UNIVERSIDADE FEDERAL DE SANTA MARIA
COLÉGIO TÉCNICO INDUSTRIAL DE SANTA MARIA
CURSO SUPERIOR DE TECNOLOGIA EM REDES DE COMPUTADORES

METODOLOGIA DE CONFIGURAÇÃO DE VULNERABILIDADES PARA O MODSECURITY

TRABALHO DE CONCLUSÃO DE CURSO

Jonas Eduardo Stein

Santa Maria, RS, Brasil

2013

CTISM/UFSM, RS   STEIN, Jonas Eduardo   Tecnólogo   2013

METODOLOGIA DE CONFIGURAÇÃO DE VULNERABILIDADES PARA O MODSECURITY

Jonas Eduardo Stein

Trabalho de conclusão de curso apresentado ao Curso Superior de Tecnologia em Redes de Computadores da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Tecnólogo em Redes de Computadores**.

Orientador: Prof. Me. Rogério Corrêa Turchetti
Coorientador: Prof. Me. Tiago Antônio Rizzetti

Santa Maria, RS, Brasil

2013

Universidade Federal de Santa Maria
Colégio Técnico Industrial de Santa Maria
Curso Superior de Tecnologia em Redes de Computadores

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Conclusão de Curso

METODOLOGIA DE CONFIGURAÇÃO DE VULNERABILIDADES PARA O MODSECURITY

elaborado por
Jonas Eduardo Stein

como requisito parcial para a obtenção do grau de
Tecnólogo em Redes de Computadores

COMISSÃO EXAMINADORA

____
Rogério Corrêa Turchetti, Me.
(Presidente/Orientador)

____
Tiago Antônio Rizzetti, Me.
(Coorientador)

____
Celio Trois, Me. (UFSM)

____
Eugênio de Oliveira Simonetto, Dr. (UFSM)

Santa Maria, 25 de janeiro de 2013.

RESUMO

Trabalho de Conclusão de Curso
Curso Superior de Tecnologia em Redes de Computadores
Universidade Federal de Santa Maria

METODOLOGIA DE CONFIGURAÇÃO DE VULNERABILIDADES PARA O MODSECURITY

Autor: Jonas Eduardo Stein
Orientador: Rogério Corrêa Turchetti
Coorientador: Tiago Antônio Rizzetti
Data e Local da Defesa: Santa Maria, 25 de janeiro de 2013.

Este trabalho se fundamenta nas soluções do *firewall* para aplicações *web* intitulado *modSecurity* e do projeto da OWASP denominado *ModSecurity Core Rule Set*. Visto que o *modSecurity* trabalha com regras, foi criado o referido projeto, a fim de ajudar seus utilizadores na melhoria para detecção das vulnerabilidades e, dessa forma, tirar o máximo de proveito do *firewall* em questão. Sendo assim, este trabalho analisa as soluções em conjunto e propõem uma configuração de forma que direciona cada tipo de vulnerabilidade para cada página da aplicação *web*, com intuito de melhorar o desempenho do servidor *web Apache*. Para comprovar que a metodologia de configuração proposta realmente funciona, são realizados testes de performance para provar o aumento de desempenho do *Apache*. Também são efetuados testes funcionais para provar que ocorre a detecção das vulnerabilidades. Há, ainda, testes manuais, para testar o projeto da OWASP, objetivando encontrar falhas nas detecções das vulnerabilidades e propondo a solução das mesmas.

Palavras-chave: Segurança na web; WAF; testes de performance; testes funcionais; *ModSecurity*.

ABSTRACT

Course Conclusion Paper
Computer Networks Technology Degree Course
Federal University of Santa Maria

VULNERABILITY CONFIGURATION METHODOLOGY FOR MODSECURITY
Author: Jonas Eduardo Stein
Adviser: Rogério Corrêa Turchetti
Coadviser: Tiago Antônio Rizzetti
Defense Place and Date: Santa Maria, January 25, 2013.

This work is based on the solutions of web application firewall called ModSecurity and the OWASP project called ModSecurity Core Rule Set. Since ModSecurity works with rules, the referred project has been created, in order to help its users to detect vulnerabilities, and thus taking full advantage of the firewall in question. Therefore, this paper analyzes the solutions together and proposes a configuration in a way that directs each type of vulnerability for each page of the web application, aiming to improve the performance of the Apache web server. Order to prove that the proposed configuration methodology really works, performance tests are conducted to prove the increase of Apache performance. Functional tests are also performed to prove that occurs detection of vulnerabilities. There are also manual testing to test the OWASP project, aiming to find flaws in vulnerabilities detection and proposing the solution of them.

Keywords: Web security; WAF; performance tests; functional tests; *modSecurity*

LISTA DE ANEXOS

Anexo A - Configuração personalizada no arquivo *httpd.conf* do *Apache*...............61
Anexo B - Configuração do arquivo de configuração do CRS..................................66
Anexo C - Diretrizes do *modSecurity*...69

LISTA DE QUADROS

Quadro 1 - Como proteger a aplicação *web*...37
Quadro 2 - Casos de teste e ambientes testados 41
Quadro 3 - Resultados dos casos de teste 47

LISTA DE TABELAS

Tabela 1 - Resultados com o *modSecurity* habilitado 51
Tabela 2 - Resultados com o *modSecurity* desabilitado...51
Tabela 3 - Comandos utilizados com o SQLMap 52
Tabela 4 - Resultados com o *modSecurity* personalizado 52

LISTA DE FIGURAS

Figura 1 - Modo incorporado....17
Figura 2 - Modo baseado na rede....18
Figura 3 - Funcionamento de uma requisição no *modSecurity*....18
Figura 4 - Exemplo de requisição....19
Figura 5 - Parte de *log debug* de uma requisição maliciosa....21
Figura 6 - Exemplo de LDAP injection....27
Figura 7 - Exemplo de SSI *injection*....28
Figura 8 - Exemplo de ataque UPDF XSS....29
Figura 9 – Exemplo de email *injection*....30
Figura 10 - Exemplo de HTTP *request smuggling*....31
Figura 11 - Exemplo de ataque *remote file inclusion*....31
Figura 12 - Exemplo de *session fixation*....32
Figura 13 - Exemplo de *command injection*....33
Figura 14 – Exemplo de sql *injection*....33
Figura 15 - Exemplo de XSS....34
Figura 16 - Exemplo de *directory traversal*....35
Figura 17 - Descoberta de métodos que uma aplicação *web* suporta....38
Figura 18 - Configuração padrão do *modSecurity*....39
Figura 19 - Arquitetura dos testes....43
Figura 20 - Vazão do *Apache* em respostas por segundo....47
Figura 21 - Vazão do *Apache* em *kilobytes* por segundo....48
Figura 22 - Tempo médio de resposta do *Apache*....49
Figura 23 - Consumo de tempo da cpu pelo *Apache*....49
Figura 24 - Consumo de memória pelo apache....50
Figura 25 - Herança de regras....83
Figura 26 - Exemplo da diretriz SecWebAppId....90

SUMÁRIO

1 INTRODUÇÃO ... 12
1.1 Objetivos ... 13
1.2 Organização ... 13
2 REVISÃO BIBLIOGRÁFICA ... 14
2.1 Modsecurity. ... 14
2.1.1 Características ... 14
2.1.1.1 *Log* de tráfego HTTP ... 15
2.1.1.2 Monitoração e detecção de ataque em tempo real ... 15
2.1.1.3 Prevenção de ataques e *virtual patching* ... 15
2.1.1.4 Mecanismo de regra flexível ... 16
2.1.1.5 Implantação em modo incorporado ... 16
2.1.1.6 Implantação baseada na rede ... 17
2.1.2 Funcionamento ... 18
2.2 Projeto OWASP Modsecurity Core Rule Set ... 21
2.2.1 Arquivos de regras do *Core Rule Set* ... 22
2.3 Ataques às aplicações *web* ... 27
2.3.1 LDAP *injection* ... 27
2.3.2 Server-Site Include injection ... 28
2.3.3 Universal PDF XSS ... 28
2.3.4 Email *injection* ... 29
2.3.5 HTTP *request smuggling/response splitting* ... 30
2.3.6 *Remote File Inclusion* ... 31
2.3.7 *Session fixation* ... 32
2.3.8 *Command injection* ... 32
2.3.9 SQL *injection* ... 33
2.3.10 XSS ... 34
2.3.11 *Directory traversal* ... 34
3 MATERIAIS E MÉTODOS ... 36
3.1 Metodologia de configuração da aplicação *web* ... 36
3.2 Metodologia dos testes ... 40
3.2.1 Testes de performance ... 40
3.2.2 Testes funcionais ... 44
3.2.3 Testes manuais ... 45
4 RESULTADOS ... 46
4.1 Testes de performance ... 46
4.2 Testes funcionais ... 50
4.3 Testes manuais ... 52
5 CONSIDERAÇÕES FINAIS ... 54
REFERÊNCIAS ... 55
ANEXOS ... 60

1 INTRODUÇÃO

A contínua expansão da internet traz diversos benefícios, porém deve haver cuidados para que a segurança na *web* não seja comprometida. Por parte dos fornecedores de serviços na *web*, são necessários diversos mecanismos de defesa, mas muitas vezes não protegem o usuário incluindo a própria empresa de ataques em seus recursos oferecidos na *web*. Em (CGISECURITY, 2009; EXTREMETECH, 2012; IFSEC, 2012; PCWORLD, 2012) citam exemplos de falhas de segurança em grandes empresas. Devido a esta constante insegurança proporcionada pela internet, e considerando a incapacidade de alguns filtros de pacotes analisarem o *payload* dos pacotes, foi que surgiram novas abordagens como o *modSecurity* (MODSECURITY, 2012), que será detalhado neste trabalho.

O *modSecurity* é um *Web Application Firewall* (WAF) que pode detectar ataques que *firewalls* de rede e detectores de intrusão não conseguem (KARAASLAN, 2004), visto que um WAF trabalha na camada de aplicação do modelo *Open Systems Interconnection* (OSI), dessa forma tem total acesso ao protocolo *HiperText Transfer Protocol* (HTTP). Um WAF posiciona-se entre o usuário e a aplicação *web*, interceptando todos os pacotes transmitidos entre eles. Nesta comunicação o WAF examina, com suas regras, se há algum tipo de ataque. Quando algum ataque é detectado, o *firewall* de aplicação pode tomar medidas corretivas ou evasivas, desde simplesmente desconectar a sessão de aplicativo atual ou uma abordagem mais sofisticada, como por exemplo, redirecionamento da sessão para um sistema *honeypot* criado para reunir detalhes das diversas técnicas de ataques (BYRNE, 2006).

WAF's são utilizados exclusivamente para proteger aplicações *web* possuindo diversos recursos, protegendo contra vários tipos de vulnerabilidades como também outras funcionalidades como rastreamento de IP, sessão, usuário, limitação de caracteres em determinados campos da aplicação web, integração com outras ferramentas, entre outros. Em SUTO (2011) há uma comparação de vários WAF's mostrando os pontos positivos e negativos na detecção de vulnerabilidades.

Considerando o exposto e seguindo esta linha de pesquisa, a seguir são definidos os objetivos do presente trabalho.

1.1 Objetivos

O objetivo deste trabalho é propor uma metodologia de configuração do *modSecurity* com a utilização do projeto OWASP[1] *ModSecurity Core Rule Set* (CRS) (OWASP, 2012), de maneira que direcione a proteção de cada tipo de vulnerabilidade em cada página *web* da aplicação, ou seja, em uma página que, por exemplo, não possua acesso ao banco de dados, não faz-se necessário proteção contra injeção SQL. Dessa forma, espera-se uma redução na utilização dos recursos onde o WAF encontra-se operando, tais como CPU, memória e canal de comunicação. Para uma verdadeira constatação do funcionamento, tanto do próprio *modSecurity* quanto da configuração personalizada, serão efetuados testes de performance para monitorar o consumo de recursos do servidor e testes funcionais para comprovar que a configuração realmente é eficaz. Serão realizados também testes manuais a fim de procurar possíveis falhas do CRS e propor uma solução do mesmo.

1.2 Organização

Este trabalho segue organizado da seguinte forma: o Capítulo 2 descreve o funcionamento e as principais características do *modSecurity*, do *Core Rule Set* e vulnerabilidades comumente encontradas em aplicações *web*; o Capítulo 3 é apresentada a proposta do trabalho, bem como os testes de performance, os testes funcionais e os testes manuais, descrevendo como foram realizados; o Capítulo 4 mostram os resultados de cada teste; e por fim, o Capítulo 9 são apresentadas as conclusões do trabalho.

[1] Open Web Application Security Project

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo serão abordados conceitos e definições relevantes para o entendimento da proposta deste trabalho. Serão abordados o *modSecurity* (seção 2.1), o *Core Rule Set* (seção 2.2) e vulnerabilidades comumente encontradas em aplicações *web* (seção 2.3).

2.1 *Modsecurity*

ModSecurity é um módulo do servidor *web Apache* (BARNETT, 2009) sendo possível instalar em ambientes *Unix, Windows* e *Macintosh*. Segundo (SOURCEFORGE, 2012), o *modSecurity* fornece proteção contra uma série de ataques contra aplicações *web*, permitindo a monitorização do tráfego HTTP e análise em tempo real com pouca ou nenhuma mudança na infraestrutura existente. A partir de 2012 o módulo está disponível também para os servidores web *nginx* (NGINX, 2012) e IIS (2012), sendo que para o *nginx* a versão ainda é beta.

2.1.1 Características

Dentre as diversas funcionalidades do *modSecurity*, destacam-se as seguintes características: *log* de tráfego HTTP, monitoração e detecção de ataque em tempo real, prevenção de ataques e *virtual patching*, mecanismo de regra flexível, implantação em modo incorporado e implantação baseada na rede. A seguir será apresentada cada característica descrita neste parágrafo.

2.1.1.1 *Log* de tráfego HTTP

Toda transação pelo protocolo HTTP pode ser enviada para o *log*, permitindo requisições e respostas completas. É possível refinar o que e quando será enviado ao *log*. Também é possível mascarar campos do cabeçalho HTTP antes que sejam enviados ao *log*, já que em algumas requisições ou respostas pode haver dados sigilosos que não devem aparecer no *log* (GITHUB, 2013). Em GITHUB (2012), encontra-se todas as informações referentes ao entendimento do *log*.

2.1.1.2 Monitoração e detecção de ataques em tempo real

O m*odSecurity* funciona como um detector de intrusão, no mesmo momento em que acontece o ataque é possível reagir ao evento suspeito (GITHUB, 2013). Isso acontece quando configurado, por exemplo, com a política padrão de bloquear a requisição quando encontrada uma ameaça, mas pode-se também redirecionar o usuário para outra página, entre outras técnicas.

2.1.1.3 Prevenção de ataques e *virtual patching*

Para prevenir ataques que podem atingir a aplicação *web*, há três abordagens geralmente utilizadas, são elas:

- Modelo de segurança negativo: Segundo SHEZAF (2007), o modelo baseia-se em um conjunto de regras que detectam anomalias, de forma que é permitido passar todo o tráfego pelo WAF, menos o que as regras não permitem.
- Modelo de segurança positivo: Segundo MACÊDO (2011) este modelo de segurança nega todas as transações por padrão, mas usa regras para

permitir que apenas as transações conhecidas sejam seguras. Com este modelo implementado é necessário conhecer a aplicação que está sendo protegida. Este modelo funciona melhor em aplicações que são bastante utilizadas mas com pouca atualização, sendo a manutenção deste modelo minimizada.

- Vulnerabilidades e deficiências conhecidas: funciona com regras, conhecido como *Virtual Patching*, sendo uma política para um dispositivo intermediário capaz de bloquear a exploração de vulnerabilidades (BARNETT, 2009). As falhas podem ser corrigidas por um *modSecurity* instalado em outro *host*, geralmente implementando um *proxy* reverso, mantendo os sistemas seguros enquanto uma correção apropriada não é aplicada para a aplicação. Mas para isso é necessário que a vulnerabilidade seja detectada e assim criar a regra que corrija a falha. A ideia do *Virtual Patching* é que a regra seja rapidamente criada, visto que a correção de uma vulnerabilidade em seu código fonte demora muitas vezes bastante tempo, deixando a aplicação exposta ou *offline* (SHINN, 2008).

2.1.1.4 Mecanismo de regra flexível

Este mecanismo é responsável por detectar as vulnerabilidades em uma aplicação *web*. É implementado a linguagem de regras do *modSecurity*, que é uma linguagem de programação especializada para trabalhar com transação de dados HTTP (GITHUB, 2013). As regras são feitas com expressões regulares em *perl* (PERL, 2012).

2.1.1.5 Implantação em modo incorporado

O *modSecurity* é um módulo, dessa forma ele trabalha incorporado no servidor *web*. De acordo com RISTIC (2010, p. 7), a implantação em modo

incorporado é uma ótima escolha para empresas onde a arquitetura está definida. O autor ainda lista algumas vantagens:

- Única maneira de proteger centenas de servidores *web*, devido que em algumas situações fica impraticável o uso de *proxy* reverso;
- Não estará implantando um ponto de falha;

O modo incorporado está representado na figura 1. A única desvantagem apontada por RISTIC (2010, p. 8) é que neste modo há o compartilhamento de recursos de *hardware* entre o servidor *web* e o *modSecurity*.

