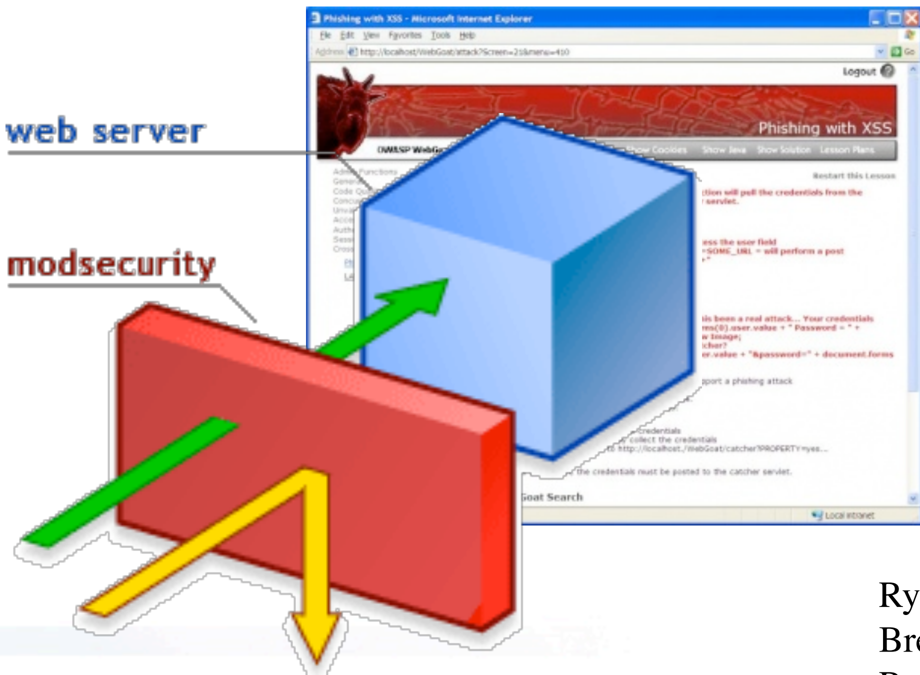
Virtual Patching Challenge

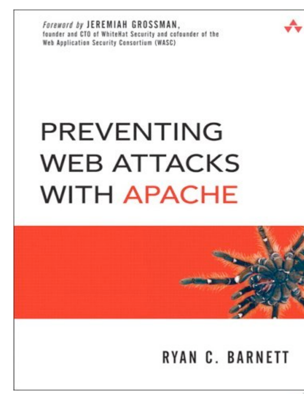
Securing WebGoat with ModSecurity



Ryan C. Barnett
Breach Security
Ryan.Barnett@breach.com

Speaker Background

- Director of Application Security Research at Breach Security
  - Lead Breach Security Labs
  - Develop ModSecurity Rules
- ModSecurity Community Manager
  - The go-between for development and the user community
- Background as an IDS/Web Security Admin
  - Operational web security for government clients
- Author
  - Preventing Web Attacks with Apache (Addison/Wesley, 2006)



Community Projects

- Open Web Application Security Project (OWASP)
  - Speaker/Instructor
  - Project Leader, ModSecurity Core Rule Set
- Web Application Security Consortium (WASC)
  - Board Member
  - Project Leader, Distributed Open Proxy Honeypots
- The SANS Institute
  - Courseware Developer/Instructor
- Center for Internet Security (CIS)
  - Apache Benchmark Project Leader

Agenda

- Virtual Patching Introduction
  - What is it?
  - Source Code/Patching Challenges
  - Value
- OWASP SoC Project
  - Securing WebGoat with ModSecurity
- Project Solution Examples
  - Cross-Site Scripting
    - Negative Security
    - Positive Security
    - AppDefect Identification
    - HTTPOnly Cookies
  - Cross-Site Request Forgery
    - Unique Token Implementation via Content-Injection
  - Session Management Flaws
    - Session Hijacking/Fixation
    - Deny Invalid Sessions
  - Hidden Parameter Tampering
    - The Need for Lua
- Conclusion/Questions

