last log message is there; if not then the script bombed somewhere - probably accessing a variable with a value of 'nil'.

Step 4: Implement the client-side MD5 hashing of the password.

A popular JavaScript MD5 hash routine from Webtoolkit is used (http://www.webtoolkit.info/javascript-md5.html); its description is: "This script is used to

process a variable length message into a fixed-length output of 128 bits using the MD5 algorithm. It is fully compatible with UTF-8 encoding. It is very useful when u want to transfer encrypted passwords over the internet."

Download and copy the MD5 library, webtoolkit.md5.js, to the 'javascript' directory where WebGoat stores its JavaScript files; e.g. '/var/lib/tomcat5.5/webapps/WebGoat/javascript'.

The ModSecurity rule that includes the library by prepending it to the response body is:

```
  SecRule RESPONSE_CONTENT_TYPE "^text/html" \
"phase:4,t:none,log,auditlog,pass,msg:'prepending javascript in \
rulefile12-1_insecure-login.conf', \
prepend:'<script language=\"JavaScript1.2\"
src=\"javascript/webtoolkit.md5.js\" \
type=\"text/javascript\"></script>'"
```

Note that double-quotes within the JavaScript code has to be escaped with a back slash.

In the same HTTP response, a chunk of JavaScript is appended to the response body; after the 'Submit' button is pushed but before the form is submitted, the chunk takes the password, MD5 hashes it, and sets it to the new hashed value.

Step 5: Implement the server-side MD5 hashing of the user name. Replace the hard-coded string the Lua script that was done in Step 2 to include the HTTP post parameters of the form:

```
  local submit = m.getvar("ARGS_POST.Submit", "none")
  local username = m.getvar("ARGS_POST.clear_user", "none")
  local hashpwd = m.getvar("ARGS_POST.clear_pass", "none")

  if submit == nil or username == nil or hashpwd == nil then
    str1 = string.format("\nLuascript: Returning; not a form submit")
    m.log(9, str1)
    return nil
  end
```

If user name is 'sniffy' and the password is 'sniffy', the post parameters will look like this:

```
clear_user=sniffy&clear_pass=31036dfdb9c254fe161e9a1e9c76cd74&Submit=Sub
mit
```

Next, compare the values and return 'nil' or non-nil accordingly:

```
  if str2 == hashpwd then
    str1 = string.format("\nLuascript: Hashed user name matches hashed
password")
    retval = str1
    m.log(9, str1)
  else
    str1 = string.format("\nLuascript: Hashed user name does not match
hashed password")
    retval = nil
```

```
    m.log(9, str1)
  end

  return str2
```

Step 6: Next, we need to pass the result of the matching in the request phase to the response because we need to notify the end user somehow whether the match failed or succeeded. This is achieved by setting a session variable in Phase 2 and reading it in Phase 4.

Put this at the beginning of Phase 2:

```
  SecRule REQUEST_COOKIES:JSESSIONID "!^$" \
"chain,log,auditlog,pass,msg:'rulefile_12-1: Setting session
collection'"
  SecAction setsid:%{REQUEST_COOKIES.JSESSIONID}
  SecAction "log,setvar:session.lesson12=0,msg:'setting
session.lesson12=0 \
initially after setsid from rulefile_12-1-initialize.conf'"
```

And modify the SecRuleScript directive:

```
  # the lua script takes the user name, md5 hashes it,
  #   and compares it with the md5hashed password.
  # If a match, then it returns a non-nil string that triggers the rule
  SecRuleScript "/etc/modsecurity/data/md5_12.lua" \
"phase:2,t:none,log,auditlog,allow:request,setvar:session.lesson12=1,\
msg:'Luascript in rulefile_12-1-initialize.conf: In request
(md5_12.lua); \
user name matches password'"
```

The response rule section of 'rulefile_12-1_insecure-login.conf' is modified to:

```
  # Here check if session variable is set;
  #   if so, then send alert that hashed user name & password are same
  SecAction "phase:4,log,initcol:session=%{SESSIONID},\
msg:'rulefile_12-1: getting session.lesson12 DEBUG4'"

  SecRule SESSION:LESSON12 "@eq 1" \
"phase:4,chain,pass,log,auditlog,msg:'rulefile_12-1:user name matches
password'"
  SecAction "phase:4,allow,log,auditlog,\
msg:'rulefile_12-1:user name does not match password',skip:1"
```

Step 7: Notify the end user somehow whether the match failed or succeeded. In a perfect world, we should be able to append JavaScript code like:

```
  SecRule SESSION:LESSON12 "@eq 1" "phase:4,allow,log,auditlog,\
msg:'rulefile_12-1:user name matches password',\
append:'<script type=\"text/javascript\" language=\"JavaScript\"> \
alert(\"User name matches password\");</script>'"
  SecAction "phase:4,allow,log,auditlog,msg:'rulefile_12-1:user name
does not \
```

```
match password',append:'<script type=\"text/javascript\"
language=\"JavaScript\"> \
alert(\"User name does not match password\");</script>'"
```

But this doesn't work because now we are appending code twice in the same pass, and evidently ModSecurity does not treat this as 2 append actions; it replaces the first append action - which calculates the MD5 hash of the password on the client - with the second append action, which clearly we don't want. To workaround this, the 2 append actions have to be combined into one. The result is not pretty so it is not shown in its gory detail and understanding it is left as an exercise for the reader.