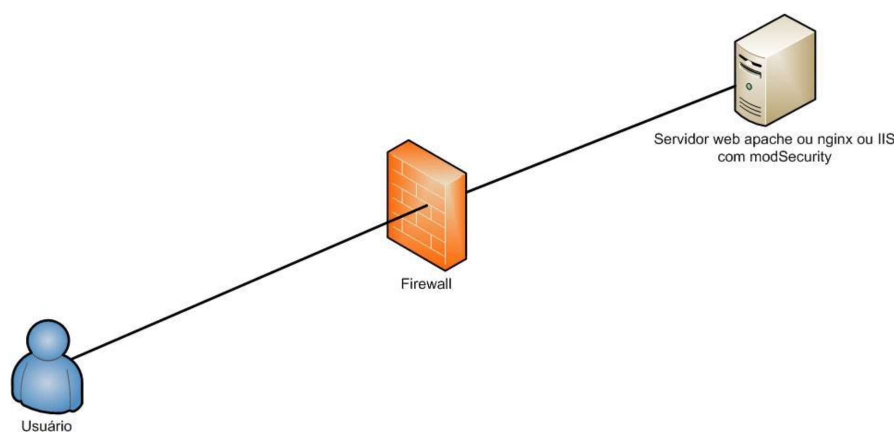

Figura 1 - Modo incorporado

2.1.1.6 Implantação baseada na rede

O *modSecurity* pode ser utilizado como um *proxy*, interceptando todas as requisições que passam por ele, sendo possível proteger diversas aplicações *web* com diferentes servidores *web* (TORRE, 2009). A figura 2 está representada este modo de implantação.

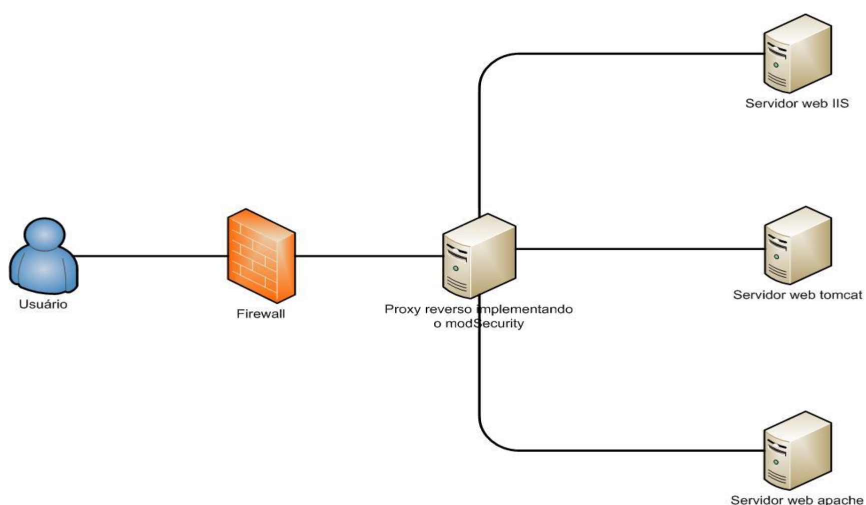

Figura 2 - Modo baseado na rede

2.1.2 Funcionamento

O funcionamento de uma requisição passa por cinco fases, conforme a figura 3. De acordo com MISCHEL (2009, p. 44), cada regra é processada em uma fase diferente.

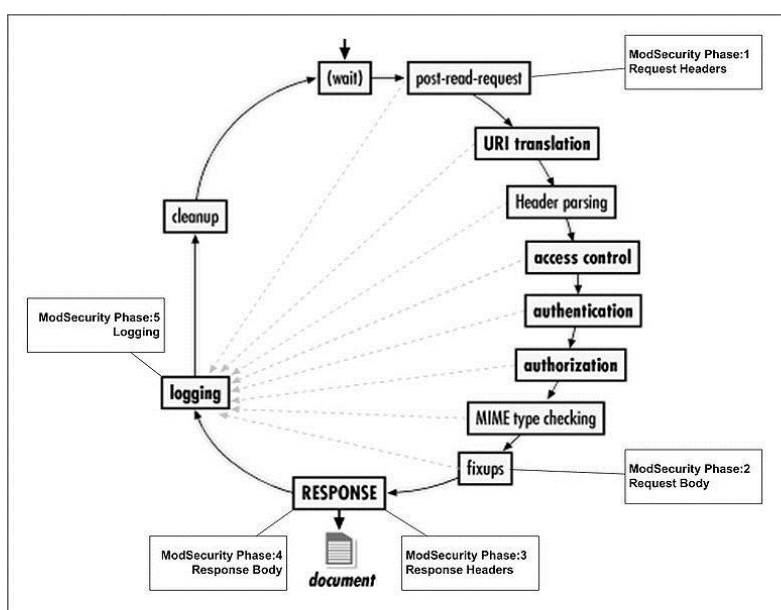

Figura 3 - Funcionamento de uma requisição no *modSecurity*
Fonte: GITHUB (2013)

Cada fase é detalhada a seguir:

- Fase 1 - cabeçalhos da requisição

De acordo com MISCHEL (2009, p. 18), esta fase ocorre logo após o servidor *web* ler os cabeçalhos da requisição. Um exemplo de dados disponíveis nesta fase está representado na figura 4, da linha 1 a 11, sendo que na linha 12 são os dados que aparecem somente na fase 2.

```
 1 POST /dvwa/login.php HTTP/1.1
 2 Host: localhost
 3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:10.0.1) Gecko/20100101 Firefox/10.0.1
 4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
 5 Accept-Language: pt-br,pt;q=0.8,en-us;q=0.5,en;q=0.3
 6 Accept-Encoding: gzip, deflate
 7 Connection: keep-alive
 8 Referer: http://localhost/dvwa/login.php
 9 Cookie: security=high; PHPSESSID=etv0a7us7j51suri74l1c73t64
10 Content-Type: application/x-www-form-urlencoded
11 Content-Length: 44
12 username=admin&password=password&Login=Login
```

Figura 4 - Exemplo de requisição

- Fase 2 – corpo da requisição

De acordo com RISTIC (2010, p. 8), esta é a principal fase de análise da requisição, ocorrendo após processado e recebido todo o corpo de resposta pelo servidor *web*. A maior parte das regras trabalha nesta fase, e para que os dados sejam capturados e comparados a opção *SecRequestBodyAccess* deve estar habilitada. O *modSecurity* suporta três tipos de codificação, que são:

- *application/x-www-form-urlencoded* – transferência dos dados de formulário;
- *multipart/form-data* – tranferência de arquivos;
- *text/xml* – transmissão de dados XML.

- Fase 3 – cabeçalhos da resposta

De acordo com GITHUB (2013), esta fase acorre pouco antes dos cabeçalhos de resposta serem enviados de volta ao cliente. Nenhuma das regras padrão do CRS trabalham nessa fase. O detalhamento das regras será abordado na próxima seção.

- Fase 4 – corpo da resposta

De acordo com MISCHEL (2009, p. 18), esta fase ocorre antes do corpo da resposta ser enviado de volta ao cliente. Para que os dados disponíveis nesta fase sejam capturados e analisados, a diretriz *SecResponseBodyAccess* deve estar habilitada.

- Fase 5 – *logging*

De acordo com GITHUB (2013), esta fase ocorre antes do envio dos dados para o *log*. RISTIC (2010, p. 8) diz que esta é a única fase que não pode ser bloqueada e que as regras nesta fase são executadas para controlar como o processo de *logging* é realizado.

Quando uma regra que trabalha na fase 1 detectar uma anomalia, e a política padrão for de bloquear e especificado na regra que deve bloquear (ação *deny*), a requisição será bloqueada e então encaminhada à fase 5 (envio dos dados para o *log*), como exemplificado na figura 5. Na linha 1 é iniciado a fase 1 (cabeçalhos da requisição) e então é chamado a regra (linha 2) que trabalha na fase 1, processada e não encontrado nenhuma anomalia (linha 3). A linha 4 inicia a fase 2 (corpo da requisição), chamando cada regra que trabalha nesta fase, na figura mostra apenas a regra que detectou a anomalia (linha 5). Na linha 6 mostra detalhes da regra que detectou e bloqueou a anomalia e nas linhas 7 e 8 não processando a fase 3 (cabeçalhos da resposta) e a fase 4 (corpo da resposta). Na linha 9 inicia a fase 5 (*logging*), enviando os dados para o *log*.

```
1 [28/Dec/2012:13:06:50 --0200] [localhost/sid#7fc6f208be70][rid#7fc6f25405b8][/dvwa/login.php][4] Starting phase REQUEST_HEADERS.
2 [28/Dec/2012:13:06:50 --0200] [localhost/sid#7fc6f208be70][rid#7fc6f25405b8][/dvwa/login.php][4] Recipe: Invoking rule 7fc6f207df90; [file "/etc/apache2/regras-
  modsecurity/modsecurity-crs_2.2.5/modsecurity_crs_10_setup.conf.example"] [line "25"] [id "200000"].
3 [28/Dec/2012:13:06:50 --0200] [localhost/sid#7fc6f208be70][rid#7fc6f25405b8][/dvwa/login.php][4] Rule returned 0.
4 [28/Dec/2012:13:06:50 --0200] [localhost/sid#7fc6f208be70][rid#7fc6f25405b8][/dvwa/login.php][4] Starting phase REQUEST_BODY.
5 [28/Dec/2012:13:06:50 --0200] [localhost/sid#7fc6f208be70][rid#7fc6f25405b8][/dvwa/login.php][4] Recipe: Invoking rule 7fc6f2231c98; [file "/etc/apache2/regras-
  modsecurity/modsecurity-crs_2.2.5/base_rules/modsecurity_crs_41_sql_injection_attacks.conf"] [line "64"] [id "981318"] [rev "2"].
6 [28/Dec/2012:13:06:50 --0200] [localhost/sid#7fc6f208be70][rid#7fc6f25405b8][/dvwa/login.php][1] Access denied with code 403 (phase 2). Pattern match "(^[\"'`\xc2\xb4
  \xe2\x80\x99\xe2\x80\x98;]+|[\"'`\xc2\xb4\xe2\x80\x99\xe2\x80\x98;]+$)" at ARGS:username. [file "/etc/apache2/regras-modsecurity/modsecurity-crs_2.2.5/base_rules/
  modsecurity_crs_41_sql_injection_attacks.conf"] [line "64"] [id "981318"] [rev "2"] [msg "SQL Injection Attack: Common Injection Testing Detected"] [data "Matched
  Data: ' found within ARGS:username: ' username=' 1 or 1-- &password=aaa"] [severity "CRITICAL"] [ver "OWASP_CRS/2.2.6"] [maturity "9"] [accuracy "8"] [tag "OWASP_CRS/
  WEB_ATTACK/SQL_INJECTION"] [tag "WASCTC/WASC-19"] [tag "OWASP_TOP_10/A1"] [tag "OWASP_AppSensor/CIE1"] [tag "PCI/6.5.2"]
7 [28/Dec/2012:13:06:50 --0200] [localhost/sid#7fc6f208be70][rid#7fc6f25405b8][/dvwa/login.php][4] Skipping phase 3 as request was already intercepted.
8 [28/Dec/2012:13:06:50 --0200] [localhost/sid#7fc6f208be70][rid#7fc6f25405b8][/dvwa/login.php][4] Skipping phase 4 as request was already intercepted.
9 [28/Dec/2012:13:06:50 --0200] [localhost/sid#7fc6f208be70][rid#7fc6f25405b8][/dvwa/login.php][4] Starting phase LOGGING.
```

Figura 5 - Parte de *log debug* de uma requisição maliciosa

Nesta seção foi abordado características e o funcionamento do *modSecurity*, a fim de mostrar como o módulo funciona. Na próxima seção será abordado o projeto da OWASP chamado *ModSecurity Core Rule Set*.

2.2 Projeto OWASP *Modsecurity Core Rule Set*

O projeto OWASP *ModSecurity Core Rule Set* (CRS) foi criado para que seus utilizadores possam ter um melhor proveito do *modSecurity* (OWASP, 2012). As regras estão localizadas em diversos arquivos, sendo alguns expecíficos para determinado tipo de vulnerabilidade. Ao contrário de sistemas de detecção e prevenção de intrusão, que se baseiam em assinaturas específicas para vulnerabilidades conhecidas, o CRS fornece proteção genérica contra vulnerabilidades desconhecidas, muitas vezes encontradas em aplicações *web* (GITHUB, 2013). O CRS é bastante comentado para permitir que seja utilizado como um guia de implantação passo a passo para o *modSecurity*.

O m*odSecurity* funciona através de regras, e estas regras precisam ser usadas para a detecção dos ataques. Desta forma o CRS foi criado, para que não seja necessário que os utilizadores do módulo criem regras para vulnerabilidades conhecidas, sendo um processo que necessita conhecimentos profundos sobre

diversas tecnologias. Este projeto é *open source* e fornece proteção efetiva fazendo uso do modelo de segurança negativa (SHEZAF, 2007).

Tendo como objetivo fornecer proteção genérica para as aplicações *web*, o CRS utiliza algumas das seguintes técnicas (OWASP, 2012):

- Proteção HTTP – Detecta violação no protocolo e utiliza a política de uso que foi definida localmente.
- Pesquisas em *blacklist* em tempo real – Utiliza reputação de IP de terceiros.
- Detecção de *malware* baseado na *web* – Identifica conteúdo malicioso utilizando a API *Google Safe Browsing*.
- Proteções HTTP de negação de serviço – Defesa contra HTTP *flooding* e ataques de negação de serviço pelo protocolo HTTP.
- Proteção a ataques *web* comuns – Detecção de ataques de segurança comuns em aplicações *web.*
- Detecção automatizada – Detecta *bots*, *crawlers*, *scanners* e outras atividades maliciosas.
- Integração com antivírus para *upload* de arquivos – Detecta arquivos maliciosos enviados para a aplicação *web.*
- Rastreamento de dados sensíveis – Controla o uso do cartão de crédito e vazamentos de blocos.
- Proteção contra *trojans* – Detecção de acesso a cavalos de troia.
- Identificação de defeitos de aplicativos – Alertas sobre erros de configuração de aplicativos.
- Detecção de erro e *hiding* – Camufla mensagens de erro enviadas pelo servidor.

2.2.1 Arquivos de regras do *Core Rule Set*

Nesta subseção serão apresentados os arquivos de regras do CRS, mostrando detalhadamente os arquivos de regras padrão. Há também a descrição

do arquivo de configuração. O CRS possui quatro diretórios de arquivos de regras, conforme são apresentados:

- *base_rules* – diretório principal de arquivos de regras.
- *experimental_rules* – diretório de arquivos de regras que podem não funcionar corretamente, pois estão em modo experimental. Possuem regras que protegem contra ataques DoS, HTTP *Parameter Pollution*, força bruta entre outros.
- *optional_rules* – diretório de arquivos de regras que oferecem proteção a outros tipos de vulnerabilidades e outras funcionalidades. Possui regras que protegem contra *Cross-Site Request Forgery* (CSRF), *comment spam*, roubo de sessão entre outros.
- *slr_rules* – diretório de arquivos de regras da *SpiderLabs Research*[2] (SLR), oferecendo proteção nas plataformas *Joomla*, *PHPbb* e *Wordpress*. Há também proteção contra ataques do tipo SQL *injection*, *Cross-Site Scripting*, *Local File Injection* e *Remote File Injection*.

A seguir é apresentado o diretório *base_rules*, constando cada nome de arquivo e detalhes de cada um, bem como o tipo de vulnerabilidade que cada um protege. Há também a apresentação do arquivo de configuração do CRS.

- Arquivo *modsecurity_crs_10_setup.conf*

Este é o arquivo de configuração do *Core Rule Set*, presente no Anexo B. Neste arquivo são setados as diretrizes de configuração, presentes no Anexo C, bem como regras que setam variáveis que são utilizados por regras, nos arquivos de regras.

- Arquivo *modsecurity_crs_20_protocol_violations.conf*

Neste arquivo há regras que checam se as requisições seguem padrões estabelecidos pelo protocolo HTTP. Há 23 regras habilitadas neste arquivo, sendo que 16 delas são executadas na fase 2, 6 regras na fase 1 e 1 regra na fase 5.

[2] https://www.trustwave.com/spiderlabs/

- Arquivo *modsecurity_crs_21_protocol_anomalies.conf*

Neste arquivo as regras fazem checagem dos cabeçalhos *Host*, *Accept*, *User-Agent* e *Content-Type* do protocolo HTTP, analisando se os mesmos existem e se estão vazios. Há também uma regra que checa se é usado endereço de IP no cabeçalho *Host* e também uma regra que coloca no *log* se uma das requisições foi rejeitada (código 400), porém esta última está, por padrão, comentada. Há 8 regras habilitadas neste arquivo, sendo que 7 delas são executadas na fase 2 e 1 regra na fase 1.

- Arquivo *modsecurity_crs_23_request_limits.conf*

Neste arquivo as regras limitam o tamanho de argumentos nas requisições, podem limitar o tamanho máximo para enviar um arquivo por exemplo. Por padrão as regras estão desabilitadas. Há 6 regras habilitadas neste arquivo, senfo que 5 delas são executadas na fase 2 e 1 regra na fase 1.

- Arquivo *modsecurity_crs_30_http_policy.conf*

Neste arquivo as regras definem os métodos de requisição permitidos (GET, HEAD, POST, entre outros), os tipos de conteúdo *(application/x-www-form-urlencoded, multipart/form-data* e *text/xml*), versões do protocolo HTTP (0.9, 1.0 e 1.1), as extensões de arquivos que são bloqueados (*.backup*, *.conf*, *.old*, entre outros) e os cabeçalhos do protocolo HTTP que são bloqueados (*Proxy-Connection*, *Content-Range*, *Lock-Token*, entre outros). Há 5 regras habilitadas neste arquivo, sendo que 3 delas são executadas na fase 2 e 2 regras na fase 1.

- Arquivo *modsecurity_crs_35_bad_robots.conf*

Neste arquivo as regras checam cabeçalhos do protocolo HTTP para detectar *scanners* de vulnerabilidades. Há 4 regras habilitadas neste arquivo e todas elas são executadas na fase 2.

- Arquivo *modsecurity_crs_40_generic_attacks.conf*

Neste arquivo há diversas regras que protegem contra os seguintes tipos de vulnerabilidades: OS *Command Injection*, ColdFusion *Injection*, LDAP *Injection*, SSI (*Server-Site Include*) *Injection*, UPDF XSS, Email *Injection*, HTTP *Request Smuggling*, HTTP *Response Splitting*, *Remote File Inclusion* (RFI), *Session Fixation*, *File Injection*, *Command Access*, *Command Injection* e PHP *Injection*. Há 23 regras habilitadas neste arquivo, sendo que 22 delas são executadas na fase 2 e 1 regra na fase 1.