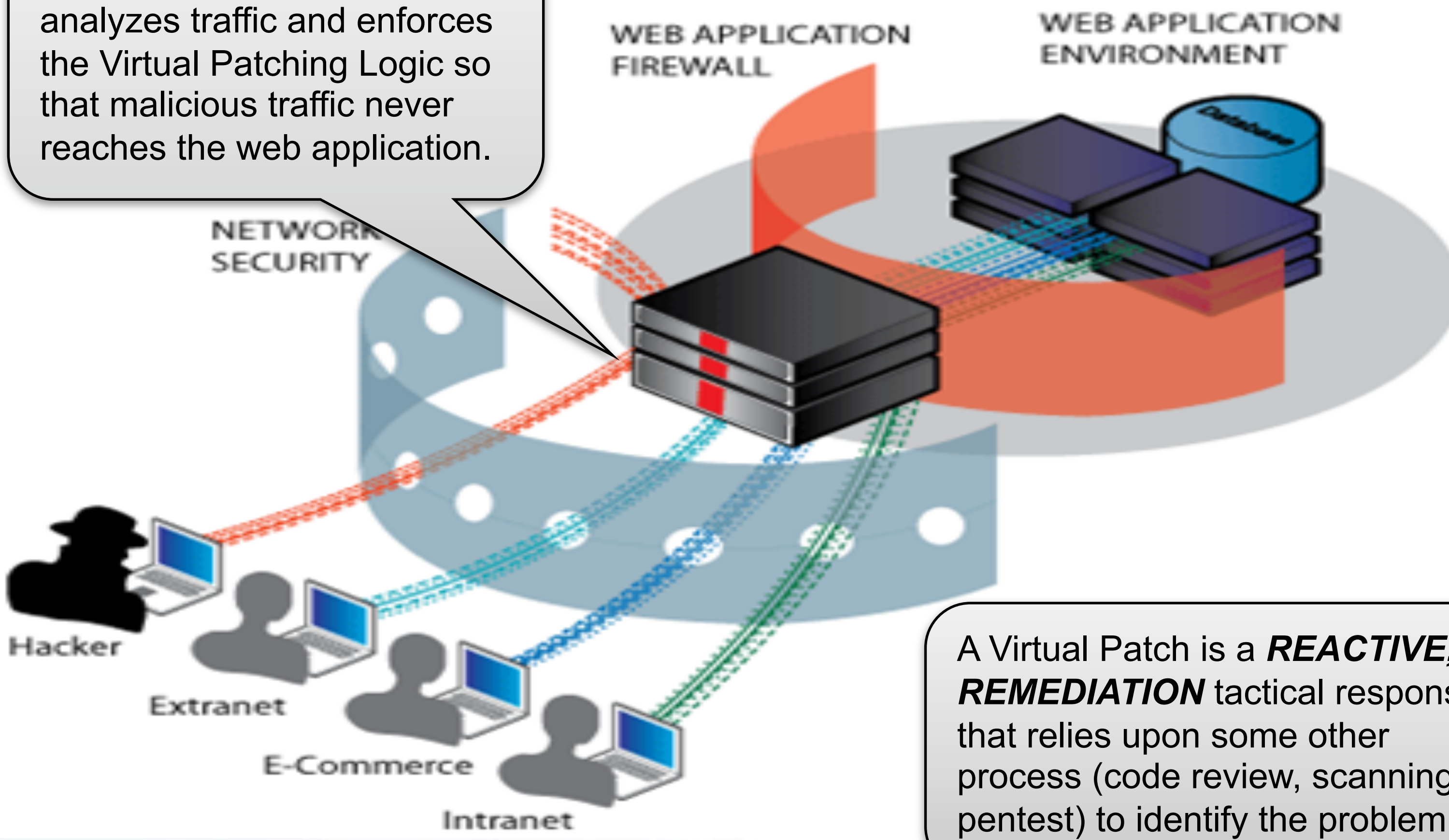
VIRTUAL PATCHING:

WHAT IS IT?

Black Hat Briefings

What is Virtual Patching?

A Web Application Firewall analyzes traffic and enforces the Virtual Patching Logic so that malicious traffic never reaches the web application.



A Virtual Patch is a ***REACTIVE, REMEDIATION*** tactical response that relies upon some other process (code review, scanning or pentest) to identify the problem.

Why Not Just Fix the Code?