That's it. The password can be encrypted before it goes over the wire and compared with another value on the back end - all within ModSecurity.

Rebuilding the Lua library and standalone engine

See 'Appendix C: Building the Lua library and standalone executable' for details.

Comments

- First, it must be noted that long after this solution was implemented, it was discovered that ModSecurity provides an MD5 transformation function. Obviously, that would have been the easiest solution for the back end by far but there's no time machine to go back in time.

- What this solution showed:
    - The potency of having access to a programming language on both ends of the pipe: JavaScript on the client side via content injection and Lua script on the server side.
    - Lua in ModSecurity is extensible and, at the same time, a fully compatible Lua standalone engine is built.
    - Lua scripts can almost be fully built before integration into ModSecurity; the modules at this stage could also be used as standalone test cases.
    - JavaScript 'append' and 'prepend' in ModSecurity is directly useful for security when the attacker is not the end user; in this case, it is a man-in-the-middle attack.

- Some limitations were observed with ModSecurity:
    - ModSecurity only takes a binary return code from a Lua script - either nil or non-nil. A nice feature would be to have a built-in variable that sets a return string (or empty string for nil)
    - JavaScript content cannot be appended to a response body twice in same pass of a phase; the 2nd append will replace the first one.
    - The current JavaScript 'prepend' & 'append' implementation is not placed in the proper position in the HTML file; 'prepend' should go just before the '<body>' or the '<head>' tag (preferably both choices) and not before 'DOCTYPE' and '<html>' tags as is done now; 'append' content should be placed after the '</body>' end tag and before the '</html>' end tag - now it is place after the '</html>' tag. Not only is the current implementation

contrary to the HTML 4.01 and HTML 5 Specification, well-formedness for HTML documents is being pushed as a best practice security measure, and the current 'prepend' and 'append' implementation will cause a document not to be well-formed.

- A nice enhancement to this solution would be the ability to use asymmetric key encryption such as PGP and encrypt the payload with a public key - sent to the client as a hidden field - and decrypt it on the back end with the private key. However, a PGP implementation (e.g. via GnuPGP) for Lua does not seem to be currently available.

5.39 Sublesson 13.1: Forced Browsing

13. Insecure Configuration -> 13.1 Forced Browsing

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

The easiest way to mitigate the lesson would be to block all accesses to '/WebGoat/conf' and its derivatives in the '`<LocationMatch>`' tag but that would be too easy.

Instead, we'll will make it a little more complicated and block access the the 'conf' directory only if the visitor is coming from the forced browser lesson page.

Implementation

The lesson is mitigated partially by the ruleset 'rulefile_13_insecure-configuration.conf':

```
<LocationMatch "^/WebGoat/attack$">

  SecRule ARGS:menu "!@eq 1400" "t:none,skip:2"

  # Set a session variable to mitigate if next request the user will try
to
  #   access the '/WebGoat/conf' file which is checked in
rulefile00_initialize.conf
  SecAction "pass,log,initcol:session=%{SESSIONID}, \
    setvar:session.lesson13=1, \
    msg:'Setting session.lesson13=1 from rulefile_13_insecure-
configuration.conf'"

  SecAction "allow:request,t:none, \
    msg:'Returning; nothing bad on this page (rulefile_13).'"
```

```
</LocationMatch>
```

We are setting a session toggle switch (first the collection has to be loaded with 'initcol') which says "this page is coming from Lesson 13".

The tricky part is in 'rulefile_00_initialize.conf':

```
<LocationMatch "^/.*$">
  # Check session variable to see if the the user is trying to access
the
  #  '/WebGoat/conf' file which is set in rulefile13_initialize.conf
  #  (Lesson 13 - Insecure Configuration -> Forced Browsing); if
'/WebGoat/conf'
  #  is being attempted to be accessed from other lessons, we don't want
to block
  #  it with ModSecurity.
  # In real life, this is a lot of overhead - loading the collection for
  #  every request - but it's done to illustrate the use of collections

  SecAction "log,initcol:session=%{SESSIONID},msg:'getting
session.lesson13 DEBUG4'"
  SecRule SESSION:LESSON13 "@eq 1" "chain,deny,log,auditlog, \
    msg:'Insecure  Configuration',tag:'WEB_ATTACK/INSECURE_CONFIGURATION',
\
    severity:'3',redirect:/_error_pages_/lesson13-1.html"
  SecRule REQUEST_FILENAME "^/WebGoat/conf$" "t:none"
</LocationMatch>
```

The comments are fairly self-explanatory. The LocationMatch section is processed only if the HTTP request does not have an URL of '/WebGoat/attack' (which is highly unusual in WebGoat because of the way it is set up) from its LocationMatch section located before this in the initialization file. First, we check that if the request came from Lesson 13, then if so, we check if the '/WebGoat/conf' directory was requested and deny the request if those conditions are met.