- Arquivo *modsecurity_crs_41_sql_injection_attacks.conf*

Neste arquivo estão as regras para a detecção de ataques do tipo SQL. Há 54 regras habilitadas neste arquivo e todas são executadas na fase 2.

- Arquivo *modsecurity_crs_41_xss_attacks.conf*

Neste arquivo as regras detectam ataques do tipo *Cross-Site Scripting* (XSS). Há 105 regras habilitadas neste arquivo e todas são executadas na fase 2.

- Arquivo *modsecurity_crs_42_tight_security.conf*

Neste arquivo as regras detectam *Directory Traversal*. Por padrão elas estão desabilitadas, para utilizá-las basta habilitar o *Paranoid Mode* no arquivo de configuração. Há 1 regra habilitada neste arquivo, sendo executada na fase 2.

- Arquivo *modsecurity_crs_45_trojans.conf*

Neste arquivo as regras detectam *upload* de *trojans*. Há 3 regras habilitadas neste arquivo, sendo que 2 delas são executadas na fase 2 e 1 regra na fase 4.

- Arquivo *modsecurity_crs_47_common_exceptions.conf*

Neste arquivo há regras para remover falsos positivos, para o *Apache* e o *Adobe Flash Player*. Há 3 regras habilitadas neste arquivo e todas são executadas na fase 2.

- Arquivo *modsecurity_crs_48_local_exceptions.conf.example*

Neste arquivo não há nenhuma regra habilitada, pois o mesmo é utilizado para adicionar exceções. Essas exceções podem ser utilizadas quando ocorre algum falso positivo, podendo criar uma regra dentro deste arquivo para que não ocorra o problema novamente.

- Arquivo *modsecurity_crs_49_inbound_blocking.conf*

Neste arquivo as regras são utilizadas para detectar o score de anomalias nos dados de entrada, configurado no arquivo de configuração. Quando o score é alto uma medida é tomada, de acordo com a configuração. Há 2 regras habilitadas neste arquivo e ambas são executadas na fase 2.

- Arquivo *modsecurity_crs_50_outbound.conf*

Neste arquivo as regras analisam a resposta encaminhada pela aplicação. Se a mensagem que for retornada estiver nessas regras, a requisição será bloqueada. Há 29 regras habilitadas neste arquivo, todas são executadas na fase 4.

- Arquivo *modsecurity_crs_59_outbound_blocking.conf*

Neste arquivo a regra bloqueia os dados de saída, quando há altos escores de anomalias. Há 1 regra habilitada neste arquivo e é executada na fase 4.

- Arquivo *modsecurity_crs_60_correlation.conf*

Neste arquivo as regras analisam na fase de *logging*, tendo por objetivo se houve sucesso em algum ataque ou se houve tentativas sem sucesso. Há 5 regras habilitadas neste arquivo e todas são executadas na fase 5.

A seguir o leitor poderá conhecer características e funcionamentos de tipos de ataques conhecidos e executados em aplicações *web*.

2.3 Ataques às aplicações *web*

Nesta seção há a apresentação, descrição e exemplos de algumas vulnerabilidades, com o intuito de mostrar como funcionam os ataques às aplicações *web*. As vulnerabilidades escolhidas foram em relação aos ataques que as regras padrão do CRS estão aptas a detectar.

2.3.1 LDAP *injection*

Esta vulnerabilidade é semelhante a SQL *injection* (vide Seção 2.3.9), a diferença é que o LDAP é utilizado apenas para autenticação de usuários (ALONSO; et al, 2009). Um exemplo de uma vulnerabilidade LDAP *injection* está representado na figura 6. A injeção de código está no campo “USER”, sendo que o campo “PASSWORD” foi inserido qualquer caractere. Observe que na resposta o usuário aparece autenticado. Este ataque acontece da seguinte forma (ALONSO; et al, 2009):

Em uma requisição normal a busca no LDAP é *(&(USER = usuário) (PASSWORD = senha))*, mas com esta técnica fazendo uso do &, a busca fica *(&(USER = slisberger)(&))(PASSWORD = senha))*, sendo que é processado apenas *(&(USER = slisberger)(&))*, representando que a requisição é sempre verdadeira.

Requisição

ACCESS CONTROL

USER slisberger)(&))

PASSWORD ****************

Log In

Resposta

HOME

Steven Lisberger

Resources Documents

Figura 6 - Exemplo de LDAP injection
Fonte: Adaptado de (ALONSO; et al, 2009)

2.3.2 Server-Site Include injection

Esta vulnerabilidade permite a inclusão de arquivos dinâmicos simples em aplicações *web* (HOPE; WALTHER, 2009, p. 261), como também executar comandos nos sistema com a diretriz *exec*, mas para isso o *Server-Site Include* (SSI) deve estar habilitado no servidor *web*. Um exemplo de uma vulnerabilidade SSI está representado na figura 7. Na página do mecanismo de busca avançado possui o SSI habilitado e foi possível explorar executando o comando *mysql -V*.

Figura 7 - Exemplo de SSI *injection*

2.3.3 Universal PDF XSS

Esta vulnerabilidade pode ser explorada no *download* de qualquer arquivo PDF, utilizando *javascript* para efetuar o ataque (MODSECURITY, 2010). Na versão lançada em 2007 do *Adobe Reader* foi corrigida esta vulnerabilidade, descoberta em 2005. Um exemplo está representado na figura 8. A URL do arquivo está modificada, com uma cerquilha (#), a variável, e o código em *javascript*, caracterizando o ataque.

Usuários vulneráveis a esta vulnerabilidade são apenas os usuários que ainda utilizam uma versão anterior a 2007 do *Adobe Reader*.

Requisição

http://www.site.com/arquivos/exemplo.pdf#a=javascript:<script>alert("XSS")</script>

Resposta

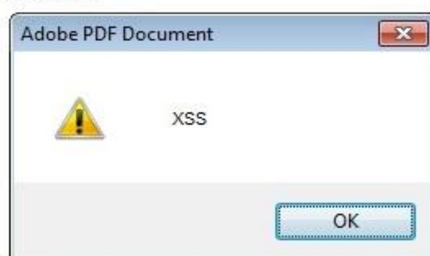

Figura 8 - Exemplo de ataque UPDF XSS

2.3.4 Email *injection*

Esta vulnerabilidade é explorada quando é possível mandar e-mail pela aplicação web, através dos protocolos *Internet Message Access Protocol* (IMAP) ou *Simple Mail Transfer Protocol* (SMTP). Essa vulnerabilidade é geralmente utilizada para adquirir e-mails por *spammers*[3] (HARWOOD; GONCALVES; PEMBLE, 2011, p. 180). Um exemplo de e-mail *injection* está representado na figura 9. Observe no campo *From* que após o primeiro e-mail tem os caracteres %0A, representando um *Line Feed* (\n) em formato codificado, e Cc, significando *Carbon Copy* que é uma cópia para outro e-mail. Seguindo ainda no campo *From*, encontra-se outro %0A e Bcc, este último significando *Blind Carbon Copy* que é uma cópia oculta para outro e-mail. Dessa forma enviando a mesma mensagem para diversos e-mails de forma anônima.

[3] Pessoas que enviam mensagens eletrônicas para várias outras pessoas sem que esteja autorizado.

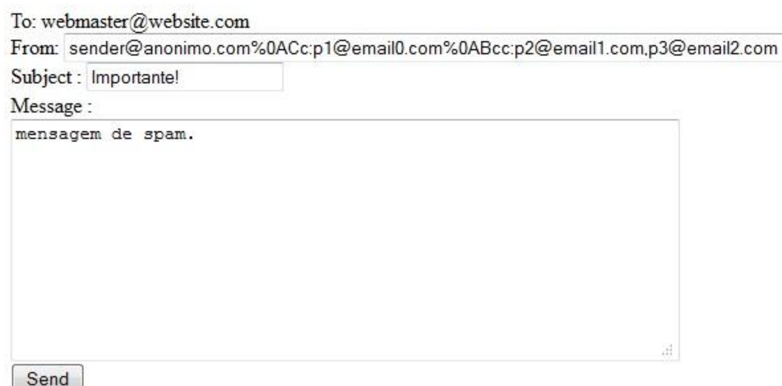

Figura 9 – Exemplo de email *injection*

2.3.5 HTTP *request smuggling/response splitting*

Estas vulnerabilidades permitem diversos ataques, como *Web Cache Poisoning* (*defacement*), sequestro de sessão, XSS e é possível contornar a proteção de um WAF (KLEIN, 2004; LINHART; et al, 2005).

Um exemplo de HTPP *Request Smuggling* está representado na figura 10.

> O ataque efetua uma requisição através do *proxy*, que está a frente da aplicação *web*. O *proxy*, ao receber a requisição, considera o último cabeçalho *Content-Lenght*, dessa forma ele considera que o corpo da requisição tem 44 *bytes*, sendo assim assumindo que a segunda requisição feita são os dados da linha 8 até a 10. Quando o *proxy* envia essa requisição à aplicação *web*, a mesma considera o primeiro cabeçalho *Content-Lenght*, sendo assim não possuindo corpo, e a segunda requisição sendo a partir da linha 8 (considerando que o método GET na linha 11 é passado para a aplicação *web* como um valor do cabeçalho “Bla” da linha 10). Dessa forma, as requisições que a aplicação *web* executou foram: “POST /exemplo.html” e “GET /auth.html”, enviando duas respostas com os conteúdos de cada página. O *proxy* combina os resultados das duas requisições que ele acha que foram enviadas pelo cliente – “POST /exemplo.html” e “GET /site.html”. Assumindo que a página é possível armazenar em *cache*, o *proxy* armazena os conteúdos de “auth.html” sobre a URL “site.html”. Todas as requisições feitas através do *proxy* na página “site.html”, receberá como resposta a página “auth.html” (LINHART; et al, 2005).

```
1 POST /exemplo.html HTTP/1.1
2 Host: www.exemplo.com
3 Connection: Keep-Alive
4 Content-Type: application/x-www-form-urlencoded
5 Content-Length: 0
6 Content-Length: 44
7 \r\n
8 GET /auth.html HTTP/1.1
9 Host: www.exemplo.com
10 Bla:
11 GET /site.html HTTP/1.1
12 Host: www.exemplo.com
13 Connection: Keep-Alive
14 \r\n
```

Figura 10 - Exemplo de HTTP *request smuggling*

2.3.6 Remote File Inclusion

Esta vulnerabilidade permite enviar arquivos de fora do servidor onde a aplicação está hospedada, sendo possível acessar dados fora da raiz do documento web, incluir scripts e outros tipos de arquivos de sites externos (KAPCZYŃSKI; TKACZ; ROSTANSKI, 2012, p. 191). Um exemplo dessa vulnerabilidade está na figura 11. O arquivo “r57.txt”, que possui código malicioso, é executado pelo servidor alvo e então mostrará os resultados que o código malicioso conseguiu retornar na própria página do domínio “site.com”.

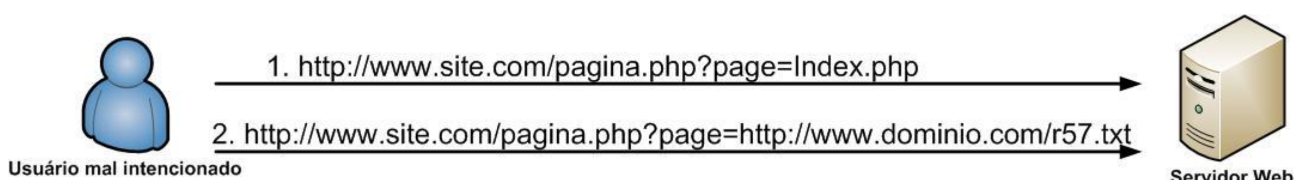

Figura 11 - Exemplo de ataque *remote file inclusion*

2.3.7 *Session fixation*

Esta vulnerabilidade quando explorada tem por objetivo forçar um usuário a utilizar um ID de sessão, podendo ser feita através de XSS. Após isso o usuário é forçado a fazer *login* e então a ID de sessão é capturada e utilizada pelo usuário mal intencionado (STUTTARD; PINTO, 2011, p. 537). Um exemplo de *Session Fixation* está representado na figura 12. Vale ressaltar que o ID de sessão recebida pela aplicação ao acessar a página é diferente do ID de sessão recebida depois de efetuado o *login*.

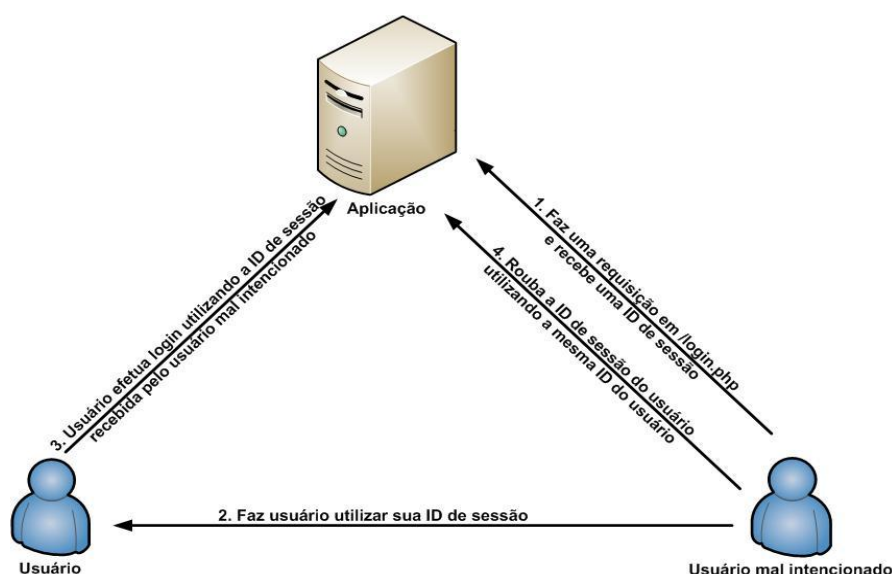

Figura 12 - Exemplo de *session fixation*
Fonte: STUTTARD; PINTO, 2011, p. 538

2.3.8 *Command injection*

Esta vulnerabilidade permite usuários enviarem comandos ou códigos maliciosos para o sistema para descobrir informações de usuários, do próprio sistema entre outros (SCHEMA, 2012, p. 227). Um exemplo de *command injection* está representado na figura 13. Nesse exemplo o comando que é executado é o

ping, mas com a utilização do *pipe* (“|”) foi possível realizar outro comando. Na linha de comando ficaria da seguinte forma: “ping | /bin/cat /etc/passwd”.

Requisição

Ping for FREE

Enter an IP address below:

| /bin/cat /etc/passwd [submit]

Resposta

Ping for FREE

Enter an IP address below:

[submit]

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

Figura 13 - Exemplo de *command injection*

2.3.9 SQL *injection*

Esta vulnerabilidade é explorada inserindo códigos em SQL nos parâmetros da aplicação *web*, sendo possível descobrir a versão do SQL utilizado, inserir/modificar/deletar informações nas tabelas. Qualquer procedimento que constrói instruções SQL pode ser vulnerável (CLARKE, 2012, p. 6). Um exemplo de SQL *injection* está representado na figura 14. O interessante do parâmetro “uuid()” é que os últimos 12 caracteres são formados pela MAC *Address* do servidor *Mysql*, podendo ser utilizado para algum fim.

Requisição

User ID:

ion all SELECT 1,uuid()-- - [Submit]

' union all SELECT 1,uuid(...

' union all SELECT 1,uuid()-- -

Resposta

User ID:

[Submit]

```
ID: ' union all SELECT 1,uuid()-- -
First name: 1
Surname: 54b38398-4a35-11e2-b1e2-0800277ccfe8
```

Figura 14 – Exemplo de sql *injection*

2.3.10 XSS

Esta vulnerabilidade é explorada inserindo códigos em *javascript*, podendo ser possível adquirir informações confidenciais do usuário alvo, manipular ou roubar *cookies* e executar códigos maliciosos no sistema do usuário (SPETT, 2005). Através desta técnica é realizado um ataque bastante conhecido chamado de *phishing*, que redireciona o usuário para uma página de *login* falsa para roubar suas credenciais do verdadeiro *website*. Um exemplo de ataque XSS está representado na figura 15. A vulnerabilidade foi explorada de forma a capturar e mostrar o *cookie* do usuário.

Figura 15 - Exemplo de XSS

2.3.11 Directory traversal

Esta vulnerabilidade é explorada de maneira a navegar em diretórios que não poderiam, sendo possível ler ou modificar arquivos sem ter o devido acesso (CWE, 2008). Um exemplo de *directory traversal* está representado na figura 16. Exemplo muito parecido com o RFI, porém aqui foi feita uma requisição de um arquivo do próprio sistema que executa o servidor web e ao invés de utilizar a barra (“/”) é utilizado o caractere codificado (%2f).

Resquisição
http://www.site.com/?page=%2fetc%2fgroup

Resposta
root:x:0: daemon:x:1: bin:x:2: sys:x:3: adm:x:4:ldap tty:x:5: disk:x:6: lp:x:7: mail:x:
www-data:x:33: backup:x:34: operator:x:37: list:x:38: irc:x:39: src:x:40: gnats:x:4
fuse:x:104: lpadmin:x:105:ldap ssl-cert:x:106: messagebus:x:107: mlocate:x:1
gdm:x:120: nopasswdlogin:x:121: ldap:x:1000: sambashare:x:122:ldap mysql

Figura 16 - Exemplo de *directory traversal*

Nesta seção foram introduzidos exemplos de vulnerabilidades que podem ser exploradas em aplicações *web*. A ideia foi ressaltar a relevância de atuação de um *firewall* na camada de aplicação.

Neste capítulo foram apresentados conceitos e o funcionamento sobre o *modSecurity*, *Core Rule Set* e sobre algumas vulnerabilidades comumente encontradas em aplicações *web*.