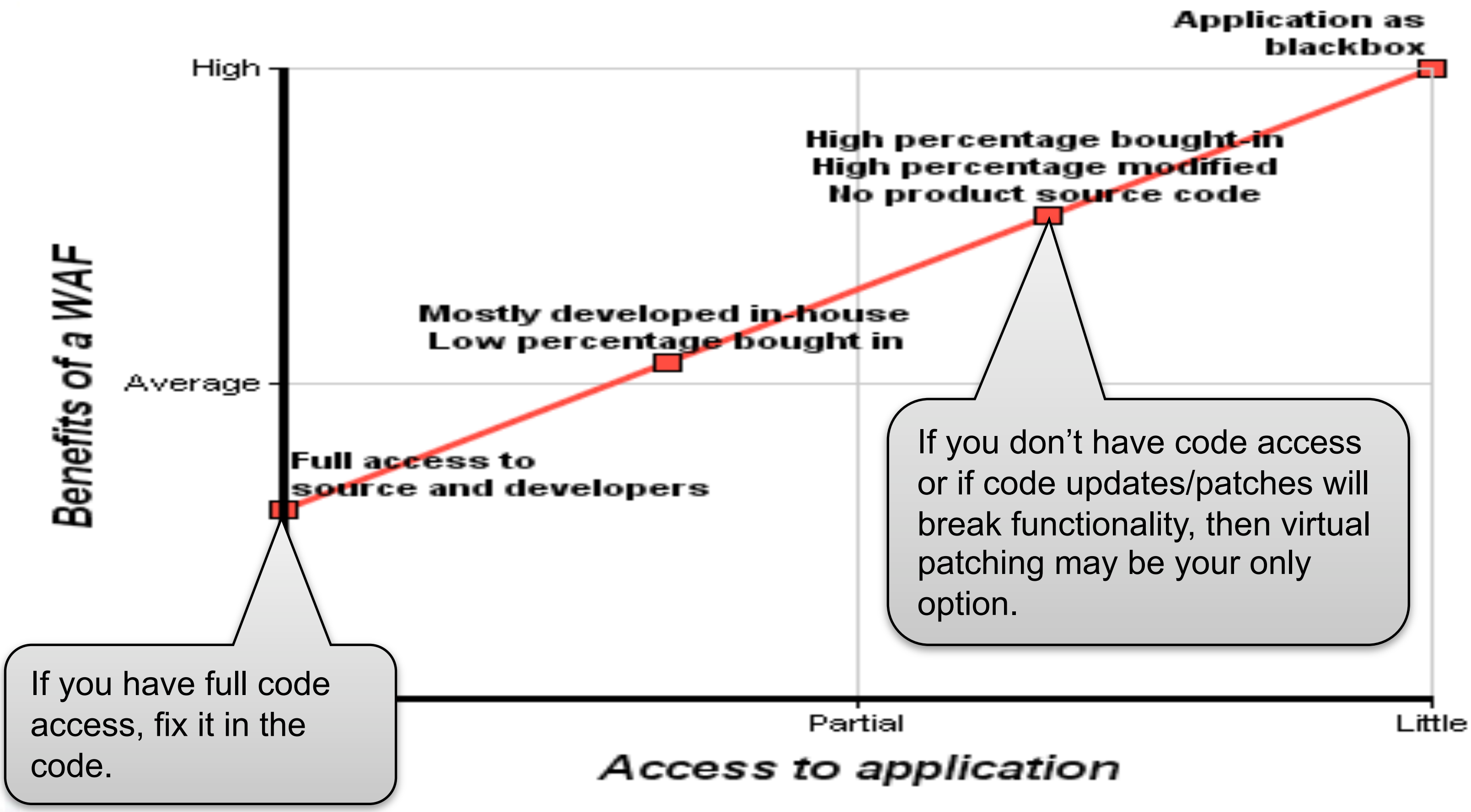
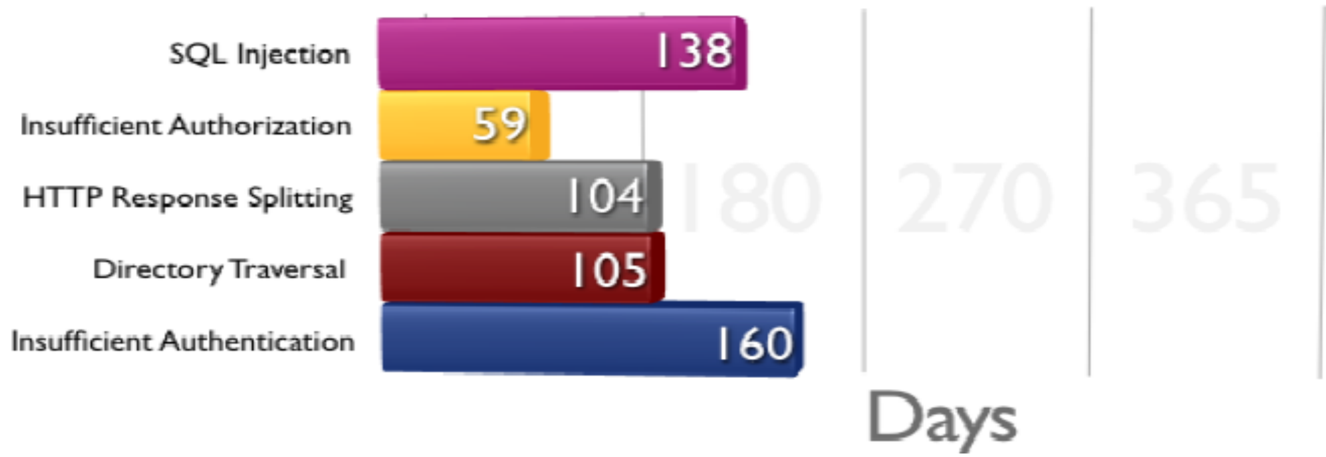


Image – OWASP Best Practices: Use of Web Application Firewalls

Vulnerability Scanning Statistics

Time-to-Fix

- Average # of days for the top 5 URGENT severity vulnerabilities to be fixed[1]



- Identification of the vulnerability was not the problem
- *Exploit Code Availability Average – 6 days[2]*
- Traditional code fixes take too long…

1 – Whitehat Website Security Statistics Report, March 2008
2 – Symantec Internet Security Threat Report, H3, 2007

Why Not Just Fix the Code?