One more thing that we have to take care of: when the request is not coming from Lesson 13, we want to toggle off the session variable (otherwise, once it is set it will be on all of the time for the rest of the session). At the end of the initialization section for the LocationMatch section for '/WebGoat/attack', we insert the rule:

```
  # If we are not in the 'Insecure Configuration -> Forced Browsing'
lesson,
  #   then toggle the session variable
  SecRule ARGS:menu "!@eq 1400" "nolog,setvar:session.lesson13=0"
```

which takes care of this.

It's informative to examine the initialization file in its entirety: in this rule, it was not necessary to load the session collection with 'initcol', while in the previous one - in a different LocationMatch section but in the same phase (2) and the same file, it is necessary to load the collection with 'initcol'. Also, note that to access the session variable in the 'rulefile_13_insecure-configuration.conf' file, even though the HTTP request, the LocationMatch section, and the phase is the same, it is necessary to load the collection with 'initcol'. Perhaps because it is in a different file? The documentation is not clear on

this, so a rule of thumb is to always check the debug to make sure that the session variable is being set if there are problems with it. The error message "cannot set variable because the collection does not exist" means that 'initcol' must be used.

5.40 Sublesson 15.1: Exploit Hidden Fields

5.41 Sublesson 15.2: Exploit Unchecked Email

15. Parameter Tampering -> 15.1 Exploit Hidden Fields

15. Parameter Tampering -> 15.2 Exploit Unchecked Email

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

To mitigate Lesson 15.1, the attacker cannot be allowed to alter the hidden field '`Price`'.

Lesson 15.2 has 2 parts:

Stage 1 is mitigated by preventing an XSS attack, so one line is added to the rule file 'rulefile_08_xss.conf' to take care of this.

```
  SecRule ARGS:menu "!@eq 1600" "chain,t:none,..."
```

Stage 2 has the same issue of altering a hidden field, in this case the email address in:

```
<input name='to' type='HIDDEN' value='webgoat.admin@owasp.org'>
```

Mitigating the hidden field value issues uses the same strategy that is used to solve '4.4 Multi Level Login 1' and '4.5 Multi Level Login 2'. In both cases, a check is implemented to see whether the hidden fields have been altered.

Implementation

The lesson is mitigated by the ruleset 'rulefile_15_parameter-tampering.conf':

```
  SecRule ARGS:menu "!@eq 1600" "phase:2,t:none,skip:2"

  # action is triggered if script returns non-nil value
  SecRuleScript "/etc/modsecurity/data/read-hidden-values_15.lua" \
    "phase:2,t:none,log,auditlog,deny,severity:3, \
```

```
      msg:'Parameter Tampering; Hidden field',tag:'PARAMETER_TAMPERING',
\
      redirect:/_error_pages_/lesson15-1.html"
  SecAction "phase:2,allow:request,t:none,log,auditlog, \
    msg:'Luascript: hidden field not altered or does not exist'"

#############################################################

  SecRule TX:MENU "!@eq 1600" "phase:4,t:none,pass,skip:1"

  # parse response body and write hidden values to file
  SecRuleScript "/etc/modsecurity/data/write-hidden-values1.lua" \
    "phase:4,t:none,log,auditlog,allow, \
     msg:'Writing RESPONSE BODY & parsed input fields to file using luascript'"
```

The Lua script used here in Phase 4 that writes the HTML input values to file is the exact same file as the one used in Lessons 4.4 and 4.5. The output is:

```
Entry{
  name = "gId",
  type = "TEXT",
  value = "GMail id"
}

Entry{
  name = "gPass",
  type = "PASSWORD",
  value = "password"


Entry{
  name = "subject",
  type = "TEXT",
  value = "Comment for WebGoat"
}

Entry{
  name = "to",
  type = "HIDDEN",
  value = "webgoat.admin@owasp.org"
}

Entry{
  name = "SUBMIT",
  type = "SUBMIT",
  value = "Send!"
```

Note that the Lua script could be modified with one line of code to write only hidden input fields, but we do not in order to illustrate writing all input types to a persistent data store.

The 'read-hidden-values_15.lua' file only has to be modified ever so slightly.

First, modify the debug messages such as:

```
  m.log(9, "entering Luascript read-hidden-values_15.lua"
```

Keep in mind that it is a good practice to include a word like 'luascript' in every debug message in order to be able to more easily search a debug file when necessary.

Next, to get the values from the POST request body:

```
local dprice = m.getvar("ARGS_POST.Price", "none")
local to1 = m.getvar("ARGS_POST.to", "none")
```

Then, filter by the field names:

```
if dprice ~= nil and type:lower() == "hidden" and name == "Price"
then ...
```

And:

```
if to1 ~= nil and type:lower() == "hidden" and name == "to" then ...
```

5.42 Sublesson 15.3: Bypass Client Side JavaScript Validation

15. Parameter Tampering -> 15.3 Bypass Client Side JavaScript Validation

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson demonstrates bypassing client-side validation using a web browser plugin such as IEWatch for IE or Firebug for Firefox. There are 7 different validation types and any field that does not meet the requirements will return an error message.

Often in real word implementations, the RegExs in the HTML source code are taken and re-implemented within ModSecurity so that validation is properly enforced and cannot be bypassed.