No próximo capítulo há a metodologia de configuração proposta, mostrando de que forma é realizada a configuração personalizada no *modSecurity*. Há também a descrição dos testes bem como das ferramentas utilizadas em cada tipo de teste.

3 MATERIAIS E MÉTODOS

Tendo em vista o pleno entendimento dos conceitos sobre o *modSecurity*, o *Core Rule Set* e as vulnerabilidades apresentadas no capítulo anterior, neste capítulo serão descritos de que forma foi realizada a configuração personalizada no *modSecurity*. Bem como a descrição dos testes realizados, mostrando as ferramentas utilizadas e de que forma foram conduzidos os testes.

3.1 Metodologia de configuração da aplicação *web*

Nesta subseção será apresentado a proposta da metodologia de configuração de vulnerabilidades para o *modSecurity*. A configuração personalizada da aplicação *web* foi realizada conforme o quadro 1.

O *modSecurity* possui a regra para detectar *upload* de *trojans* como também permite a integração com o antivírus *ClamAV*[4], sendo mais uma opção para incrementar a segurança em aplicações que tem a funcionalidade de *upload* de arquivos.

(continua)

Vulnerabilidade	Onde proteger
XSS	• Qualquer página onde é possível inserir algum texto em algum campo.
Injeção SQL	• Qualquer página que tenha acesso ao banco de dados SQL.
Injeção LDAP	• Qualquer página que tenha acesso ao banco de dados LDAP.
Server Side Include	• Utilize a regra apenas na página onde estiver habilitado o SSI.

[4] http://www.clamav.net

(conclusão)

Vulnerabilidade	Onde proteger
Email *Injection*	• Utilize esta regra onde há o formulário de envio de e-mail.
Session Fixation	• Utilize esta regra na página de autenticação.
OS Command Injection / Command Injection / Command Access	• Todos esses tipos de ataques são em relação a comandos do sistema, sendo que são necessários em toda a aplicação quando não se tem conhecimento dos pontos de injeção dessas falhas. • Geralmente ocorre em parâmetros de chamada de algum arquivo.
File Inclusion / Remote File Inclusion / Directory Traversal / PHP *Injection*	• Onde há parâmetros que chamem algum arquivo no servidor, podendo este ser uma página *web*.
ColdFusion Injection	• Regra específica para esta linguagem de marcação.
HTTP *Request Smuggling/Response Splitting*	• Se utilizar um *proxy* para acesso então utilize em toda a aplicação.
UPDF XSS	• Em páginas de *download* de arquivos PDF.

Quadro 1 - Como proteger a aplicação *web*

Na figura 17 está representada a maneira de descobrir quais métodos uma aplicação *web* aceita. Neste exemplo foi utilizado o comando *netcat* (*nc)*[5], mas funciona também com o comando *telnet*[6]. Para esta técnica funcionar é necessário que o método *OPTIONS* esteja habilitado no *website* alvo. Após conectar ao *website* na porta 80 foi colocado o método (*OPTIONS*), o diretório ("/") e o protocolo (HTTP)

[5] http://linux.die.net/man/1/nc

[6] http://linux.die.net/man/1/telnet

com a versão (1.1). Na outra linha foi colocado o cabeçalho (*Host*) acompanhado do domínio (www.ufsm.br). A importância de saber quais métodos uma aplicação *web* possui é que determinados ataques são efetuados com métodos específicos, por isso a importância de usar somente os métodos que a aplicação precisa, geralmente são os métodos *GET* e *POST*.

```
$nc www.ufsm.br 80
OPTIONS / HTTP/1.1
Host: www.ufsm.br

HTTP/1.1 200 OK
Date: Tue, 29 Jan 2013 02:58:36 GMT
Server: Apache/2.2.14 (Ubuntu)
Allow: GET,HEAD,POST,OPTIONS
Vary: Accept-Encoding
Content-Length: 0
Content-Type: text/html
```

Figura 17 - Descoberta de métodos que uma aplicação *web* suporta

Além dos métodos, é importante permitir somente os tipos de codificação que a aplicação usa, bem como a versão do protocolo HTTP, bloqueando as outras versões, visto que são utilizadas por *scanners* de vulnerabilidades.

A configuração padrão do *modSecurity* está representado na figura 18, sendo que foi realizada no arquivo *httpd.conf*. Após a inicialização da biblioteca e do próprio módulo, faz a verificação se o módulo está inicializado e também inclue o arquivo de configuração e todas as regras do diretório *base_rules* do CRS. Dessa forma são utilizadas todas as regras em todas as requisições para o servidor *web Apache*. Todos os arquivos de regras devem ser colocados entre as diretrizes *IfModule*, este verificando se o módulo está em uso, e então utilizar a diretriz *Include*, que inclui o arquivo de regras.

```
1 LoadFile /usr/lib/libxml2.so
2 LoadModule security2_module /usr/lib/apache2/modules/mod_security2.so
3       <IfModule mod_security2.c>
4               Include regras-modsecurity/modsecurity-crs_2.2.5/modsecurity_crs_10_setup.conf.example
5               Include regras-modsecurity/modsecurity-crs_2.2.5/base_rules/*.conf
6       </IfModule>
```

Figura 18 - Configuração padrão do *modSecurity*

A configuração personalizada no arquivo *httpd.conf* do *Apache* está presente no Anexo A. Nessa configuração foram utilizados apenas determinados arquivos de regras para serem utilizados em toda a aplicação. Para o direcionamento das regras para URL's específicas foi utilizado o *LocationMatch*[7], sendo possível utilizá-lo com expressões regulares. Sendo assim a realização da configuração pode ser trabalhosa dependendo da quantidade de URL's que a aplicação possui. Vale ressaltar que as regras das vulnerabilidades de LDAP *injection*, *Server Side Include*, Email *injection*, PHP *injection* e *ColdFusion injection* somente são úteis quando há este tipo de serviço disponível na aplicação, caso contrário haverá mais consumo de *hardware* (CPU e memória) tendo como consequência maior *delay* na resposta (vide Seção 4.1)

Observe que, no Anexo A, existem diversas regras que não estão sendo utilizadas, que foram removidas pela diretriz *SecRuleRemoveById*. Nunca comente uma regra, que não irá utilizar, diretamente no arquivo, devido que quando for atualizar o CRS todos os arquivos serão substituídos pelos novos, dessa forma todas as modificações efetuadas serão sobrescritas.

Os arquivos de regras incluídos para serem utilizados em toda a aplicação foram:

- *modsecurity_crs_20_protocol_violations.conf*
- *modsecurity_crs_21_protocol_anomalies.conf*
- *modsecurity_crs_30_http_policy.conf*
- *modsecurity_crs_50_outbound.conf*

Os três primeiros arquivos foram utilizados devido que verificam o protocolo HTTP por irregularidades, mantendo o padrão estabelecido por suas *Request for*

[7] http://httpd.apache.org/docs/2.2/mod/core.html#locationmatch

Comments (RFC). O último arquivo foi escolhido visto que quando ocorrer algum erro na aplicação, e este estiver nas regras, o mesmo não será exposto para o usuário. Na configuração foram removidas diversas regras deste último arquivo, pois não são úteis para a aplicação que está sendo protegida.

Toda a configuração personalizada do *modSecurity* foi realizada conforme a metodologia proposta, adicionando arquivos de regras em URL's específicas e removendo as regras que não são úteis. Dessa forma protegendo a aplicação corretamente, sendo que os resultados dos testes para a comprovação da metodologia proposta, será abordada no próximo capítulo.

3.2 Metodologia dos testes

Nesta seção é detalhado cada tipo de teste realizado, apontando de que forma foram conduzidos e quais ferramentas foram utilizadas. Houve três tipos de testes: testes de performance, para provar que a metodologia de configuração proposta aumenta o desempenho do *Apache*; testes funcionais, para provar que a proposta detecta todas as vulnerabilidades expostas pelas ferramentas; e testes manuais, para buscar falhas no CRS na detecção de vulnerabilidades.

3.2.1 Testes de performance

De acordo com TESTARME ([2010?]), testes de performance tem por objetivo avaliar a capacidade, robustez e disponibilidade de uma aplicação, avaliando seu desempenho principalmente em alta carga de trabalho e considerando seu comportamento em circunstâncias normais. Testes de performance são bastante úteis para descobrir como o sistema se comporta com determinada quantidade de usuários fazendo determinadas tarefas, sendo determinante na real necessidade de mais infraestrutura para uma determinada aplicação.

Em geral qualquer adição de serviços em um sistema computacional que disponibiliza funcionalidades de acesso, acaba gerando sobrecarga no sistema. Com a incrementação da segurança, a performance de acesso aos serviços são, consequentemente, afetados negativamente. Nessa concepção, foram efetuados testes de performance no servidor *web Apache* utilizando o módulo *modSecurity*, a fim de se ter uma ideia das diferenças de utilização de recursos com o módulo habilitado num primeiro momento e, desabilitado em outro. O objetivo principal é mensurar o custo de utilização de recursos que o módulo em questão adiciona ao ambiente.

Serão efetuados quatro tipos de testes de performance em uma aplicação web, conforme o quadro 2. O primeiro teste será realizado com o *modSecurity* desabilitado, o segundo teste com o *modSecurity* habilitado e utilizando todas as regras do *Core Rule Set* e em todas as páginas da aplicação *web* (padrão), o terceiro teste foi com o *modSecurity* habilitado mas somente com algumas regras e em algumas páginas (personalizado) e por fim o quarto teste com o *modSecurity* habilitado mas sem nenhuma regra. A ideia desses testes é demonstrar que uma melhor configuração do módulo para uma aplicação *web*, aumentará o desempenho do servidor *web Apache*.

Caso de teste	**Ambiente testado**
ModSecurity desabilitado	Instalação do *Apache* com suas configurações padrão, sem nenhum tipo de segurança implementado.
ModSecurity habilitado utilizando todas as regras padrão	Instalação do *Apache* com a utilização do *modSecurity* e do CRS. Utilizando a configuração padrão do *modSecurity*.
ModSecurity personalizado	Instalação do *Apache* utilizando *modSecurity* e o CRS, porém com a configuração do *modSecurity* personalizada.
ModSecurity habilitado sem regras	Instalação do *Apache* utilizando *modSecurity* sem o uso de regras.

Quadro 2 - Casos de teste e ambientes testados

O caso de teste com o *modSecurity* desabilitado é efetuado visto que será necessário na comparação com o caso de teste proposto, que é o personalizado. O teste com o módulo habilitado sem a utilização das regras é feito visto que será possível mensurar quanto é afetado negativamente no desempenho do *Apache* somente o *modSecurity*. Será possível mensurar quanto as regras afetam negativamente no desempenho do *Apache*, fazendo comparação com o caso de teste com o *modSecurity* habilitado com todas as regras do CRS e em todas as páginas.

Para os experimentos práticos serão considerados quatro aspectos, que são: a vazão da rede (em respostas por segundo e *kilobytes* por segundo), o tempo médio de resposta, a percentagem média de utilização da CPU e a quantidade de memória principal consumida pelo *Apache*. A vazão da rede é considerada visto que demonstra a velocidade do apache de enviar as respostas, o mesmo acontecendo com o tempo médio de resposta. A utilização da CPU e a quantidade de memória ocupada foram consideradas visto que é essencial para saber da real necessidade de *hardware*.

As ferramentas utilizadas para os testes foram o *JMeter*[8] e o *vmstat*[9]. De acordo com FÖRTSCH (2010) programas como *top* e *ps* não contém o verdadeiro valor utilizado por um processo, pois ambos assumem que um processo que utiliza memória compartilhada, como acontece com as bibliotecas, simplesmente adicionam a memória alocada pela biblioteca em cada processo-filho do *Apache*, retornando um valor errado. Uma maneira apontada foi utilizar o *vmstat*, que retorna a memória total consumida do sistema, e assim se tem um valor mais próximo da utilização de memória de um processo, no caso o *Apache*. A ferramenta *JMeter* foi escolhida para adquirir a vazão da rede e o tempo médio de resposta. O *vmstat* foi escolhido para monitorar a utilização da CPU e a memória principal.

O *JMeter* é um projeto da *Apache Software Foundation*[10], escrito em Java e é um dos aplicativos *open source* de testes distribuídos mais utilizados e completos disponíveis na internet (HALILI, 2008, p. 16). A ferramenta é bastante simples de usar, possui uma interface gráfica amigável, várias opções de coleta de dados e

---

8 http://jmeter.apache.org/

9 http://linuxcommand.org/man_pages/vmstat8.html

10 http://www.apache.org/

permite a geração de diversos gráficos. Esta ferramenta cria *threads* para realizar as requisições, são criadas quantas *threads* a memória permitir. Neste trabalho será considerado que cada *thread* criada pela aplicação, é um usuário.

A arquitetura dos testes está representada na figura 19. Conforme a figura, os testes foram efetuados utilizando o recurso de máquina virtual, sendo que o banco de dados e o servidor *web* foram instaladas em diferentes máquinas. A ferramenta *JMeter* ficou estabelecida na máquina principal, possuindo uma comunicação rápida com as máquinas virtuais.

Como mostra a figura 19, o *JMeter* se comunica com a aplicação *web* que está localizada na Maquina Virtual 1, que este se comunica com o banco de dados que está estabelecido na Máquina Virtual 2. A utilização desta arquitetura foi escolhida pelo motivo de que empresas, por razões de segurança, deixam o banco de dados em servidores diferentes que as aplicações *web* (APPSEC, 2008). O custo disso é o ligeiro aumento do *delay* que o usuário tem com a comunicação com a aplicação *web*, quando é necessário o acesso ao banco de dados.

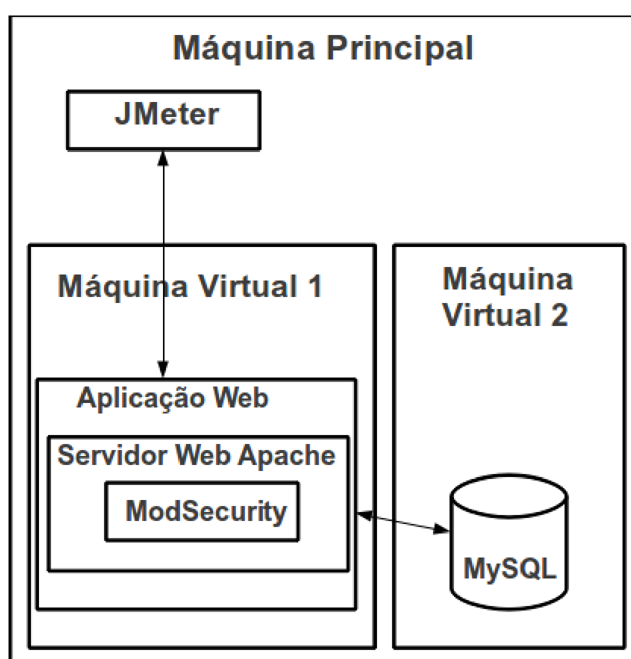

Figura 19 - Arquitetura dos testes

Todos os testes seguiram um padrão sequencial de requisições na aplicação *web*, sendo realizadas exatamente as mesmas requisições em todos os casos de teste. Cada usuário/*thread* faz 28 requisições em diferentes recursos da aplicação. Todas as requisições são válidas para os quatro casos de testes, ou seja, nenhuma das requisições o *modSecurity* trata como ameaça. O procedimento foi realizado desta forma visto que, caso uma das requisições seja tratada como ameaça, quando na utilização do *modSecurity*, o módulo retorna uma página de erro e a requisição não chega ao banco de dados, resultando diferença nos testes.

A configuração de *hardware* da máquina principal conta com um processador Intel Xeon 2.4 GHz com 16 núcleos e memória principal de 23 GB com sistema operacional Ubuntu 12.04. Em todos os testes a configuração da Máquina Virtual 1 (servidor *web*) conta com oito núcleos de 2.4 GHz e memória principal de 7 GB e a máquina virtual 2 (banco de dados) conta com três núcleos de 2.4 GHz e 7 GB de memória principal, ambos com sistema operacional Ubuntu 10.04.

3.2.2 Testes funcionais

O processo de teste funcional, também chamado de caixa preta ou *black-box*, tem como objetivo analisar se um *software* possui vulnerabilidades, não se preocupando com o código-fonte da aplicação testada (NETO, 2007). Esse tipo de teste pode ser automatizado, utilizando ferramentas que enviam dados maliciosos de entrada no *software* que está sendo testado e posteriormente analisa a saída dos dados, detectando ou não uma falha.

Os testes funcionais foram efetuados para comprovar a eficácia do *modSecurity* na utilização das regras do CRS. Houve três casos de teste, o primeiro com o *modSecurity* habilitado e utilizando todas as suas regras e em todas as páginas, o segundo com o *modSecurity* desabilitado e por fim, o terceiro, com o *modSecurity* utilizando a configuração personalizada.

Em todos os casos de teste foram utilizados as mesmas ferramentas, sendo todas *open source*. O conjunto de *softwares* utilizados foram os seguintes: *Arachni*[11], *Grendel-Scan*[12], *Skipfish*[13], *Uniscan*[14], *w3af*[15], *WebSecurify*[16] e *SQLMap*[17]*.* A única ferramenta que não é um *software* de teste funcional é o *SQLMap*, sendo utilizada para exploração de ataques SQL *Injection*.

Todos os resultados dos testes não foram considerados se haviam falsos positivos, que são considerados como vulnerabilidades mas na verdade não são, e falsos negativos, que são vulnerabilidades existentes mas que não foram encontrados pela ferramenta. Isso foi adotado pelo motivo de que o sistema utilizado nos testes é a *Damn Vulnerable Web Application* (DVWA)[18], sendo uma aplicação *web* implementada com diversas vulnerabilidades. Dessa forma, mesmo que as ferramentas não encontrem todas as vulnerabilidades, o importante é que com a utilização do *modSecurity* não haja detecção de vulnerabilidades, sendo o objetivo dos testes.