Business Considerations

Risk Analysis =

Threat ×

Vulnerability ×

Impact ×

Cost to Fix

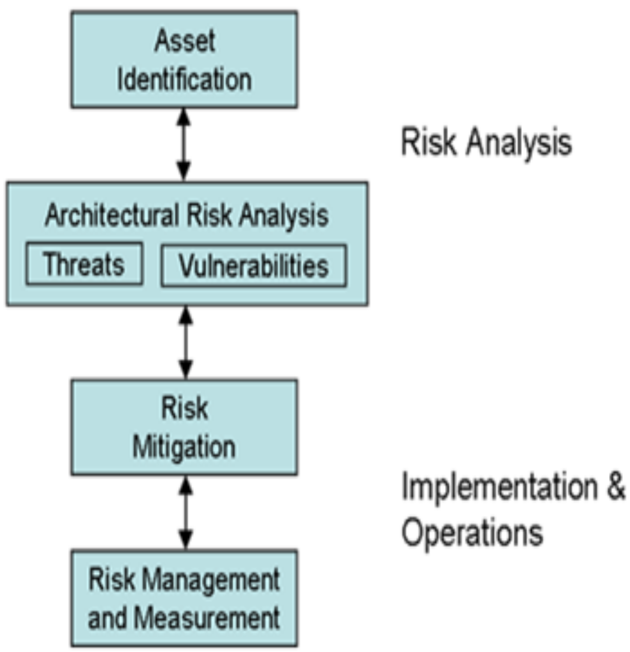
Why Not Just Fix the Code?

Emergent Behavior Phenomenon

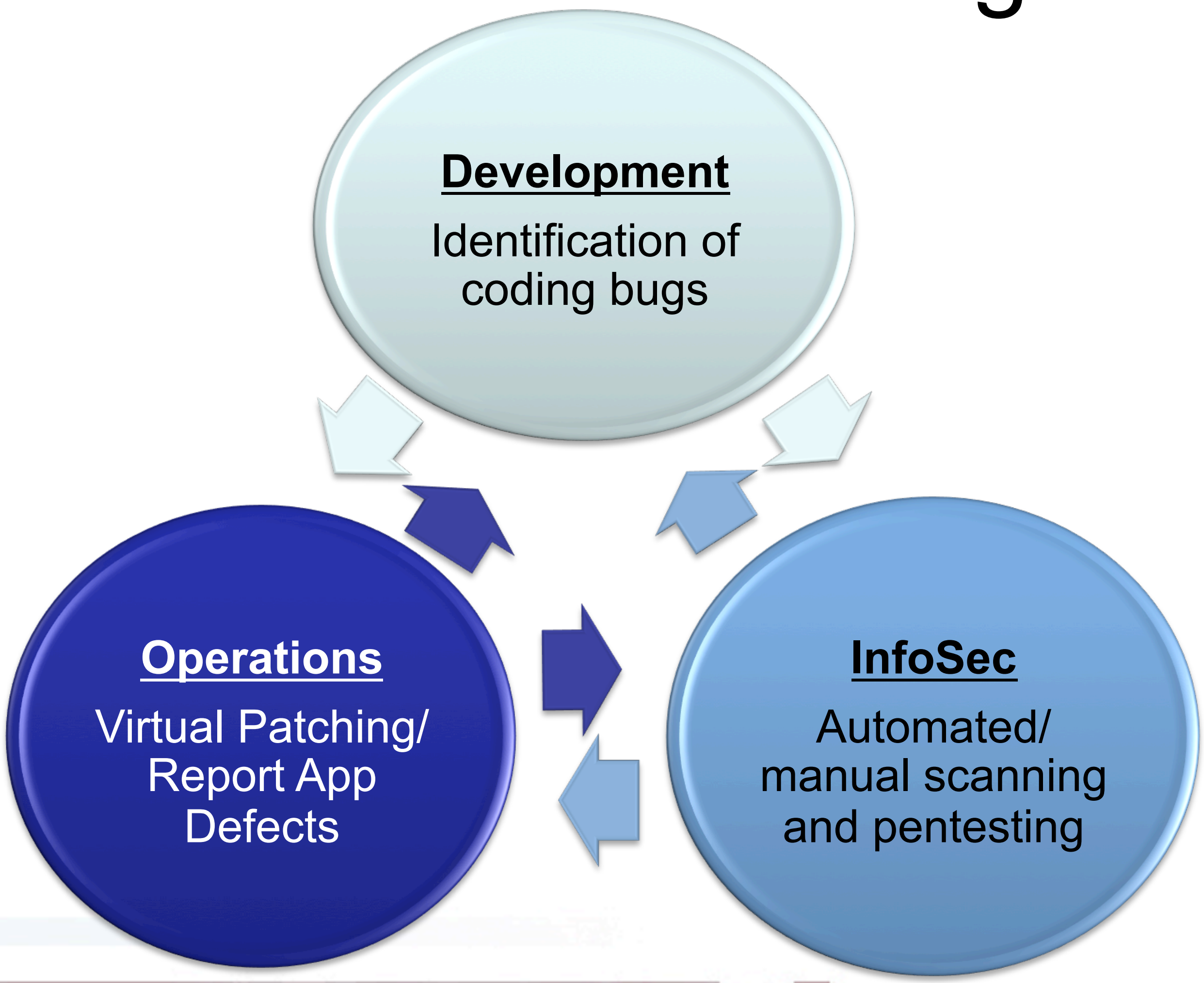
- Some vulnerabilities only manifest themselves *in production* when inter-connected systems share data.
- These are *Architectural Flaws* that exhibit Emergent Behaviors:

  Two pieces of code put together, one with a limited spec for strong data typing, and the other with weak handling of output, result in a new set of ***behaviors*** *that fail to meet specification, though each unit of code individually meets it's own specification.*

  – Arian Evans (WebSecurity Mail-list Post)
- May not be identifiable or correctable within the application's code.