The ModSecurity solution will be to copy the RegExs from the HTML source and use them in ModSecurity rules.

Implementation

A sample POST URI and parameters are (the 2nd entry should be on one line):

```
POST http://192.168.0.5/WebGoat/attack?Screen=31&menu=1600
```

```
field1=abc&field2=123&field3=abc+123+ABC&field4=seven&field5=90210&
field6=90210-1111&field7=301-604-4882
```

The 'rulefile_15-3_bypass-client-side-validation.conf' is:

```
  # exactly three lowercase characters
  SecRule &ARGS_POST:field1 "@eq 0" "nolog,skipAfter:1530"
  SecRule ARGS_POST:field1 "!^[a-z]{3}$" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'15. Parameter
Tampering -> \
15.3 Bypass Client Side JavaScript Validation: malicious attempt to
enter \
invalid data',tag:'INJECTION_ATTACK',redirect:/_error_pages_/lesson15-
3.html"

  # exactly three digits
  SecRule &ARGS_POST:field2 "@eq 0" "nolog,skipAfter:1530"
  SecRule ARGS_POST:field2 "!^[0-9]{3}$" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'15. Parameter
Tampering -> \
15.3 Bypass Client Side JavaScript Validation: malicious attempt to
enter \
invalid data',tag:'INJECTION_ATTACK',redirect:/_error_pages_/lesson15-
3.html"

  # letters, numbers, and space only
  SecRule &ARGS_POST:field3 "@eq 0" "nolog,skipAfter:1530"
  SecRule ARGS_POST:field3 "!^[a-zA-Z0-9 ]*$" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'15. Parameter
Tampering -> \
15.3 Bypass Client Side JavaScript Validation: malicious attempt to
enter \
invalid data',tag:'INJECTION_ATTACK',redirect:/_error_pages_/lesson15-
3.html"

  # enumeration of numbers
  SecRule &ARGS_POST:field4 "@eq 0" "nolog,skipAfter:1530"
  SecRule ARGS_POST:field4
"!^(one|two|three|four|five|six|seven|eight|nine)$" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'15. Parameter
Tampering -> \
15.3 Bypass Client Side JavaScript Validation: malicious attempt to
enter \
invalid data',tag:'INJECTION_ATTACK',redirect:/_error_pages_/lesson15-
3.html"

  # simple zip code
  SecRule &ARGS_POST:field5 "@eq 0" "nolog,skipAfter:1530"
  SecRule ARGS_POST:field5 "!^\d{5}$" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'15. Parameter
Tampering -> \
15.3 Bypass Client Side JavaScript Validation: malicious attempt to
enter \
invalid data',tag:'INJECTION_ATTACK',redirect:/_error_pages_/lesson15-
3.html"
```

```
  # zip with optional dash four
  SecRule &ARGS_POST:field6 "@eq 0" "nolog,skipAfter:1530"
  SecRule ARGS_POST:field6 "!^\d{5}(-\d{4})?$" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'15. Parameter
Tampering -> \
15.3 Bypass Client Side JavaScript Validation: malicious attempt to
enter \
invalid data',tag:'INJECTION_ATTACK',redirect:/_error_pages_/lesson15-
3.html"

  # US phone number with or without dashes
  SecRule &ARGS_POST:field7 "@eq 0" "nolog,skipAfter:1530"
  SecRule ARGS_POST:field7 "!^[2-9]\d{2}-?\d{3}-?\d{4}$" \
"phase:2,t:none,log,auditlog,deny,severity:3,msg:'15. Parameter
Tampering -> \
15.3 Bypass Client Side JavaScript Validation: malicious attempt to
enter \
invalid data',tag:'INJECTION_ATTACK',redirect:/_error_pages_/lesson15-
3.html"
  SecAction "t:none,nolog,id:'1530'"
```

Comments

- This solution takes the RegExs used for client-side validation and incorporates them in ModSecurity rules.

5.43 Sublesson 16.1: Hijack a Session

16. Session Management Flaws -> 16.1 Hijack a Session

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson teaches how to hijack another user's session using brute force attacks. The vulnerability is that the session ID is not complex and random enough; the exploit is complex and is done by using WebScarab and another tool called Crowbar.

Session Management Flaws are one of the few areas where a WAF will not be of much help.

The only action that can be taken is a reactive one instead of a preventative action: issue an alarm in the event of irregularities (e.g. a changing IP) or terminate a session with changing IP. Besides that, a ModSecurity solution is not possible for this lesson.

Implementation

None

Reviewer comments

You had listed this as a Lesson where Mod (WAF) can't really help however two practical solutions are possible:

1. Anti-automation rules: In order to gather a large enough collection of SessionIDs to analyze their entropy, most attackers will use automation such as with the Session Collector plugin of WebScarab. When these tools run, they will send a large number of requests that will go over a ModSecurity set threshold of request/time and then you can block them. This is not a direct mitigation for the weak SessionIDs but may tactically help to identify people conducting this type of analysis.

2. The critical point in identifying Session Hijacking attacks is to bind the client IP (and possibly other browser data) to the SessionID when the webapp hands it out in the Set-Cookie response header. If this is done, you can then identify when someone guesses a SessionID and adds it to their browser as the REMOTE_ADDR variable data will not match what ModSecurity saved in the setsid Session collection when the app handed it out.