3.2.3 Testes manuais

O processo de testes manuais foi realizado visto que testes funcionais não encontram todas as vulnerabilidades. Nos trabalhos de (BAU; et al, 2010; KHOURY; et al, 2011; VIEIRA; ANTUNES; MADEIRA, 2009) mostram que testes funcionais possuem muitos falsos positivos, pois encontram apenas uma parcela das vulnerabilidades. Dessa forma foram realizados testes de ataques manuais para descobrir possíveis falhas no CRS para o *modSecurity*.

---

[11] http://arachni-scanner.com/

[12] http://sourceforge.net/projects/grendel/

[13] http://code.google.com/p/skipfish/

[14] http://uniscan.sourceforge.net/

[15] http://w3af.sourceforge.net/

[16] http://www.websecurify.com/

[17] http://sqlmap.org/

[18] http://www.dvwa.co.uk/

4 RESULTADOS

Este capítulo serão expostos os resultados dos testes de performance (seção 4.1), dos testes funcionais (seção 4.2) e dos testes manuais (4.3).

4.1 Testes de performance

Nesta subseção serão mostrados os resultados comparativos dos quatro casos de teste. Os resultados estão no quadro 3, separado por cada caso de teste sendo que foi efetuado cada um com 500, 1000 e 1500 usuários/*threads*. Como já citado anteriormente, os aspectos considerados foram a vazão, este sendo em respostas por segundo e *kilobytes* por segundo, o tempo médio de resposta em milissegundos, a percentagem média de uso da CPU e a memória principal utilizada em *megabytes* pelo *Apache*. Este último sendo somente a memória utilizada na execução do *Apache*, não levando em consideração a memória ocupada na inicialização do mesmo.

(continua)

Caso de teste	Usuários	Requisições	Vazão		TMR (ms)	CPU (%)	Memória (MB)
			resp/s	KB/s			
ModSecurity habilitado com regras	500	14000	441	732	19	59,03	193
	1000	28000	658	1092	257	94,16	474
	1500	42000	555	867	650	96,03	540
ModSecurity personalizado	500	14000	448	744	18	44,28	191
	1000	28000	755	1242	181	85,75	471
	1500	42000	606	928	474	91,83	504
ModSecurity habilitado sem regras	500	14000	450	754	20	39,64	187
	1000	28000	757	1269	171	71,21	462
	1500	42000	605	939	476	73,65	484

(conclusão)

Caso de teste	Usuários	Requisições	Vazão		TMR	CPU	Memória
			resp/s	KB/s	(ms)	(%)	(MB)
ModSecurity desabilitado	500	14000	453	760	20	35,65	183
	1000	28000	779	1306	150	67,19	452
	1500	42000	613	946	439	71,22	478

Quadro 3 - Resultados dos casos de teste

Na figura 20 está representada a vazão do *Apache* em respostas por segundo. Com a configuração do *modSecurity* personalizada há uma vantagem no melhor dos casos de até 15% sobre o *modSecurity* com a configuração padrão, representando 97 respostas por segundo mais rápida e no pior dos casos 2%, sendo 7 respostas por segundo mais rápido. Observa-se também que o desempenho diminui com 1500 usuários, isso acontece devido que a máquina virtual do servidor *web* é sobrecarregada, dessa forma demorando mais no processamento das requisições.

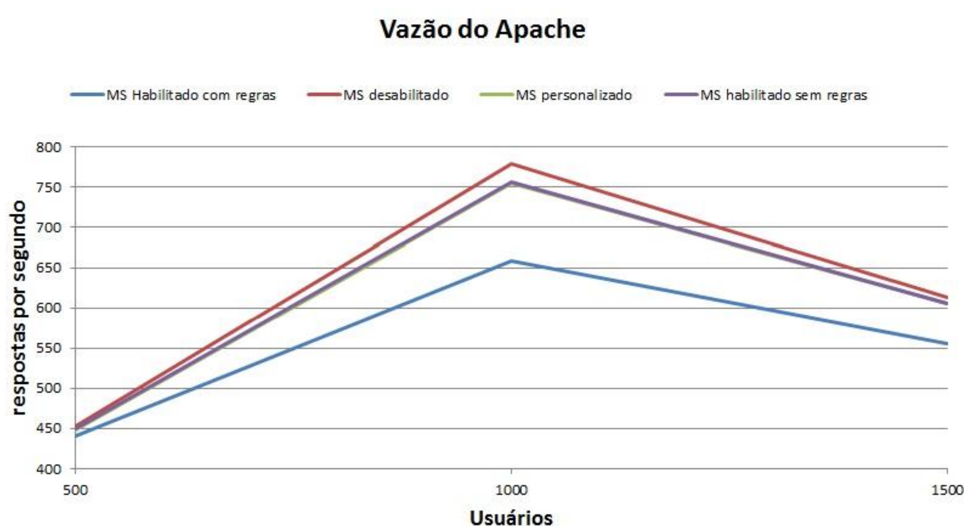

Figura 20 - Vazão do *Apache* em respostas por segundo

Na figura 21 está representada a vazão do apache em *kilobytes* por segundo. A utilização do módulo com a configuração personalizada em comparação com o mesmo habilitado e utilizando todas as regras em todas as páginas há uma diferença de 14% no melhor dos casos, representando 150 *kilobytes* por segundo mais rápido e no pior dos casos 2%, representando 12 *kilobytes* por segundo mais rápido. Da mesma forma e motivo que a vazão em respostas por segundo, neste aspecto também houve maior desempenho do *Apache* com a configuração personalizada do *modSecurity*. Observa-se também que com 1500 usuários acontece queda de desempenho, o motivo já esclarecido na figura anterior.

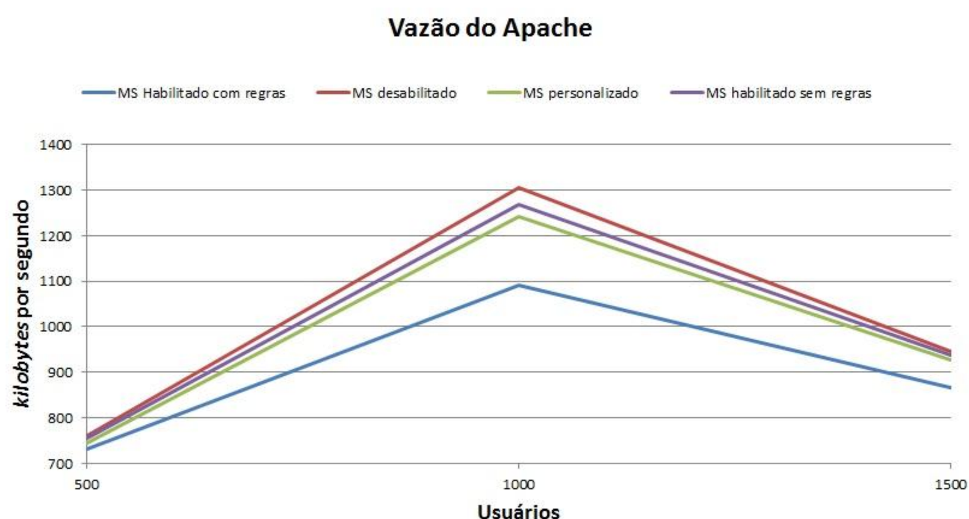

Figura 21 - Vazão do *Apache* em *kilobytes* por segundo

A figura 22 representa o tempo médio de resposta do *Apache*. O desempenho do *modSecurity* com a configuração personalizada em relação com o mesmo habilitado com sua configuração padrão há uma diferença, no melhor dos casos, de 37%, representando 176 ms abaixo, e no pior dos casos 6%, sendo 1 ms abaixo.

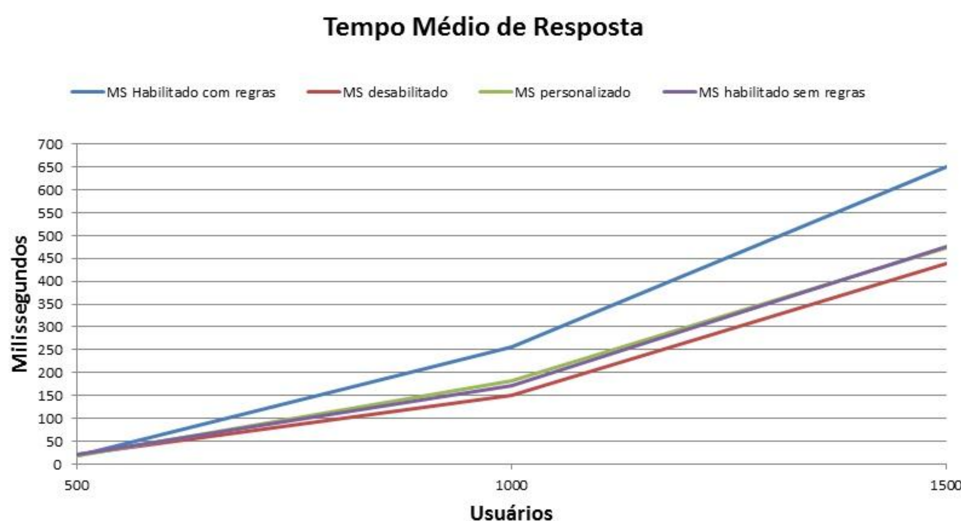

Figura 22 - Tempo médio de resposta do *Apache*

Na figura 23 está representado o consumo de tempo da CPU pelo *Apache*. O consumo com a utilização das regras (personalizado e padrão) é superior do que nos casos de teste sem o uso das regras. Isso acontece pelo motivo de que as regras precisam ser processadas pela CPU para procurar as vulnerabilidades nas requisições e respostas.

O consumo de tempo da CPU do *modSecurity* com a configuração personalizada em comparação com o mesmo habilitado e sua configuração padrão há, no melhor dos casos, uma diferença de 15% inferior e no pior dos casos de 4% inferior.

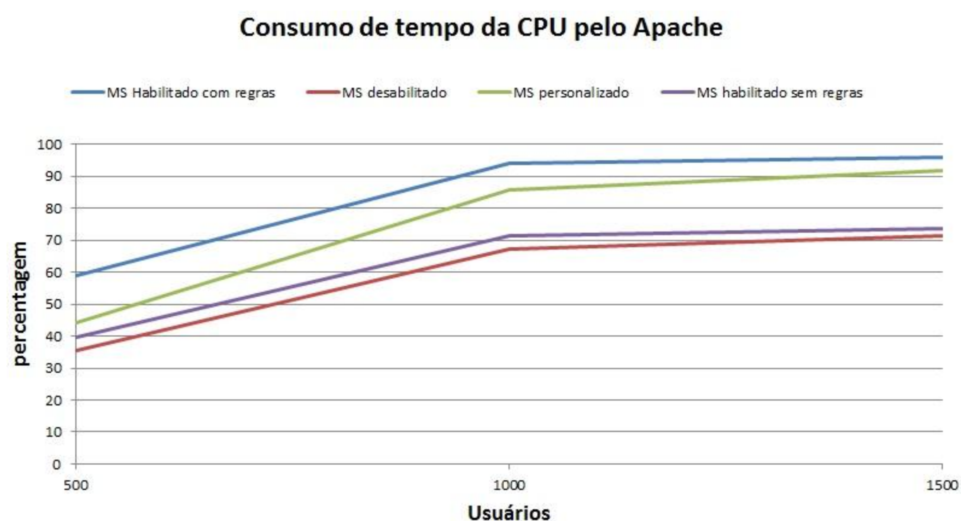

Figura 23 - Consumo de tempo da cpu pelo *Apache*

Na figura 24 está representado o consumo de memória principal pelo *Apache*. O módulo com sua configuração personalizada em comparação com o mesmo em sua configuração padrão há, no melhor dos casos, uma diferença de 7% inferior, representando 36 MB e no pior dos casos de 1%, representando 3 MB.

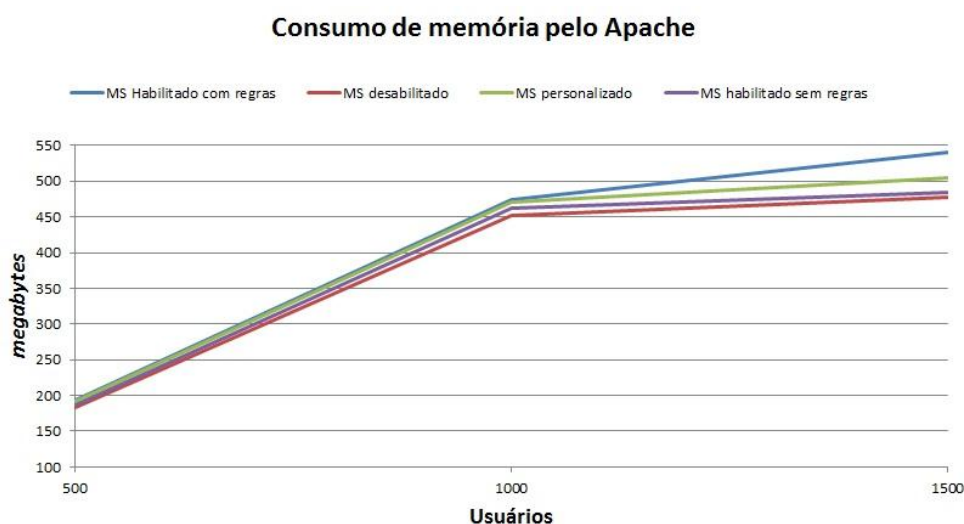

Figura 24 - Consumo de memória pelo apache

4.2 Testes funcionais

Os resultados dos testes funcionais com o *modSecurity* habilitado e utilizando todas as regras e em todas as páginas da aplicação *web* estão na tabela 1. Como é possível observar, nenhuma ferramenta foi capaz de detectar vulnerabilidades. A ferramenta *SQLMap*, que não está presente na tabela, também não obteve sucesso. Os comandos utilizados pelo *SQLMap* serão abordados no próximo caso de teste. Os quesitos de alta/média/baixa criticidade foram de acordo com o que cada aplicação considera do nível da vulnerabilidade encontrada.

Tabela 1 - Resultados com o *modSecurity* habilitado

Ferramenta	Alta Criticidade	Média Criticidade	Baixa Criticidade
Arachni	0	0	0
Grendel-Scan	0	0	0
Skipfish	0	0	0
Uniscan	0	0	0
w3af	0	0	0
WebSecurify	0	0	0

Os resultados referentes aos testes funcionais com o *modSecurity* desabilitado estão apresentados na tabela 2. A única ferramenta que não encontrou nenhuma vulnerabilidade foi o *Uniscan*, pelo motivo de que seu *crawler* encontrou, erroneamente, uma quantidade muito grande de URL's e no momento de verificar se haviam vulnerabilidades ocorreu falha de segmentação.

Tabela 2 - Resultados com o *modSecurity* desabilitado

Ferramenta	Alta Criticidade	Média Criticidade	Baixa Criticidade
Arachni	21	40	2
Grendel-Scan	0	0	2
Skipfish	588	11	589
Uniscan	0	0	0
w3af	23	59	17
WebSecurify	3	0	0

O *SQLMap* obteve resultados interessantes, com duas sequências de comandos, mostradas na tabela 3, conseguiu acesso com privilégios no banco de dados. O primeiro comando obteve a senha de acesso decriptada do usuário *root*. O segundo comando foi possível visualizar todas as bases de dados.

Tabela 3 - Comandos utilizados com o SQLMap

Sequência	Comando
1	python sqlmap.py -u "http://192.168.1.11/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit#" --users --password --string="Surname"
2	python sqlmap.py -u "http://192.168.1.11/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit#" --dbs

Os resultados com o *modSecurity* habilitado e utilizando uma configuração personalizada está apresentada na tabela 4. Como no teste com o módulo habilitado e com sua configuração padrão, nenhuma vulnerabilidade foi detectada. O SQLMap também não obteve sucesso na injeção SQL.

Tabela 4 - Resultados com o *modSecurity* personalizado

Ferramenta	Alta Criticidade	Média Criticidade	Baixa Criticidade
Arachni	0	0	0
Grendel-Scan	0	0	0
Skipfish	0	0	0
Uniscan	0	0	0
w3af	0	0	0
WebSecurify	0	0	0

4.3 Testes manuais

Com os testes realizados foi constatado um ataque SQL *injection* não detectado e ataques de *command injection*. Apesar de o CRS ser um projeto da OWASP bastante utilizado e frequentemente atualizado, é possível encontrar ataques não detectados. Estes testes visam somente encontrar falhas no CRS, não entrando no mérito da vulnerabilidade encontrada, ou seja, não é abordado se a

falha é de alta criticidade e não informando o que é possível fazer com a falha detectada.

O ataque SQL não detectado tem a seguinte estrutura: caractare’caractare”caractare. Um exemplo do ataque pode ser r’5ds”2, sendo possível praticamente colocar qualquer e quantos caracteres entre cada aspas. Dessa forma foi feita uma regra para detectar este ataque, apresentada a seguir.

```
SecRule ARGS:id "^(?:[(\w|\W)+?]'[(\w|\W)+?]\"[(\w|\W)+?])$"
"phase:2,deny,log,id:18"
```

Esta regra procura no argumento *id*, que é o parâmetro utilizado pela aplicação *web* para receber o valor injetado, um ou mais caracteres antes das aspas simples, depois das aspas simples e antes das aspas duplas e depois das aspas duplas, caracterizando o ataque que não era detectado. A expressão regular *\w* representa os caracteres de a-z, A-Z, 0-9 e _, o *\W* representa todos os caracteres menos os presentes em *\w*, e o *+?* representa um ou mais caracteres, que neste caso são os presentes em *\w* e *\W*.

Os ataques de *command injection* que não eram detectados foram pelo motivo de que se colocado o caminho inteiro do comando, não há a detecção. Por exemplo, quando inserido o comando *uname -a* é detectado a anomalia. Quando utilizado o caminho inteiro do comando, */bin/uname -a*, não ocorre a detecção.