Code Reviews + Scanning + WAF



Organizational Value

Business Value	• Allows organizations to maintain normal patching cycles. • Reduced or eliminated time and money spent performing emergency patching.
Risk Value	• Reduces risk until a vendor-supplied patch is released or while a patch is being tested and applied. • Protection for mission-critical systems that may not be taken offline.
Technical Value	• Less likelihood of introducing conflicts as libraries and support code files are not changed. • Scalable solution as it is implemented in few locations vs. installing patches on all hosts.

Security Consultant Value

- When it is not possible to edit the application's code (for either business or technical reasons), security consultants are limited in the services they can provide.
  - Virtual Patching offers additional options.
- How web "multi-lingual" are you?
  - PHP, ASP/ASP.NET, Java, Python, Ruby, VB.NET, C#...
  - Actual Security Consultant Quote -

    *For the purpose of patching painfully old systems, that should really have been taken out and shot but are kept running for 'business-reasons', **I'd rather learn ModSecurity + e.g Lua properly than having to learn every thinkable and unthinkable language and platform ever used for throwing together web content.***

OWASP SUMMER OF CODE PROJECT:

SECURING WEBGOAT WITH MODSECURITY

Project Team and Objective

- **Project Team**
  - **Leader: Stephen Evans (Security Consultant)**
  - **1st reviewer: Ivan Ristic & Ryan Barnett (Breach Security Labs)**
  - **2nd reviewer: Christian Folini (ModSecurity Power User)**