5.44 Sublesson 16.2: Spoof an Authentication Cookie

16. Session Management Flaws -> 16.2 Spoof an Authentication Cookie

(This sublesson was not formally solved by the project)

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson shows how to guess another user's authentication cookie. The vulnerability is that the cookie is easy to guess even from only 2 unique user accounts and their authentication cookie; it is exploited by logging in as another user and supplying the predictable authentication cookie.

A ModSecurity solution is not possible for this lesson because it is not possible to distinguish between an authentication cookie that comes from a valid user versus one that comes from an attacker.

Implementation

None

Reviewer comments

For this particular lesson, it is possible to use mitigation #2 of Sublesson 16.1:

2. The critical point in identifying Session Hijacking attacks is to bind the client IP (and possibly other browser data) to the SessionID when the webapp hands it out in the Set-Cookie response header. If this is done, you can then identify when someone guesses a SessionID and adds it to their browser as the REMOTE_ADDR variable data will not match what ModSecurity saved in the setsid Session collection when the app handed it out.

We know that the cookie is weak/easily guessed however what we will be enforcing is that the webapp did *NOT* give this out to the client in a Set-Cookie response header so it is not valid.

5.45 Sublesson 16.3: Session Fixation

16. Session Management Flaws -> 16.3 Session Fixation

(This sublesson was not formally solved by the project)

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat session demonstrates a session fixation attack. An attacker chooses a Session ID, sends it in a link to the victim, the victim clicks the link and logs in, then the attacker uses that session ID to log in to the same banking web site and thereafter masquerades as the victim.

A ModSecurity solution for this specific lesson would be to attach the user name to each unique session ID and persist it using Lua. Afterwards, if another user tried to log in using the same session ID, then the attacker would not be allowed to log in to the site.

Implementation

None

Reviewer comments

I believe that there are 2 potential Application Defects which make web application vulnerable to Session Fixation attacks that we can identify with ModSecurity (blocking is optional):

1. Web Applications that allow SessionIDs to be passed within a QUERY_STRING

As the lesson scenario shows, Phishing attacks that include links with previously known, pre-authentication SessionIDs are common. You can create a ModSecurity rule that will generate an alert when it sees a SessionID being passed in the QUERY_STRING. Flag is as a possible Session Fixation Attack.

2. Web Applications that do not set a new SessionID upon authentication

This is where you would need to identify the actual login page and then you can raise a flag if a successful auth does *NOT* result in the web application sending out a new Set-Cookie SessionID value. We apps that take an existing, pre-auth SessionID and then simply dubs it as now being authenticated leaves open a possibility of Session Fixation attacks.

One additional note in relation to the 2 previous Session Management Flaws rules – Sublessons 16.1 and 16.2 - a similar check for SessionIDs + IPs might work here however in reverse. The attacker would connect to the site and receive a pre-auth SessionID. We would then tie that SessionID to the attacker's data. When the attacker sends the Phishing link to the victim and the victim uses it, our rules could possibly flag this as a Session Hijacking attack. We would have to have rules that will be able to identify/handle inbound SessionIDs in either/or the SessionID Request Cookie data or a parameter with these names.

5.46 Sublesson 17.1: Create a SOAP Request

17. Web Services -> 17.1 Create a SOAP Request

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson demonstrates the use of WSDL files and examines a SOAP request.

The ModSecurity solution will be to discuss XML Web services and describe the configuration changes that would be required for ModSecurity to inspect this data.

Implementation

SOAP is a protocol for exchanging XML-based messages over computer networks, normally using HTTP/HTTPS. It forms the foundation layer of the web services protocol stack providing a basic messaging framework upon which abstract layers can be built. The most common use of SOAP is the Remote Procedure Call, in which one network node (the client) sends a request message to another node (the server) and the server immediately sends a response message to the client.

The purpose of WSDL (Web Services Markup Language) is to describe the interfaces of Web services; SOAP is used to communicate with the Web services.

The project solution for this WebGoat lesson is to consolidate from the reference manual how ModSecurity supports XML and Web services.

In order for ModSecurity to process XML in the HTTP response body, add 'text/xml' to this directive:

```
SecResponseBodyMimeType text/plain text/html text/xml
```

If it is not present, the response body buffer will be empty and the normal debug log messages that ModSecurity is loading chunks of the response body will be missing. Once XML is enabled, XPath expressions can be used against the special variable 'XML'; the following attempts to match 'dirty':

```
SecRule XML:/xPath/Expression dirty
```

To process XML in the request body, it necessary to turn on XML in the request header phase using the configuration option 'ctl:requestBodyProcessor=XML'; after the request body is processed as XML, XML-related features can be used to inspect it as in the following example.