A regra criada para a correta detecção dessas vulnerabilidades está abaixo:

```
SecRule ARGS:ip "\/sbin\/|\/bin\/" "id:25,phase:2,deny,log"
```

Após a utilização desta regra, comandos utilizando o diretório *sbin* ou *bin* são detectados e bloqueados.

5 CONSIDERAÇÕES FINAIS

Este trabalho abordou a eficiência do *modSecurity* e do CRS nos quesitos de detecção de vulnerabilidades e na viabilidade da implantação desta solução em um ambiente de alto fluxo de usuários. Após efetuados todos os testes, para a real constatação do funcionamento da solução em questão, a proposta de configuração obteve resultados satisfatórios.

O *modSecurity* mostrou-se robusto, viável para implantação em ambientes de alto processamento, não tendo problemas em manejar altas quantidades de requisições, visto que possui uma metodologia de busca de vulnerabilidades eficiente e de baixa carga nos recusos de *hardware*. O CRS detecta diversos tipos de vulnerabilidades, possuindo vários recursos, dessa forma sendo uma ótima escolha na melhoria da segurança de uma aplicação *web*.

Os resutados nos testes de performance mostraram que a configuração personalizada utiliza menos recursos de *hardware* resultando em respostas mais rápidas e vazão mais alta. Também foi possível concluir que quanto mais regras forem utilizadas nas requisições, há mais consumo de *hardware* e resultando em queda no desempenho do *Apache*.

Os testes funcionais mostraram que o CRS é uma ótima solução para a detecção de vulnerabilidades, visto que as ferramentas não detectaram nenhuma vulnerabilidade em sua utilização. A configuração personalizada do *modSecurity* obteve os mesmos resultados que com a configuração padrão, sendo que a proposta de configuração atingiu os objetivos de proteger corretamente a aplicação e aumentar o desempenho do servidor *web Apache*.

Os testes manuais expuseram duas falhas nas detecções de vulnerabilidades de SQL *injection* e *command injection* no CRS. Foi realizada a criação das regras e a correta detecção e bloqueio das anomalias.

REFERÊNCIAS

ALONSO, J. M. et al**. LDAP Injection Techniques**. In: Wireless Sensor Network, vol. 1, n. 4, 2009, p. 233-244.

APPSEC. **Database Security Best Practices**: 10 Steps to Reduce Risk. 2008. Disponível em: <http://www.appsecinc.com/news/newsletter/feb2008/page02.shtml>. Acesso em: 2 fev. 2013.

BARNET, R**. WAF Virtual Patching Challenge**: Securing WebGoat with ModSecurity. Breach Security, 2009.

BAU, J. et al. **State of the Art**: Automated Black-BoxWeb Application Vulnerability Testing. In: Security and Privacy (SP), 2010 IEEE Symposium, p. 332-345, Oakland, USA, Mai. 2010.

BYRNE, P. **Application firewalls in a defence-in-depth design**. Network Security, n. 9, p. 9-11, Set. 2009.

CGISECURITY. **Two XSS Worms Slam Twitter**. 2009. Disponível em: <http://www.cgisecurity.com/2009/04/two-xss-worms-slam-twitter.html>. Acesso em: 10 jan. 2013.

CLARKE, J. **SQL Injection Attacks and Defense**, Second Edition. Elsevier, 2012, 548 p.

CWE. **CWE-35**: Path Traversal. 2012. Disponível em: <http://cwe.mitre.org/data/definitions/35.html>. Acesso em: 18 dez. 2012.

EXTREMETECH. **How the PlayStation Network was Hacked**. 2011. Disponível em: <http://www.extremetech.com/gaming/84218-how-the-playstation-network-was-hacked>. Acesso em: 10 jan. 2013.

FÖRTSCH, T. **Measuring memory consumption**. 2010. Disponível em: <http://foertsch.name/ModPerl-Tricks/Measuring-memory-consumption/index.shtml>. Acesso em: 26 nov. 2012.

GITHUB. **Reference Manual**. 2013. Disponível em: <https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual>. Acesso em: 6 jan. 2013.

GITHUB. **ModSecurity 2 Data Formats**. 2012. Disponível em: <https://github.com/SpiderLabs/ModSecurity/wiki/ModSecurity-2-Data-Format>. Acesso em: 27 jan. 2013.

HALILI, E. **Apache JMeter**. Birmingham: Packt Publishing, 2008, 140 p.

HARWOOD, M.; GONÇALVES, M.; PEMBLE, M. **Security Strategies in Web Applications and Social Networking**. Jones & Bartlett Learning, 406 p.

HOPE, P.; WALTHER, B. **Web Security Testing Cookbook**. O'Reilly Media, 2009, 288 p.

IFSEC. **Two Facebook vulnerabilities**. 2012. Disponível em: <http://ifsec.blogspot.com.br/2012/03/two-facebook-vulnerabilities.html>. Acesso em: 10 jan. 2013.

IIS. **IIS**. 2012. Disponível em: <http://www.iis.net/>. Acesso em: 15 jan. 2013.

KAPCZYŃSKI, A.; TKACZ, E.; ROSTANSKI, M. **Internet**: Technical Developments and Applications. Spring Heidelberg, 2012, 286 p.

KARAASLAN , E.; TUGLULAR , T.; SENGONCA , H. **Enterprise Wide Web Application Security**: An Introduction . In: 13th Annual EICAR Conference, Mai. 1-4, 2004, Luxembourg, Luxembourg.

KHOURY, N. et al. **An Analysis of Black-Box Web Application Security Scanners against Stored SQL Injection**. In: Privacy, Security, Risk and Trust (PASSAT), 2011 IEEE Third International Conference e 2011 IEEE Third International Conference On Social Computing (SOCIALCOM) , p. 1095-1101, Out. 2011.

KLEIN, A. **HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics**. 2004. Disponível em: <http://www.cgisecurity.com/lib/whitepaper_httpresponse.pdf>. Acesso em: 18 dez. 2012.

LINHART, C. et al. **HTTP Request Smuggling**. 2005. Disponível em: <http://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf>. Acesso em: 18 dez. 2012.

MACÊDO, M. A.; QUEIROZ, R. J. G. B. de; DAMASCENO , J. C**. Uma Ferramenta Adaptativa Baseada em Agentes Móveis Inteligentes para Segurança de Aplicações Web**. In: VII Simpósio Brasileiro de Sistemas de Informação, 2011, Salvador. Anais do VII SBSI, 2011, Salvador, p. 105-116.

MISCHEL, M. **ModSecurity 2.5**: Securing your Apache installation and web applications. Packt Publishing, 2009, 201 p.

MODSECURITY. **PDF Universal XSS Protection**. 2010. Disponível em: <http://www.modsecurity.org/projects/modsecurity/apache/feature_universal_pdf_xss.html>. Acesso em: 18 dez. 2012.

MODSECURITY. **ModSecurity**: Open Source Web Application Firewall. 2012. Disponível em: <http://www.modsecurity.org/>. Acesso em: 7 jan. 2013.

NETO, A. C. D. **Introdução a Teste de Software**. Engenharia de Software Magazine, Rio de Janeiro, n. 1, p. 54-59, 2007.

NGINX. **NGINX**. 2012. Disponível em: <http://nginx.com/>. Acesso em: 15 jan. 2013.

OWASP. **OWASP ModSecurity Core Rule Set Project**. 2012. Disponível em: <https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project>. Acesso em: 12 abr. 2012.

OWASP. **OWASP TESTING GUIDE**. 2008. Disponível em: <https://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf >. Acesso em: 14 dez. 2012

PCWORLD. **Bug Bounty Hunters Reveal Eight Vulnerabilities in Google Services**. 2012. Disponível em: <http://www.pcworld.com/article/256151/bug_bounty_hunters_reveal_eight_vulnerabilities_in_google_services.html>. Acesso em: 10 jan. 2013.

PERL. **Perlre**. 2012. Disponível em: <http://perldoc.perl.org/perlre.html>. Acesso em:

17 jan. 2012.

QUALYS. **How to Protect Against Slow HTTP Attacks**. 2011. Disponível em: <https://community.qualys.com/blogs/securitylabs/2011/11/02/how-to-protect-against-slow-http-attacks>. Acesso em: 29 dez. 2012.

RISTIC, I. **ModSecurity Handbook**. Feisty Duck Limited, 2010, 340 p.

SASAKI, S. Q.; OKAMOTO, E.; YOSHIURA, H. **Security and Privacy in the Age of Ubiquitous Computing**. Springer Science Business Media, 2005, 616 p.

SCHEMA, M. **Hacking Web Apps**: Detecting and Preventing Web Application Security Problems. Elsevier, 2012, 276 p.

SHEZAF, O. **ModSecurity Core Rule Set**: An Open Source Rule Set for Generic Detection of Attacks against Web Applications. Breach Security, 2007.

SHINN, M**. Virtual Patching**. 2008. Disponível em: <http://www.prometheus-group.com/blogs/36-web-security/106-virtual-patching.html>. Acesso em: 23 abr. 2012.

SOURCEFORGE. **Reference Manual**. 2012. Disponível em: <http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual>. Acesso em: 11 abr. 2012.

SPETT, K**. Cross-Site Scripting**: Are your web applications vulnerable?. 2005. Disponível em: <http://www.rmccurdy.com/scripts/docs/spidynamics/SPIcross-sitescripting.pdf>. Acesso em: 18 dez. 2012.

STUTTARD, D.; PINTO, M. **The Web Application Hacker's Handbook**: Finding and Exploiting Security Flaws, Second Edition. Wiley Publishing, 2011, 914 p.

SUTO, L. **Analyzing the Effectiveness of Web Application Firewalls**. 2011. Disponível em: <https://www.cirosec.de/fileadmin/pdf/berichterstattung/2012/Analyzing_Effectivenes_ofWeb_Application_Firewalls.pdf>. Acesso em: 15 jan. 2013.

TESTARME. **Testar.me**: Desempenho de sistemas e softwares. [2010?]. Disponível

em: <http://www.testar.me/pages/testar_me_teste_de_performance.html>. Acesso em: 30 jan. 2013.

TORRE, D. **Web Application Firewall Primer with ModSecurity**. 2009. Disponível em: <http://www.atomicfission.com/index.php/white-papers/42-information-security/60-web-application-firewall-primer-with-modsecurity>. Acesso em: 28 jan. 2013.

TRUSTWAVE. **Detecting Malice with ModSecurity**: Open Proxy Abuse. 2011. Disponível em: <http://blog.spiderlabs.com/2011/03/detecting-malice-with-modsecurity-open-proxy-abuse.html>. Acesso em: 29 dez. 2012.

VIEIRA, M.; ANTUNES, N.; MADEIRA, H. **Using Web Security Scanners to Detect Vulnerabilities in Web Services**. In: IEEE/IFIP Intl Conf. on Dependable Systems and Networks, DSN 2009, Lisboa, Portugal, jun. 2009.

ANEXOS

Anexo A - Configuração personalizada no arquivo *httpd.conf* do *Apache*

```
LoadFile /usr/lib/libxml2.so
LoadModule security2_module /usr/lib/apache2/modules/mod_security2.so
	<IfModule mod_security2.c>
		Include /etc/apache2/regras-modsecurity/modsecurity-
	crs_2.2.5/base_rules/modsecurity_crs_20_protocol_violations.conf
		Include /etc/apache2/regras-modsecurity/modsecurity-
crs_2.2.5/base_rules/modsecurity_crs_21_protocol_anomalies.conf
		Include /etc/apache2/regras-modsecurity/modsecurity-
crs_2.2.5/base_rules/modsecurity_crs_30_http_policy.conf
		Include /etc/apache2/regras-modsecurity/modsecurity-
crs_2.2.5/modsecurity_crs_10_setup.conf.example
		Include /etc/apache2/regras-modsecurity/modsecurity-
crs_2.2.5/base_rules/modsecurity_crs_50_outbound.conf
			# Zope Information Leakage
			SecRuleRemoveById 970007
			# CF Information Leakage
			SecRuleRemoveById 970008
			# ISA server existence revealed
			SecRuleRemoveById 970010
			# Microsoft Office document properties leakage
			SecRuleRemoveById 970012 970903
			# CF source code leakage
			SecRuleRemoveById 970016
			# IIS default location
			SecRuleRemoveById 970018
			# The application is not available
			SecRuleRemoveById 970118
			# Weblogic information disclosure
			SecRuleRemoveById 970021
			# File or Directory Names Leakage
			SecRuleRemoveById 970011
```

```
		# IFrame Injection
		SecRuleRemoveById 981177 981000 981001 981003
		# Generic Malicious JS Detection
		SecRuleRemoveById 981004 981005 981006 981007
		# ASP/JSP source code leakage
		SecRuleRemoveById 970014
		# Statistics pages revealed
		SecRuleRemoveById 970002
		# IIS Errors leakage
		SecRuleRemoveById 970004 970904
	</IfModule>
####SQL INJECTION
<LocationMatch ^/dvwa/vulnerabilities/sqli/$>
	<IfModule mod_security2.c>
		SecDefaultAction "phase:1,deny,log"
		Include /etc/apache2/regras-modsecurity/modsecurity-
crs_2.2.5/base_rules/modsecurity_crs_41_sql_injection_attacks.conf
	</IfModule>
</LocationMatch>
####AUTENTICAÇÃO
<LocationMatch ^/dvwa/login.php$>
	<IfModule mod_security2.c>
		SecDefaultAction "phase:1,deny,log"
		Include regras-modsecurity/modsecurity-
crs_2.2.5/base_rules/modsecurity_crs_41_sql_injection_attacks.conf
		Include /etc/apache2/regras-modsecurity/modsecurity-
crs_2.2.5/base_rules/modsecurity_crs_40_generic_attacks.conf
		#Desabilitar ColdFusion Injection
		SecRuleRemoveById 950008
		#Desabilitar LDAP Injection
		SecRuleRemoveById 950010
		#Desabiltiar SSI Injection
		SecRuleRemoveById 950011
```

```
                #Desabilitar UPDF XSS
                SecRuleRemoveById 950018
                #Desabilitar Email Injection
                SecRuleRemoveById 950019
                #Desabilitar RFI
                SecRuleRemoveById 950117 950118 950119 950120
                #Desabilitar File Injection
                SecRuleRemoveById 950005
                #HTTP Request Smuggling
                SecRuleRemoveById 950012
                #HTTP Response Splitting
                SecRuleRemoveById 950910 950911
        </IfModule>
</LocationMatch>
####COMMAND EXECUTION
<LocationMatch ^/dvwa/vulnerabilities/exec/$>
        <IfModule mod_security2.c>
                SecDefaultAction "phase:1,deny,log"
                Include regras-modsecurity/modsecurity-
crs_2.2.5/base_rules/modsecurity_crs_40_generic_attacks.conf
                #Desabilitar ColdFusion Injection
                SecRuleRemoveById 950008
                #Desabilitar LDAP Injection
                SecRuleRemoveById 950010
                #Desabiltiar SSI Injection
                SecRuleRemoveById 950011
                #Desabilitar UPDF XSS
                SecRuleRemoveById 950018
                #Desabilitar Email Injection
                SecRuleRemoveById 950019
                #Desabilitar RFI
                SecRuleRemoveById 950117 950118 950119 950120
                #Desabilitar File Injection
```

```
            SecRuleRemoveById 950005
            #HTTP Request Smuggling
            SecRuleRemoveById 950012
            #HTTP Response Splitting
            SecRuleRemoveById 950910 950911
            #Session fixation
            SecRuleRemoveById 950009 950003 950000
      </IfModule>
</LocationMatch>
####FILE INJECTION
<LocationMatch ^/dvwa/vulnerabilities/fi/$>
      <IfModule mod_security2.c>
            SecDefaultAction "phase:1,deny,log"
            Include regras-modsecurity/modsecurity-
crs_2.2.5/base_rules/modsecurity_crs_40_generic_attacks.conf
            #Desabilitar ColdFusion Injection
            SecRuleRemoveById 950008
            #Desabilitar LDAP Injection
            SecRuleRemoveById 950010
            #Desabiltiar SSI Injection
            SecRuleRemoveById 950011
            #Desabilitar UPDF XSS
            SecRuleRemoveById 950018
            #Desabilitar Email Injection
            SecRuleRemoveById 950019
            #HTTP Request Smuggling
            SecRuleRemoveById 950012
            #HTTP Response Splitting
            SecRuleRemoveById 950910 950911
            #Session fixation
            SecRuleRemoveById 950009 950003 950000
      </IfModule>
</LocationMatch>
```

```
####XSS
<LocationMatch ^/dvwa/vulnerabilities/xss_(r|s)/$>
	<IfModule mod_security2.c>
		SecDefaultAction "phase:1,deny,log"
		Include regras-modsecurity/modsecurity-crs_2.2.5/base_rules/modsecurity_crs_41_xss_attacks.conf
	</IfModule>
</LocationMatch>
```

Anexo B - Configuração do arquivo de configuração do CRS

#Mecanismo de detecção
SecRuleEngine On
#Ação padrão a ser tomada
SecDefaultAction "phase:1,deny,log"
#Nome do servidor web a ser mostrado no cabeçalho de resposta
SecServerSignature "Apache"
#Acesso ao corpo das requisições
SecRequestBodyAccess On
#Armazenar até 128 KB por requisição na memória
SecRequestBodyInMemoryLimit 131072
#Regra que seta as variáveis utilizadas pelo arquivo *modsecurity_crs_30_http_policy.conf*
SecAction \
"id:'900012', \
phase:1, \
t:none, \
setvar:'tx.allowed_methods=GET POST', \
setvar:'tx.allowed_request_content_type=application/x-www-form-urlencoded|multipart/form-data', \
setvar:'tx.allowed_http_versions=HTTP/1.1', \
setvar:'tx.restricted_extensions=.asa/ .asax/ .ascx/ .axd/ .backup/ .bak/ .bat/ .cdx/ .cer/ .cfg/ .cmd/ .com/ .config/ .conf/ .cs/ .csproj/ .csr/ .dat/ .db/ .dbf/ .dll/ .dos/ .htr/ .htw/ .ida/ .idc/ .idq/ .inc/ .ini/ .key/ .licx/ .lnk/ .log/ .mdb/ .old/ .pass/ .pdb/ .pol/ .printer/ .pwd/ .resources/ .resx/ .sql/ .sys/ .vb/ .vbs/ .vbproj/ .vsdisco/ .webinfo/ .xsd/ .xsx/', \
setvar:'tx.restricted_headers=/Proxy-Connection/ /Lock-Token/ /Content-Range/ /Translate/ /via/ /if/', \
nolog, \
pass"
#Regra para checar se houve erro no corpo da requisição
SecRule REQBODY_ERROR "!@eq 0" \

"id:'200001', phase:2,t:none,log,deny,status:400,msg:'Failed to parse request body.',logdata:'%{reqbody_error_msg}',severity:2"

#Regra que analisa aspectos incomuns no corpo da requisição no formato *multipart/form-data*

SecRule MULTIPART_STRICT_ERROR "!@eq 0" \
"id:'200002',phase:2,t:none,log,deny,status:44, \
msg:'Multipart request body failed strict validation: \
PE %{REQBODY_PROCESSOR_ERROR}, \
BQ %{MULTIPART_BOUNDARY_QUOTED}, \
BW %{MULTIPART_BOUNDARY_WHITESPACE}, \
DB %{MULTIPART_DATA_BEFORE}, \
DA %{MULTIPART_DATA_AFTER}, \
HF %{MULTIPART_HEADER_FOLDING}, \
LF %{MULTIPART_LF_LINE}, \
SM %{MULTIPART_MISSING_SEMICOLON}, \
IQ %{MULTIPART_INVALID_QUOTING}, \
IP %{MULTIPART_INVALID_PART}, \
IH %{MULTIPART_INVALID_HEADER_FOLDING}, \
FL %{MULTIPART_FILE_LIMIT_EXCEEDED}'"

#Regra que detecta limite incompatível no corpo da requisição no formato *multipart/form-data*

SecRule MULTIPART_UNMATCHED_BOUNDARY "!@eq 0" \
"id:'200003',phase:2,t:none,log,deny,status:44,msg:'Multipart parser detected a possible unmatched boundary.'"