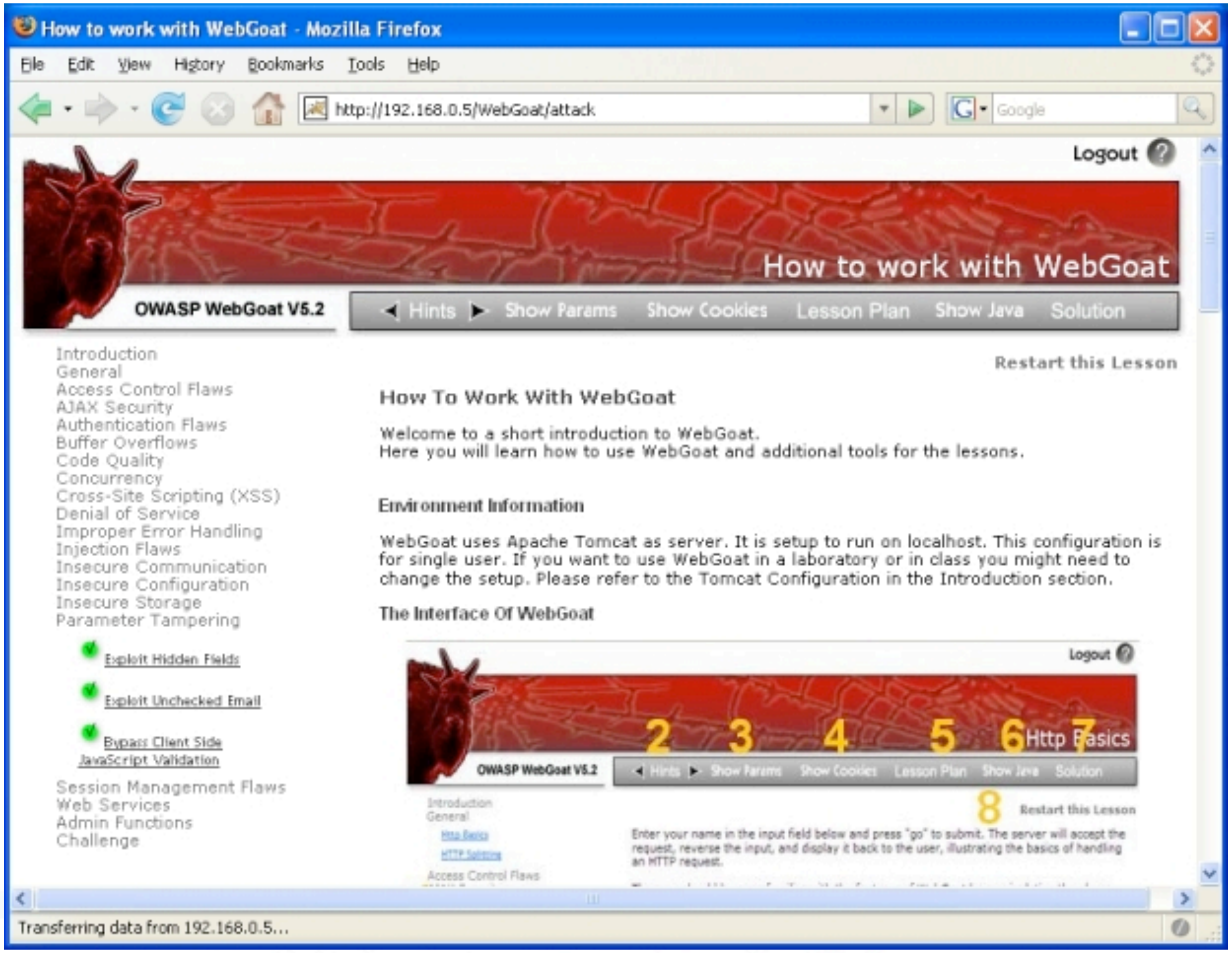
- **Objective**

- *"To create custom ModSecurity rulesets that, in addition to the Core Set, will protect WebGoat 5.2 Standard Release from as many of its vulnerabilities as possible (the goal is 90%)* ***without changing one line of source code."***

Project Goals

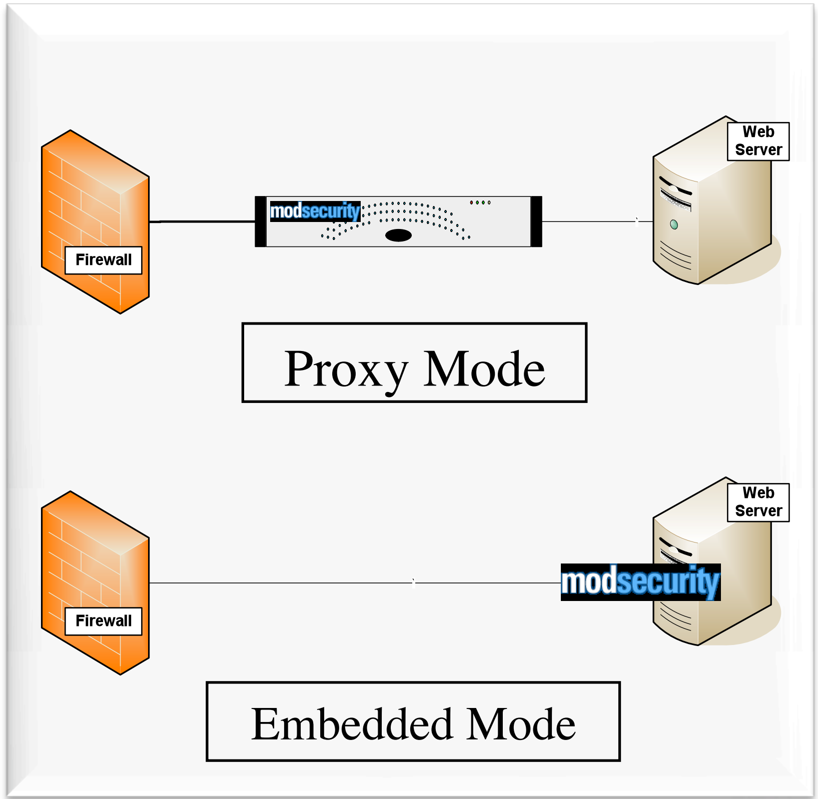
- Demonstrate cutting-edge WAF capabilities
  - **Wow, I didn't know (a WAF|ModSecurity) could do that?!**
- Tactical use-cases for virtual patching vulnerability remediation
  - Anyone can download WebGoat and ModSecurity and run their own tests.
- Virtual Patching Options
  - Block attacks to exploit the vulnerability.
  - Address the specific, underlying WebGoat vulnerability.
  - If possible, address the underlying vulnerability generically so that the virtual patch could be applied to other applications that suffer from the same issue.
  - Alert on identified Application Defects.

WebGoat Overview



Why ModSecurity?

- Open Source, Free - ☺
- Can be deployed embedded or on a reverse proxy
- Deep understanding of HTTP and HTML
- Robust Parsing
- Anti Evasion Features
- Supports Complex Rules Logic
- Advanced Capabilities
  - Persistent Collections
  - Content Injection and Lua API



www.modsecurity.org

ModSecurity Rules Language

It's a simple event-based programming language.

Five processing phases, one for each major processing step.	Look at any part of the transaction.	Transform data to counter evasion.	Combine rules to form complex logic.

Common tasks are easy (the Core Rule Set), complex tasks are possible (Virtual Patching).