The variable 'XML' can also be used as a standalone variable as a target for the validateDTD and validateSchema operators:

```
SecRule REQUEST_HEADERS:Content-Type ^text/xml$ \
  phase:1,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML
SecRule REQBODY_PROCESSOR "!^XML$" nolog,pass,skip:1
```

```
  SecRule XML "@validateDTD /path/to/apache2/conf/xml.dtd"
```

Or, the last line could be:

```
  SecRule XML "@validateSchema /path/to/apache2/conf/xml.xsd"
```

One action, 'xmlns', is used together with an XPath expression to register a namespace:

```
  SecRule REQUEST_HEADERS:Content-Type "text/xml" \
    phase:1,pass,ctl:requestBodyProcessor=XML,ctl:requestBodyAccess=On,\
xmlns:xsd="http://www.SecRule
XML:/soap:Envelope/soap:Body/q1:getInput/id() 123",\
phase:2,deny
```

Comments

- This ModSecurity solution discusses XML Web services and describes the configuration changes that would be required for ModSecurity to inspect XML data.

5.47 Sublesson 17.2: WSDL Scanning

17. Web Services -> 17.2 WSDL Scanning

(This sublesson was not formally solved by the project)

Lesson overview

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solution

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

This WebGoat lesson demonstrates WSDL Scanning to get credit card information. The vulnerability is that the attacker can call a Web Service directly; in this case, it is 'getCreditCard'. The attacker retrieves the WSDL, observes the parameters of a normal request '`id=101&SUBMIT=Submit`', then intercepts and manipulates the parameters to call the function directly: '`id=101&field=getCreditCard&SUBMIT=Submit`'

A ModSecurity solution is straightforward: don't allow a user to directly call a specific set of Web Services - in our case 'getCreditCard' - by blocking on the request if that value for the 'field' parameter appears.

Implementation

None

5.48 Sublesson 17.3: Web Service SAX Injection

5.49 Sublesson 17.4: Web Service SQL Injection

17. Web Services -> 17.3 Web Service SAX Injection

17. Web Services -> 17.4 Web Service SQL Injection

Lesson overviews

The WebGoat lesson overview is included with the WebGoat lesson solution.

Lesson solutions

Refer to the zip file with the WebGoat lesson solutions. See Appendix A for more information.

Strategy

Mitigating these 2 lessons is done with a whitelist, using pinpoint strategy.

Implementation

The lesson is mitigated by the ruleset and the solution is self-explanatory:

```
<LocationMatch "^/WebGoat/attack$">

  # 'menu=1800'
  SecRule ARGS:menu "!@eq 1800" "t:none,pass,skipAfter:170"

  # The following rule is for lesson '17.4 WS SQL Injection'.
  # The regex is a whitelist only for this particular lesson and for
  #   the 'account number' parameter. The criteria is that only digits and no
  #   spaces or special chars are allowed. Modify the regex if something like
  #   '-' is allowable.
  SecRule &ARGS_POST:SUBMIT "@eq 0" "nolog,skip:4"
  SecRule ARGS_POST:SUBMIT "!@streq Go!" "nolog,skip:3"
  SecRule &ARGS_POST:account_number "@eq 0" "nolog,skip:2"
  SecRule ARGS_POST:account_number "^([0-9]+)$" \
    "t:urlDecodeUni,t:htmlEntityDecode,allow:request,skip:1"
  SecAction "deny,log,auditlog,msg:'WS SQL Injection', \
    tag:'Returning from 17.4 WS SQL Injection (rulefile_17).', \
    severity:'3',redirect:/_error_pages_/lesson17-4.html"

  # The following rule is for lesson '17.3 SAX Injection'.
  # The regex is a whitelist only for this particular lesson and
  #   for the 'password' parameter. The criteria is that no spaces are allowed.
  SecRule &ARGS_POST:SUBMIT "@eq 0" "nolog,skip:4"
  SecRule ARGS_POST:SUBMIT "!@streq Go!" "nolog,skip:3"
```

```
  SecRule &ARGS_POST:password "@eq 0" "nolog,skip:2"
  SecRule ARGS_POST:password "[ ]" \
    "t:urlDecodeUni,t:htmlEntityDecode,deny,log,auditlog, \
    msg:'WS SAX Injection',tag:'WEB_ATTACK/SAX_INJECTION', \
    severity:'3',redirect:/_error_pages_/lesson17-3.html"
  SecAction "allow:request,t:none,id:'170', \
    msg:'Returning from 17.3 SAX Injection (rulefile_17).'"

  SecAction "allow:request,t:none,msg:'Returning; nothing bad on this page
(rulefile_17).'"
</LocationMatch>
```

Note that 'skipAfter' is used so the order of the rule groupings cannot be swapped.

For some reason, the project member had difficulty using 'chain' to chain rules together - in this lesson it simply was not working so the kludgey 'skip' actions were used instead; not elegant, but it works.

6. Appendix A: WebGoat lesson plans and solutions

Phase 1 (first 50% of project)

The zip file contains the WebGoat lesson plans and solutions. The current version needs some work (an index.html file, fix broken links, etc.) and a new version will be available on 28 July 2008 (note: the new version is available as of 27 July 2008).

Please see readme.txt for instructions. The specific lesson solutions in this zip file are the ones not in the Phase 2 zip file listed below.