#Próximas duas diretrizes definem um limite na chamada da função *match()* na API do PCRE para evitar ataque de RegEx DoS

SecPcreMatchLimit 1000
SecPcreMatchLimitRecursion 1000

#Regra para mostrar algum erro interno

SecRule TX:/^MSC_/ "!@streq 0" \
"id:'200004',phase:2,t:none,deny,msg:'ModSecurity internal error flagged: %{MATCHED_VAR_NAME}'"

#Acesso ao corpo de resposta da requisição

```
SecResponseBodyAccess On
#Inspecionar os seguintes tipos de MIME
SecResponseBodyMimeType text/plain text/html
#Limita o tamanho do buffer de resposta em 512 KB
SecResponseBodyLimit 524288
#Inspeciona até o tamanho de 512 KB e após isso não há inspeção
SecResponseBodyLimitAction ProcessPartial
#Onde armanezar arquivos temporários
SecTmpDir /etc/apache2/secdir
#Onde armazenar os dados persistentes
SecDataDir /etc/apache2/secdir
#Somente enviar para o log erros 5xx e 4xx, menos o 404
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus "^(?:5|4(?!04))"
#Somente essas partes serão mostradas no log
SecAuditLogParts ABCFHZ
#Tipo de log serial
SecAuditLogType Serial
#Local onde é armazenado o log
SecAuditLog /var/log/modsecurity/modsec_audit.log
#O separador de argumentos na URL da aplicação
SecArgumentSeparator &
#Versão do cookie que é utililizado pela aplicação
SecCookieFormat 0
```

Anexo C - Diretrizes do *modSecurity*

SecAction

Esta diretriz é utilizada quando uma ação deve sempre ser tomada, independente das condições da operação.

Sintaxe: SecAction “ação1,ação2,ação3, ...”

Exemplo: SecAction "phase:1,id:'981056',t:none,nolog,pass,setuid:%{session.username}, setvar:session.sessionid=%{tx.sessionid}"

SecArgumentSeparator

Esta diretriz especifica qual caractere utilizar para separar o tipo de conteúdo “application/x-www-form-urlencoded”, que tem como padrão o caractere “&”.

Sintaxe: SecArgumentSeparator caractere

Exemplo: SecArgumentSeparator &

SecAuditEngine

Esta diretriz faz a configuração do mecanismo de log, em que envia para o log as transações completas. Os parâmetros possíveis são On (envia para o log todas as transações), Off (não envia nenhuma transação para o log) e RelevantOnly (envia apenas transações com *warning* ou erro e possuir um código de status presente na diretriz SecAuditLogRelevantStatus).

Sintaxe: SecAuditEngine parâmetro

Exemplo: SecAuditEngine RelevantOnly

SecAuditLog

Esta diretriz define o diretório para armazenar o arquivo de log.

Sintaxe: SecAuditLog “diretório”

Exemplo: SecAuditLog “/var/log/modsecurity/audit.log”

SecAuditLog2

Esta diretriz é igual a anterior (SecAuditLog), sendo que esta pode ser configurada para enviar o log para outro diretório.

Sintaxe: SecAuditLog “diretório”

Exemplo: SecAuditLog “/mnt/server/logs/modsecurity/audit.log”

SecAuditLogDirMode

Esta diretriz configura as permissões de diretórios criados para o modo de log concorrente (será abordado em SecAuditLogType), utilizando como parâmetro um valor em octal ou então utilizando o parâmetro “*default*”, significando a permissão 0600.

Sintaxe: SecAuditLogDirMode parâmetro

Exemplo: SecAuditLogDirMode 0640

SecAuditLogFileMode

Esta diretriz tem o funcionamento igual a anterior, a diferença é que configura a permissão dos arquivos de log.

Sintaxe: SecAuditLogDirMode parâmetro

Exemplo: SecAuditLogDirMode 0640

SecAuditLogParts

Esta diretriz define qual parte de cada transação será enviada para o arquivo de log, sendo possível enviar quantas partes forem necessárias. As partes de envio de informações de cada transação estão listadas abaixo:

- A – Cabeçalho do log de auditoria (obrigatório).
- B – Cabeçalho de requisição.
- C – Corpo da requisição.
- D – Reservado para cabeçalhos de resposta intermediário, não implementado.
- E – Corpo de resposta intermediário.

- F – Cabeçalhos de resposta final.
- G – Reservado para o real corpo de resposta, não implementado.
- H – Resumo do log de auditoria.
- I – Igual à parte C, exceto quando o tipo de conteúdo *multipart/form-data* é utilizado. Quando for este tipo de conteúdo, será enviado ao log um falso corpo de conteúdo do tipo *multipart/form-data*, apenas contendo informações sobre os parâmetros e não sobre os arquivos, sendo útil pois não armazena o conteúdo dos arquivos no log.
- J – Informações sobre os arquivos enviados utilizando o tipo de conteúdo *multipart/form-data*.
- K – Mostra todas as regras que detectaram alguma anomalia na requisição.
- Z – Significa o fim do log da requisição (obrigatório).

Sintaxe: SecAuditLogParts partes
Exemplo: SecAuditLogParts ABCFKHZ

SecAuditLogRelevantStatus

Esta diretriz configura qual código de status será considerado como relevante para ser enviado ao arquivo de log.
Sintaxe: SecAuditLogRelevantStatus “expressão regular em *perl*”
Exemplo: SecAuditLogRelevantStatus "^(?:5|4(?!04))"

SecAuditLogStorageDir

Esta diretriz configura o diretório onde será armazenado o log em modo concorrente (*Concurrent*).
Sintaxe: SecAuditLogStorageDir diretório
Exemplo: SecAuditLogStorageDir /var/log/modsecurity/audit_concurrent_logs/

SecAuditLogType

Esta diretriz configura o tipo de mecanismo de log. São dois os tipos: *Concurrent* e *Serial*. O primeiro é criado um arquivo de log por transação, sendo o único modo de enviar arquivos de log remotamente e o segundo será enviado todas as transações para um único arquivo de log.

Sintaxe: SecAuditLogType tipo_de_log

Exemplo: SecAuditLogType Serial

SecCacheTransformations (Rejeitado / Experimental)

Esta diretriz controla o armazenamento em cache de transformações, podendo aumentar o processamento de conjuntos de regras complexas. Possui dois parâmetros (*On* e *Off*) e opções (*incremental*, *maxitems*, *minlen*, *maxlen*). As opções estão abaixo listadas e explicadas:

- incremental:on|off - Armazenará em cache toda transformação
- maxitems:N – Não permite mais do que N transformações para serem armazenadas em cache. O valor 0 (zero) é interpretado como ilimitado.
- minlen:N - Não armazenar em cache a transformação se o tamanho da variável é menor do N bytes.
- maxlen:N - Não armazenar em cache a transformação se o tamanho da variável é maior do que N bytes. Zero significa ilimitado.

Sintaxe: SecCacheTransformations parâmetro [opções]

Exemplo: SecCacheTransformations On "minlen:64,maxlen:0"

SecChrootDir

Esta diretriz configura o diretório que será usado para prender o processo do servidor web. Este mecanismo (chroot) permite rodar serviços separadamente e protegidos.

Sintaxe: SecChrootDir diretório

Exemplo: SecChrootDir /chroot

SecComponentSignature

Esta diretriz adiciona uma assinatura no nome de um componente.

Sintaxe: SecComponentSignature “componente/x.y.z”

Exemplo: SecComponentSignature "Core RuleSet/2.2.7"

SecContentInjection

Esta diretriz permite a injeção de conteúdo utilizando as ações *append* e *prepend* (abordadas mais adiante). Os dois parâmetros de utilização são *On* e *Off*.

Sintaxe: SecContentInjection parâmetro

Exemplo: SecContentInjection On

SecCookieFormat

Esta diretriz seleciona o formato do *cookie* que será utilizado. Os parâmetros possíveis são 0 e 1, sendo que o valor 0 (referente a versão do cookie), é utilizado na maioria das aplicações e sendo o valor padrão.

Sintaxe: SecCookieFormat parâmetro

Exemplo: SecCookieFormat 0

SecDataDir

Esta diretriz é utilizada para armazenar dados persistentes (IP, sessão, entre outros). O diretório deve ser configurado antes de utilizar o initcol, setsid e setuid.

Sintaxe: SecDataDir diretório

Exemplo: SecDataDir /var/log/modsecurity/dados/

SecDebugLog

Esta diretriz é utilizada para setar o diretório para o arquivo do log de depuração.

Sintaxe: SecDebugLog diretório_até_arquivo

Exemplo: SecDebugLog /var/log/modsecurity/debug.log

SecDebugLogLevel

Esta diretriz configura o nível detalhamento dos dados do log de depuração. Os parâmetros estão detalhados abaixo:

- 0 – Não envia nada para o log.
- 1 – Erros (requisições interceptadas).
- 2 – *Warnings.*
- 3 – Notificações.
- 4 – Detalhes de como as transações são realizadas.
- 5 – Igual ao 4, só que inclui detalhes sobre cada pedaço de informação manipulada.
- 9 – Registra tudo, incluindo informações bastante detalhadas de depuração.

Sintaxe: SecDebugLogLevel parâmetro

Exemplo: SecDebugLogLevel 9

SecDefaultAction

Esta diretriz executa as ações padrões, que serão utilizados em cada regra. Toda a regra que conter a ação *deny*, e esta diretriz estiver configurada para bloquear, quando a regra detectar alguma anomalia é bloqueado a requisição.

Sintaxe: SecDefaultAction “ação1,ação2,ação3,...”

Exemplo: SecDefaultAction "phase:1,deny,log"

SecDisableBackendCompression

Esta diretriz desativa a compressão do servidor *backend*, deixando a compressão habilitada no *frontend*. É necessário utiliza-lo em modo de *proxy* reverso se desejar inspecionar as dados de resposta, visto que o servidor *backend* comprime os dados na resposta, dessa forma o modSecurity não consegue inspecionar. Os parâmetros possíveis são *On* e *Off*.

Sintaxe: SecDisableBackendCompression parâmetro

Exemplo: SecDisableBackendCompression *On*

SecHashEngine

Esta diretriz configura o mecanismo de *hash*. Possui dois parâmetros: *On* (processa dados de requisição de resposta) e *Off* (não processa nada).

Sintaxe: SecHashEngine parâmetro

Exemplo: SecHashEngine On

SecHashKey

Esta diretriz define a chave que será utilizada pelo *Hash Message Authentication Code* (HMAC). Há dois tipos de parâmetros, o parâmetro1 há *rand* (gera uma chave randômica) ou um texto que será gerado a chave, e o parâmetro2 há o *KeyOnly* (utiliza somente a chave), *SessionID* (utiliza o ID de sessão do usuário) ou *RemoteIP* (utiliza o IP remoto do usuário).

Sintaxe: SecHashKey parâmetro1 parâmetro2

Exemplo: SecHashKey “minha_chave” KeyOnly

SecHashParam

Esta diretriz define o nome do parâmetro que receberá o *hash* MAC. O mecanismo de *hash* do modSecurity irá adicionar um novo parâmetro para proteger elementos HTML contendo o *hash* MAC.

Sintaxe: SecHashParam texto

Exemplo: SecHashParam “hmac”

SecHashMethodRx

Esta diretriz configura que tipo de dado HTML o mecanismo de *hash* deve assinar baseado na expressão regular. É possível proteger os elementos HTML: *HREF*, *FRAME*, *IFRAME* e *FORM ACTION*, como também o cabeçalho de resposta *Location* quando o código de redirecionamento HTTP é enviado. Há dois parâmetros

nesta diretriz, o parâmetro1 possui cinco valores (abaixo) e o parâmetro2 é uma expressão regular.

- HashHref – Assina elementos “href=”.
- HashFormAction – Assina o elemento “form action=”.
- HashIframeSrc – Assina o elemento “iframe src=”.
- HashframeSrc – Assina o elemento “frame src=”.
- HashLocation – Assina o cabeçalho de resposta *Location*.

Sintaxe: SecHashMethodRx parâmetro1 parâmetro2

Exemplo: SecHashMethodRx HashHref "product_info|list_product"

SecHashMethodPm

Esta diretriz configura que tipo de dado HTML o mecanismo de *hash* deve assinar baseado no algoritmo de busca de *string*. Todos os parâmetros e recursos são iguais ao anterior (SecHashMethodRx), com a diferença que o parâmetro2 é colocado *strings*.

Sintaxe: SecHashMethodPm parâmetro1 “string1 string2 string3…”

Exemplo: SecHashMethodRx HashHref "product_info list_product"

SecGeoLookupDb

Esta diretriz define o diretório para o arquivo que contém a base de dados para pesquisas de localização geográfica. No site da *MaxMind*[19] é possível baixar gratuitamente uma base de dados.

Sintaxe: SecGeoLookupDb diretório

SecGeoLookupDb /opt/modsecurity/db/country.dat

SecGsbLookupDb

Esta diretriz define o diretório para o arquivo que contém a base de dados para pesquisas na API do Google Safe Browsing (GSB).

---

[19] http://www.maxmind.com/pt/opensource

Sintaxe: SecGsbLookupDb diretório

Exemplo: SecGsbLookupDb /opt/modsecurity/db/country_gsb.dat

SecGuardianLog

Esta diretriz configura um programa externo que receberá as informações sobre cada transação por um canal de log. Há um programa que tem compatibilidade com o *GuardianLog* que é o *httpd-guardian*[20]. Esta diretriz só vai funcionar quando o *httpd-guardian* estiver instalado e devidamente configurado.

Sintaxe: SecGuardianLog |diretorio

Exemplo: SecGuardianLog |/usr/local/apache/bin/httpd-guardian

SecHttpBlKey

Esta diretriz configura a chave do usuário registrado no projeto *Honeypot* HTTP-BL[21] para utilizar com o operador rbl (será aboradado na subsessão "operadores").

Sintaxe: SecHttpBlKey chave_de_acesso

Exemplo: SecHttpBlKey whdkfieyhtnf

SecInterceptOnError

Esta deretriz configura como responder quando o processamento de uma regra falhar. Quando a execução de um operador falhar (retorna < 0), a regra é deixada de lado e o processamento do restante continua normal. Há dois parâmetros, *On* e *Off*.

Sintaxe: SecInterceptOnError parâmetro

Exemplo: SecInterceptOnError *On*

SecMarker

---

20 http://apache-tools.cvs.sourceforge.net/viewvc/apache-tools/apache-tools/httpd-guardian?view=log

21 http://www.projecthoneypot.org/httpbl_api.php

Esta diretriz adiciona um marcador que pode ser utilizado na ação *skipAfter* (será abordado na subsessão "ações"). Esta diretriz não faz nada, apenas carrega um determinado parâmetro (ID ou texto).

Sintaxe: SecMarker parâmetro

Exemplo: SecMarker INICIO_CHECAGEM_HOST

SecPcreMatchLimit

Esta diretriz define o limite de combinações na biblioteca Perl Compatible Regular Expressions (PCRE). Esta diretriz é importante, pois se não definido o limite pode ocorrer um ataque de DOS no próprio WAF. Isso acontece quando um conteúdo muito complexo que é analisado pelas regras, fazendo que o WAF pare de analisar as requisições.

Sintaxe: SecPcreMatchLimit valor

Exemplo: SecPcreMatchLimit 1000

SecPcreMatchLimitRecursion

Esta diretriz define o limite de recursividade de combinações na biblioteca PCRE, possuindo a mesma importância da diretriz anterior.

Sintaxe: SecPcreMatchLimitRecursion valor

Exemplo: SecPcreMatchLimitRecursion 1000

SecReadStateLimit

Esta diretriz estabelece um limite por endereço IP de quantas conexões estão autorizados a estar em estado SERVER_BUSY_READ (estado indicando que o servidor está lendo dados de um cliente). Esta medida é efetiva contra ataque do tipo *Slowloris*[22] HTTP DoS.