Example Rule Syntax

Tells ModSecurity ***how to process data***

```
SecRule TARGETS OPERATOR [ACTIONS]
```

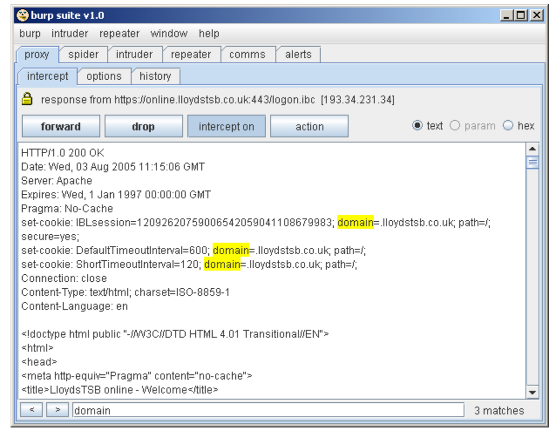
Tells ModSecurity ***where to look***

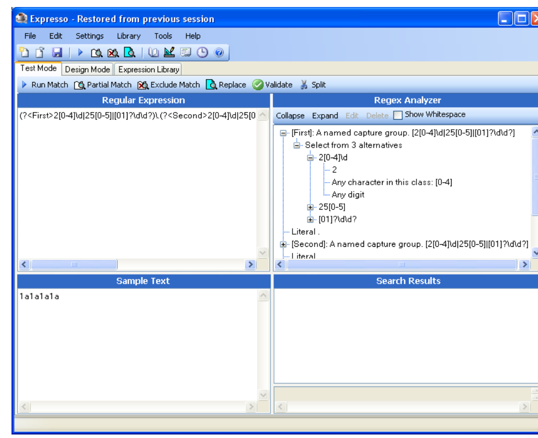
Tells ModSecurity ***what to do*** if a rule matches

Tools

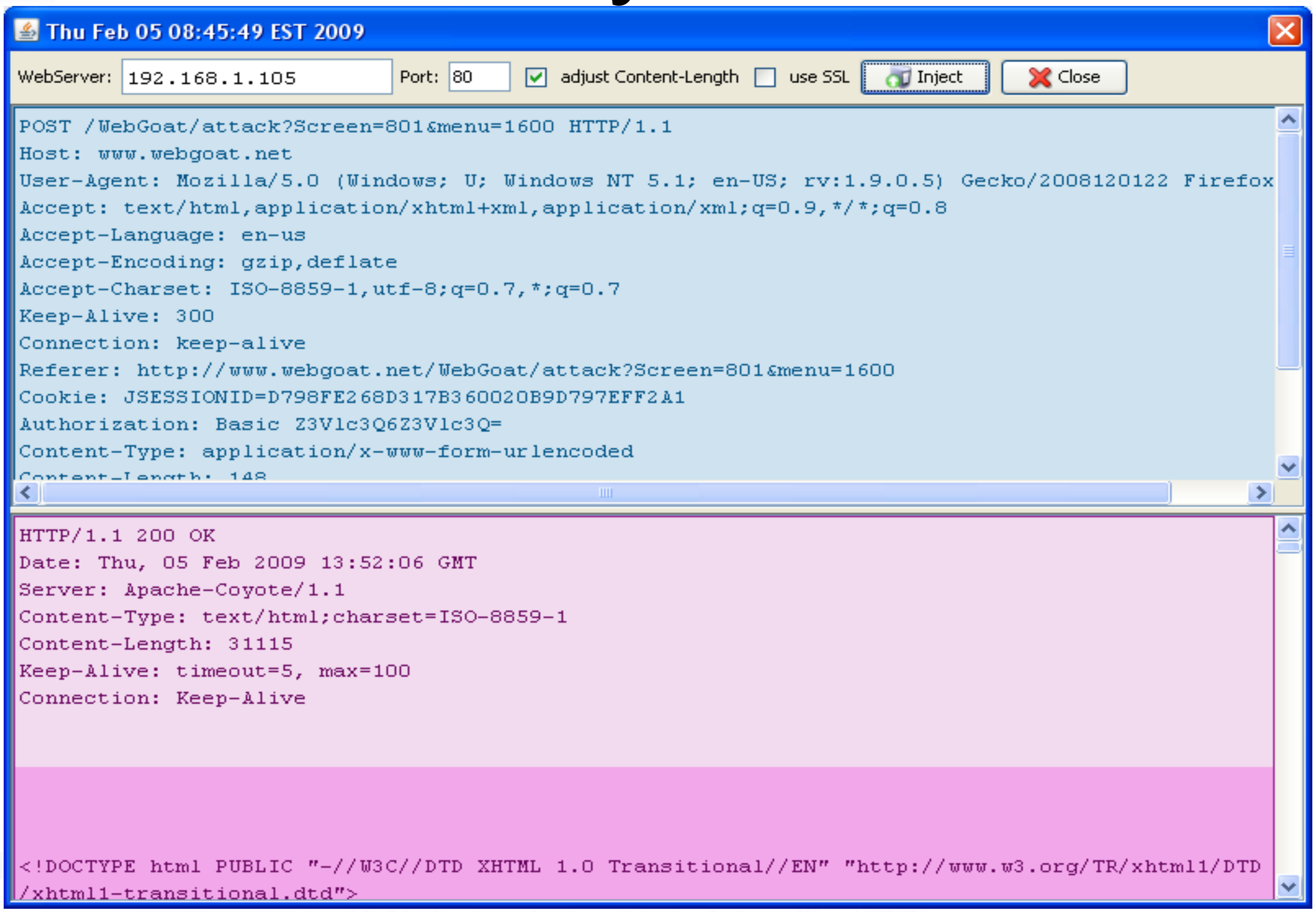
Testing Tools

- In order to accurately test out the virtual patch, it may be necessary to use other tools –
  - cURL - Command line web client
  - Burp Proxy – Local Proxy
  - Expresso – RegEx GUI Tool
- These tools will aid in both the construction and testing of virtual patching rules.