Image:OWASP Securing WebGoat using ModSecurity WebGoat Lessons.zip

Phase 2 (second 50% of project)

The zip files contain the WebGoat lesson solutions for the project lessons for Phase 2 that can be viewed off-line (meaning, not as a part of WebGoat plus with no broken links to the images). The files total around 12 meg but are broken into smaller chunks (unzip in the same directory). They allow someone to understand the WebGoat lessons fairly well without having to install and use WebGoat. Many images embedded in the pages are low-resolution *.png files; in the lesson's respective subdirectories, there are higher resolution *.jpg files which are helpful, for example, to get the exact text being used in WebScarab.

Image:ModSec on WebGoat solutions1 Phase2.zip

Image:ModSec on WebGoat solutions2 Phase2.zip

Image:ModSec on WebGoat solutions3 Phase2.zip

Image:ModSec on WebGoat solutions4 Phase2.zip

The lessons contained in the Phase 2 zip files are:

1.1 Http Basics

2.2 Bypass a Path Based Access Control Scheme

2.3 LAB: Role Based Access Control

3.1 LAB: DOM-Based cross-site scripting

3.2 LAB: Client Side Filtering

3.4 DOM Injection

3.5 XML Injection

3.6 JSON Injection

3.7 Silent Transactions Attacks

3.8 Dangerous Use of Eval

3.9 Insecure Client Storage

7.1 Thread Safety Problem

7.2 Shopping Cart Concurrency Flaw

8.3 Stored XSS Attacks

8.6 HTTPOnly Test

9.1 Denial of Service from Multiple Logins

12.1 Insecure Login

14.1 Encoding Basics

15.3 Bypass Client Side JavaScript Validation

16.1 Hijack a Session

16.2 Spoof an Authentication Cookie

16.3 Session Fixation

17.1 Create a SOAP Request

17.2 WSDL Scanning

All other lesson solutions are in the Phase 1 zip file.

7. Appendix B: Project solution files

The following zip file contains the project solution files for Phase 1 (the first 50%). Please see readme.txt for instructions.

Image:OWASP Securing WebGoat using ModSecurity Project Solutions.zip

The following zip file contains the project solution files for Phase 2 (the second 50% of the project). Please see readme.txt for instructions.

Image:ModSec on WebGoat solutions Phase2.zip

8. Appendix C: Building the Lua library and standalone executable

This section details how to add MD5 and DES 56 encryption to Lua and how to build liblua5.1.so and the standalone executable, lua5.1, from sources on Linux. The target system used gcc version 4 on Kubuntu 7.10 (a 2.6 kernel), but should work on almost all versions of Linux.

Strategy

Implementing MD5 hashing in Lua was not so easy. It is not part of the core library so a 3rd party module was used:

MD5 - Cryptographic Library for Lua: http://www.keplerproject.org/md5/manual.html

To backtrack a little, ModSecurity loads the Lua shared object file when Apache starts:

```
LoadFile /usr/lib/libxml2.so
LoadFile /usr/lib/liblua5.1.so
LoadModule security2_module /usr/lib/apache2/modules/mod_security2.so
```

The 3rd party MD5 library (which also includes DES56) is offered as a shared object library, and this caused problems as it was not possible to add it as a separate add-in to the existing Lua library as in:

```
LoadFile /usr/lib/libxml2.so
LoadFile /usr/lib/liblua5.1.so
LoadFile /usr/lib/libluacrypt.so
LoadModule security2_module /usr/lib/apache2/modules/mod_security2.so
```

So the decision was made to try to integrate the crypto library source code into the Lua core with the result of only one Lua *.so library. This presented its own challenges because: (1) The makefile for a source code build did not implement building a *.so library (only an archive *.a library); and (2) The 3rd party crypto library was presented as a *.so library and not as a part of one executable or shared object library.

Another obstacle was that the Lua source code makefile has no provision to be built as a standalone executable.

However, these obstacles were overcome. The MD5/DES56 crypto library was successfully added to the Lua core as one *.so library plus it was integrated into the Lua stand-alone

engine. Building an executable vs. a shared object library is accomplished by changing a few lines of the makefile.

Implementation

To download the Lua source code, click on the 'source code' link at 'http://www.lua.org/download.html'; the file is:

```
156540 2008-07-30 19:03 lua5_1_3_Sources.tar.gz
```

Note: The source code files are in C.

Note: Lua 5.1.4 was released on 22 Aug 2008; it is not known if the makefiles are exactly the same from version 5.1.3 and can be simply interchanged to build MD5 into version 5.1.4.

Get the MD5 information and sources from here:

```
MD5 - Cryptographic Library for Lua
http://www.keplerproject.org/md5/manual.html (see the end of this
Appendix to view the Kepler Project license)

Luaforge: MD5: Filelist
http://luaforge.net/frs/?group_id=155

 29768 2008-07-30 15:04 md5-1.1.2.tar.gz
```

It's a long story to describe how the Lua makefile and source code files were edited, and how the MD5/DES56 files were added, so here is the entire subdirectory of everything:

Image:OWASP ModSecurity Securing WebGoat lua513 Jul31.zip

How it works:

- There are 2 Makefiles off of 'lua513/src' directory:
    - Copy the Makefile from 'org' into 'src', 'make clean', 'make linux', to build the standalone lua engine. Then move that to '../bin' directory and run any tests from there
    - Copy from 'save' to 'src' subdirectory to build the 'liblua5.1.so.3.0.0' file

For the curious, the modified sections of the source code are designated by comments with 'SCE', the project member's initials, so search on that. The key to getting it to work was to declare the accessible functions as 'LUALIB_API', which is 'extern'. Then load the md5 and des56 libraries in 'linit.c'; declare strings and define lib functions in 'lualib.c'; change the declarations in md5.c & ldes56.c from 'static' to 'LUALIB_API' and add the declarations in md5.h & ldes56.h.