Sintaxe: SecReadStateLimit limite

Exemplo: SecReadStateLimit 50

[22] http://ha.ckers.org/slowloris/

SecSensorId

Esta diretriz define um ID do sensor que estará presente no log na parte H.

Sintaxe: SecSensorId texto

Exemplo: SecSensorId WAFSensor01

SecWriteStateLimit

Esta diretriz estabelece um limite por endereço IP de quantas conexões estão autorizados a estar em estado SERVER_BUSY_WRITE (estado indicando que o servidor está escrevendo dados para um cliente). Esta medida é eficaz contra ataques de DoS lentos no corpo da requisição (r-u-dead-yet/RUDY[23]).

Sintaxe: SecWriteStateLimit limite

Exemplo: SecWriteStateLimit 50

SecRequestBodyAccess

Esta diretriz configura se os corpos de requisição serão armazenados em buffer e processados pelo ModSecurity. Esta é uma diretriz muito importante, pois habilita a inspeção de parâmetros POST (envio de arquivo, autenticação), ou seja se não estiver habilitado o modSecurity não será capaz de analisar vulnerabilidades na autenticação e no envio de arquivos, abrindo uma enorme brecha na segurança. Os parâmetros possíveis são *On* e *Off*.

Sintaxe: SecRequestBodyAccess parâmetro

Exemplo: SecRequestBodyAccess On

SecRequestBodyInMemoryLimit

[23] http://chaptersinwebsecurity.blogspot.com.br/2010/11/universal-http-dos-are-you-dead-yet.html

Esta diretriz configura o tamanho máximo do corpo da requisição que o modSecurity irá armazenar na memória. Se o tamanho máximo é atingido, o corpo da requisição será enviado para um arquivo temporário no disco. O valor padrão é de 131072 bytes.

Sintaxe: SecRequestBodyInMemoryLimit valor_em_bytes

Exemplo: SecRequestBodyInMemoryLimit 65536

SecRequestBodyLimit

Esta diretriz configura o tamanho máximo do corpo de requisição que o modSecurity aceitará para *buffering*. Qualquer coisa acima do limite será rejeitada com código de status 413 (Entidade solicitada muito grande). O valor padrão é de 128 MB.

Sintaxe: SecRequestBodyLimit limite_em_bytes

Exemplo: SecRequestBodyLimit 134217728

SecRequestBodyNoFilesLimit

Esta diretriz configura o tamanho máximo do corpo de requisição que o modSecurity aceitará para *buffering*, excluindo o tamanho de todos os arquivos sendo transportados na requisição. Esta diretriz é útil para reduzir a susceptibilidade a ataques de negação de serviço, quando alguém está enviando corpos de requisição muito grandes. O valor padrão é de 1 MB.

Sintaxe: SecRequestBodyNoFilesLimit valor_em_bytes

Exemplo: SecRequestBodyNoFilesLimit 131072

SecRequestBodyLimitAction

Esta diretriz controla o que acontece uma vez que o limite do corpo da requisição, configurado com SecRequestBodyLimit, é encontrado. Por padrão, como já falado anteriormente, é enviado um código de erro, mas se a politica estiver como DetectionOnly (apenas detectar a possível ameaça, não bloquear) será enviado o código de erro, não sendo o objetivo da política. Para que isso não aconteça, pode-

se utilizar o parâmetro *ProcessPartial*, no qual irá analisar o corpo da requisição até o limite, e o restante deixará passar. Os parâmetros possíveis são *Reject* (enviar código de erro) ou *ProcessPartial*.

Sintaxe: SecRequestBodyLimitAction parâmetro

Exemplo: SecRequestBodyLimitAction ProcessPartial

SecResponseBodyLimit

Esta diretriz configura o tamanho máximo do corpo de resposta que será aceito para *buffering*. Acima do valor padrão de 512 KB, será enviado o código de erro 500 (Erro interno do servidor).

Sintaxe: SecResponseBodyLimit limite_em_bytes

Exemplo: SecResponseBodyLimit 524228

SecResponseBodyLimitAction

Esta diretriz controla o que acontece uma vez que o limite do corpo de resposta, configurado com SecResponseBodyLimit, é encontrado. Esta diretriz funciona da mesma forma que SecRequestBodyLimitAction, só que neste caso é o corpo de resposta. Os parâmetros são os mesmos (*Reject* e *ProcessPartial*).

Sintaxe: SecResponseBodyLimitAction parâmetro

Exemplo: SecResponseBodyLimitAction ProcessPartial

SecResponseBodyMimeType

Esta diretriz configura que tipos de *Multipurpose Internet Mail Extensions* (MIME) devem ser considerados para enviar ao buffer o corpo de resposta. Podem ser colocados vários tipos de MIME, o padrão são os tipos *text/plain* e *text/html*.

Sintaxe: SecResponseBodyMimeType tipo_MIME1 tipo_MIME2 tipo_MIME3 ...

Exemplo: SecResponseBodyMimeType text/plain text/html text/xml

SecResponseBodyMimeTypesClear

Esta diretriz limpa a lista de tipos MIME considerados para o buffer do corpo de resposta, permitindo começar a preencher a lista a partir do zero.

Sintaxe: SecResponseBodyMimeTypesClear

Exemplo: SecResponseBodyMimeTypesClear

SecResponseBodyAccess

Esta diretriz configura se os corpos de resposta devem ser enviados ao *buffer*. Esta diretriz é necessária se você pretende inspecionar respostas HTML e implementar o bloqueio de resposta. Os parâmetros possíveis são *On* (envia ao *buffer* as respostas que estão configuradas em SecResponseBodyMimeType) e *Off* (não envia para o *buffer*).

Sintaxe: SecResponseBodyAccess parâmetro

Exemplo: SecResponseBodyAccess *On*

SecRule

Esta diretriz cria uma regra que irá analisar as variáveis selecionadas utilizando o operador selecionado. É a diretriz mais utilizada, visto que é usada para criar todas as regras para a detecção de anomalias. O parâmetro “AÇÕES” era opcional até a versão 2.6.8, tornando-se obrigatório a partir de então pelo motivo de que toda regra deve ter um *id*.

Sintaxe: SecRule VARIÁVEIS OPERADORES AÇÕES

Exemplo: SecRule REMOTE_ADDR "@ipMatch 192.168.1.100" "id:'900005', phase:1, t:none, ctl:ruleEngine=DetectionOnly, setvar:tx.regression_testing=1, nolog, pass"

SecRuleInheritance

Esta diretriz configura se o contexto atual herdará as regras do contexto pai. A herança acontece quando há uma configuração personalizada em cada *container*[24], conforme figura 25, em que se estiver *On* e não for na fase 1 (phase:1), serão utilizados as configurações globais. Caso estiver *Off*, será necessário configurar da forma que desejar.

```
 1 LoadFile /usr/lib/libxml2.so
 2 LoadModule security2_module /usr/lib/apache2/modules/mod_security2.so
 3 SecDefaultAction "phase:2,deny,log,redirect:http://localhost"
 4 <VirtualHost *:80>
 5 [..]
 6        <LocationMatch ^/dvwa/vulnerabilities/sqli/$>
 7                <IfModule mod_security2.c>
 8                        SecRuleInheritance Off
 9                        SecDefaultAction "phase:1,deny,log,redirect:http://localhost/erro"
10                        Include base_rules/modsecurity_crs_41_sql_injection_attacks.conf
11                </IfModule>
12         </LocationMatch>
13 [..]
14 </VirtualHost>
15 <VirtualHost *:80>
16 [..]
17        <LocationMatch ^/dvwa/vulnerabilities/sqli/$>
18                <IfModule mod_security2.c>
19                        SecRuleInheritance On
20                        Include base_rules/modsecurity_crs_41_sql_injection_attacks.conf
21                </IfModule>
22         </LocationMatch>
23 [..]
24 </VirtualHost>
```

Figura 25 - Herança de regras

Sintaxe: SecRuleInheritance parâmetro

Exemplo: SecRuleInheritance Off

SecRuleEngine

Esta diretriz configura o mecanismo de regras, em que é possível três parâmetros: *On* (processa as regras), *Off* (não processa as regras) e *DetectionOnly* (processa as regras mas não executa as ações disruptivas (*block*, *deny*, *drop*, *allow*, *proxy* e *redirect*)).

Sintaxe: SecRuleEngine parâmetro

Exemplo: SecRuleEngine DetectionOnly

SecRulePerfTime

---

[24] http://httpd.apache.org/docs/2.0/sections.html

Esta diretriz define um limite de desempenho para as regras. Regras que gastam muito tempo serão enviadas ao log na parte H no formato id=usec (micro segundos).

Sintaxe: SecRulePerfTime micro_segundos

Exemplo: SecRulePerfTime 1000

SecRuleRemoveById

Esta diretriz remove uma regra pelo ID. Por exemplo, quando não for necessário utilizar todas as regras de um arquivo para uma determinada configuração utiliza-se esta diretriz para removê-la da busca de anomalias. Quanto menos regras mais rápido é o processamento e utiliza menos memória, dessa forma melhorando a performance.

Sintaxe: SecRuleRemoveById ID ID ID “RANGE”

Exemplo: SecRuleRemoveById 9992 9999 “100000-100010”

SecRuleRemoveByMsg

Esta diretriz remove uma regra pela variável mensagem que aparece em cada regra. Por exemplo, quando não for necessário utilizar alguma regra e a mesma não possui ID, utiliza-se esta diretriz para removê-la da busca de anomalias.

Sintaxe: SecRuleRemoveByMsg “mensagem”

Exemplo: SecRuleRemoveByMsg “Request content type is not allowed by policy”

SecRuleRemoveByTag

Esta diretriz remove uma regra pela *tag.* Todas as regras possuem pelo menos uma *tag,* cada tipo de ataque é uma *tag* diferente, conforme segue abaixo as *tags* disponíveis no CRS:

- OWASP_CRS/AUTOMATION/MALICIOUS
- OWASP_CRS/AUTOMATION/MISC

- OWASP_CRS/AUTOMATION/SECURITY_SCANNER
- OWASP_CRS/LEAKAGE/SOURCE_CODE_ASP_JSP
- OWASP_CRS/LEAKAGE/SOURCE_CODE_CF
- OWASP_CRS/LEAKAGE/SOURCE_CODE_PHP
- OWASP_CRS/WEB_ATTACK/CF_INJECTION
- OWASP_CRS/WEB_ATTACK/COMMAND_INJECTION
- OWASP_CRS/WEB_ATTACK/FILE_INJECTION
- OWASP_CRS/WEB_ATTACK/HTTP_RESPONSE_SPLITTING
- OWASP_CRS/WEB_ATTACK/LDAP_INJECTION
- OWASP_CRS/WEB_ATTACK/PHP_INJECTION
- OWASP_CRS/WEB_ATTACK/REQUEST_SMUGGLING
- OWASP_CRS/WEB_ATTACK/SESSION_FIXATION
- OWASP_CRS/WEB_ATTACK/SQL_INJECTION
- OWASP_CRS/WEB_ATTACK/SSI_INJECTION
- OWASP_CRS/WEB_ATTACK/XSS

Sintaxe: SecRuleRemoveByTab “tag”

Exemplo: SecRuleRemoveByTab “OWASP_CRS/WEB_ATTACK/LDAP_INJECTION”

SecRuleScript

Esta diretriz cria uma regra especial que executa um script Lua para decidir se coincide ou não. A principal diferença de SecRule é que não existem alvos nem operadores. O script pode buscar qualquer variável do contexto do modSecurity e usar qualquer operador para testá-los. O segundo parâmetro, que é opcional, é a lista de ações cujo significado é idêntico ao do SecRule.

Sintaxe: SecRuleScript “diretório_lua_script.lua” “[ações]”

Exemplo: SecRuleScript “/etc/lua/script.lua” “block”

SecRuleUpdateActionById

Esta diretriz atualiza a lista das ações de uma regra específica pelo ID. As únicas ações que não são possíveis de modificar são o ID e a fase da regra. Apenas as ações que podem aparecer uma vez são sobrescritas.

Sintaxe: SecRuleUpdateActionById ID_regra “ações”

Exemplo: SecRuleUpdateActionById 999745 "deny,status:403 "

SecRuleUpdateTargetById

Esta diretriz atualiza as variáveis de uma regra específica pelo ID. Será adionado ou substituído a variável da regra.

Sintaxe: SecRuleUpdateTargetById ID_regra “variável” “[trocar_pela_var_anterior]”

Exemplo: SecRuleUpdateTargetById 100018 "!ARGS:id"

SecRuleUpdateTargetByMsg

Esta diretriz atualiza as variáveis de uma regra específica pela mensagem.

Sintaxe: SecRuleUpdateTargetByMsg “mensagem” “variáveis” “[trocar_pela_var_anterior]”

Exemplo: SecRuleUpdateTargetByMsg "SQL Comment Sequence Detected." "!ARGS:cmd"

SecRuleUpdateTargetByTag

Esta diretriz atualiza as variáveis de uma regra específica pela tag.

Sintaxe: SecRuleUpdateTargetByTag “tag” “variáveis” “[trocar_pela_var_anterior]”

Exemplo: SecRuleUpdateTargetByTag “OWASP_CRS/WEB_ATTACK/SSI_INJECTION” “!ARGS:abc”

SecServerSignature

Esta diretriz altera os dados presentes no cabeçalho de resposta “Server”. É importante esta diretriz visto que se na resposta estiver enviando a versão do servidor web, pode ser um problema devido a possíveis vulnerabilidades.

Sintaxe: SecServerSignature “nome_servidor”

Exemplo: SecServerSignature “Apache”

SecStreamInBodyInspection

Esta diretriz configura a habilidade de usar o controle de fluxo para os dados de requisição de entrada em um *buffer* re-alocável. Por razões de segurança, o fluxo ainda é enviado para o *buffer*. Este recurso permite a criação da variável STREAM_INPUT_BODY, sendo útil na modificação de dados ou para combinar dados em dados não processados para qualquer tipo de conteúdo (*Content-Type*). Os parâmetros para esta diretriz são *On* e *Off*.

Sintaxe: SecStreamInBodyInspection parâmetro

Exemplo: SecStreamInBodyInspection On

SecStreamOutBodyInspection

Esta diretriz configura a habilidade de usar o controle de fluxo para os dados de requisição de saída em um *buffer* re-alocável. Por razões de segurança, o fluxo ainda é enviado para o *buffer*. Este recurso permite a criação da variável STREAM_OUTPUT_BODY, sendo útil na modificação de dados ou para combinar dados em dados não processados para qualquer tipo de conteúdo (Content-Type). Os parâmetros para esta diretriz são *On* e *Off*.

Sintaxe: SecStreamOutBodyInspection parâmetro

Exemplo: SecStreamOutBodyInspection On

SecTmpDir

Esta diretriz configure aonde os arquivos temporários serão criados.

Sintaxe: SecTmpDir diretório

Exemplo: SecTmpDir /tmp

SecUnicodeMapFile

Esta diretriz define o caminho para o arquivo que será usado pela função de transformação urlDecodeUni para mapear pontos de código Unicode durante a normalização.

Sintaxe: SecUnicodeMapFile caminho_até_unicode.mapping

Exemplo: SecUnicodeMapFile /etc/apache/crs/unicode.mapping

SecUnicodeCodePage

Esta diretriz define que ponto de código Unicode será usado pela função de transformação urlDecodeUni durante normalização.

Sintaxe: SecUnicodeCodePage código_unicode

Exemplo: SecUnicodeCodePage 20127

SecUploadDir

Esta diretriz configura o diretório onde os arquivos interceptados serão armazenados. Deve ser o mesmo caminho que foi definido pelo SecTmpDir, e sendo utilizado com SecUploadKeepFiles.

Sintaxe: SecUploadDir diretório

Exemplo: SecUploadDir /tmp

SecUploadFileLimit

Esta diretriz configura o número máximo de envio de arquivos em um POST em várias partes. O padrão é de 100 arquivos, qualquer arquivo acima do limite não será extraído e as *flags* de MULTIPART_FILE_LIMIT_EXCEEDED e MULTIPART_STRICT_ERROR serão definidos.

Sintaxe: SecUploadFileLimit número

Exemplo: SecUploadFileLimit 15

SecUploadFileMode

Esta diretriz configura a permissão dos arquivos enviados através do modo octal. O padrão é 0600, que o dono pode ler e escrever o arquivo.

Sintaxe: SecUploadFileMode número_octal ou “default”

Exemplo: SecUploadFileMode 0640

SecUploadKeepFiles

Esta diretriz configura se o arquivo ficará armazenado depois que a transação foi processada. É necessário setar a diretriz SecUploadDir. Os parâmetros possíveis são: *On* (mantém os arquivos), *Off* (não mantém os arquivos) e *RelevantOnly* (mantém apenas os arquivos que pertencem a requisições que são considerados relevantes).

Sintaxe: SecUploadKeepFiles parâmetro

Exemplo: SecUploadKeepFiles RelevantOnly

SecWebAppId

Esta diretriz cria um espaço de nomes para a aplicação, permitindo o armazenamento de sessão e usuário separadamente. É utilizado para evitar colisões entre IDs de sessão e IDs de usuário quando vários aplicativos são implantados no mesmo servidor. Se não for utilizada, uma colisão entre os IDs de sessão pode ocorrer. Um exemplo de configuração esta presente na Figura 26.

Sintaxe: SecWebAppId “nome”

Exemplo: SecWebAppId “Aplicacao01”

```
 1 <VirtualHost *:80>
 2 [..]
 3 SecWebAppId "aplicacao01"
 4 [..]
 5 </VirtualHost>
 6 <VirtualHost *:80>
 7 [..]
 8 SecWebAppId "aplicacao02"
 9 [..]
10 </VirtualHost>
```

Figura 26 - Exemplo da diretriz SecWebAppId

SecCollectionTimeout

Especifica o tempo limite de coletas. O padrão é 3600 segundos.

Sintaxe: SecCollectionTimeout tempo_em_segundos

Exemplo: SecCollectionTimeout 2400