Build both the executable and the *.so library.

Next, create a Lua source file, 'md5_standalone-test.lua' to test standalone without ModSecurity.

```
function main()
```

```
  print ("Executing luascript md5_12.lua")

  local username = "sniffy"
  binstr1 = md5.sum(username)

  -- this converts binary string to lowercase hex string of length 32
  str2 = string.gsub(binstr1, ".", function (c)
           return string.format("%02x", string.byte(c))
         end)

  str1 = string.format("\nLuascript: Hashed user name for user '%s'
is %s;", username, str2)
  print (str1)

  return
end

main()
```

Run the program from the command line:

```
root@www:/opt/lua513/bin# ./lua5.1 md5_standalone-test.lua
Executing luascript md5_12.lua

Luascript: Hashed user name for user 'sniffy' is
31036dfdb9c254fe161e9a1e9c76cd74;
```

Next, ensure that MD5 hashing is working inside of ModSecurity.

Copy the *.so file that was built, 'liblua5.1.so.3.0.0', into a standard library directory. Do 'chmod 644 liblua5.1.so.3.0.0'. Then create links to it so that a 'ls -alt' looks like this:

```
root@www:/usr/lib# ls -alt liblua*
-rw-r--r-- 1 root root 497278 2008-07-31 15:33 liblua5.1.so.3.0.0
lrwxrwxrwx 1 root root     18 2008-07-31 11:33 liblua5.1.so.0 ->
liblua5.1.so.3.0.0
lrwxrwxrwx 1 root root     18 2008-07-31 11:23 liblua5.1.so ->
liblua5.1.so.3.0.0
```

Use this Lua script to hash a hard-coded string:

```
function main()
  m.log(9, "Starting luascript file md5_12.lua")
  print ("Executing luascript md5_12.lua")

  local username = "sniffy"
  binstr1 = md5.sum(username)

  -- this converts binary string to lowercase hex string of length 32
  str2 = string.gsub(binstr1, ".", function (c)
           return string.format("%02x", string.byte(c))
         end)

  str1 = string.format("\nLuascript: Hashed user name for 'sniffy'
is %s;" str2)
```

```
  m.log(9, str1)

  m.log(9, "Ending luascript file md5_12.lua")
end
```

The ModSecurity rule is:

```
  SecRuleScript "/etc/modsecurity/data/md5_12.lua" \
"phase:2,t:none,log,auditlog,allow:request,msg:'Luascript in \
rulefile_12-1-initialize.conf: In request (md5_12.lua)'"
```

Turn DebugLevel to 9, submit the WebGoat lesson page, then search the ModSecurity debug level for the string 'luascript' and check that it worked. As with testing all Lua scripts in ModSecurity, make sure the last log message is there; if not then the script bombed somewhere - probably accessing a variable with a value of 'nil'.

Comments

Some of the benefits of this approach:

- Almost all of the functionality can be implemented in a Lua module run by the stand-alone engine before integrating it with ModSecurity. This approach makes debugging much, much easier.
- Combining all Lua functionality into one *.so file means that the ModSecurity configuration does not have to be touched; only replacing the *.so file is necessary.
- Lua is extensible and any 3rd party library with source code can be included. Note: The source code files are in C.

Kepler project license (for Lua MD5/DES56 library)

From http://www.keplerproject.org/en/License:

> The spirit of the license is that you are free to use Kepler for any purpose at no cost without having to ask us. The only requirement is that if you do use Kepler, then you should give us credit by including the appropriate copyright notice somewhere in your product or its documentation.
>
> Kepler is designed and implemented by the Kepler team. The implementation is not derived from licensed software.
>
> Copyright © 2004 Kepler Project.
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

THE BELOW ICONS REPRESENT WHAT OTHER VERSIONS ARE AVAILABLE IN PRINT FOR THIS TITLE BOOK.

ALPHA: “Alpha Quality” book content is a working draft. Content is very rough and in development until the next level of publication.

BETA: “Beta Quality” book content is the next highest level. Content is still in development until the next publishing.

RELEASE: “Release Quality” book content is the highest level of quality in a books title’s lifecycle, and is a final product.

ALPHA
PUBLISHED

BETA
PUBLISHED

RELEASE
PUBLISHED

YOU ARE FREE:

to share - to copy, distribute and transmit the work

to Remix - to adapt the work

UNDER THE FOLLOWING CONDITIONS:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. - If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

OWASP
The Open Web Application Security Project

The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software. Our mission is to make application security "visible," so that people and organizations can make informed decisions about application security risks. Everyone is free to participate in OWASP and all of our materials are available under a free and open software license. The OWASP Foundation is a 501c3 not-for-profit charitable organization that ensures the ongoing availability and support for our work.