ModSecurity AuditViewer



SOLUTION EXAMPLE:
CROSS-SITE SCRIPTING (XSS)

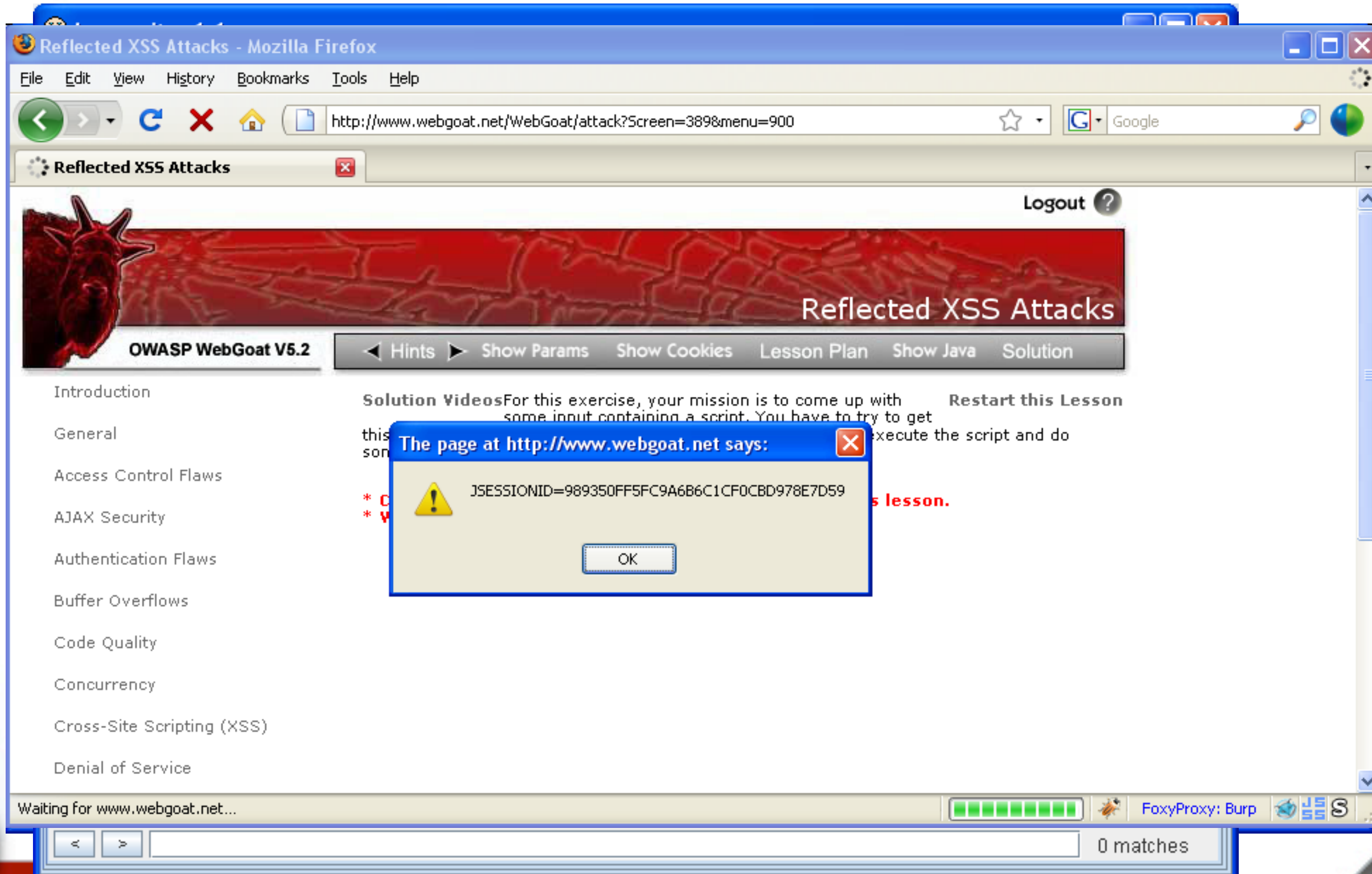
Cross-Site Scripting (XSS)

- Application Defect(s)
  - Insufficient input validation
  - Application does not properly output encode user supplied data (meta-characters)
- Vulnerability:
  - Attacker can send JavaScript to the web application and have the code execute within the victim's browser
- Technique:
  - If attackers are able to insert XSS code, they may be able to steal SessionID credentials or do other harm
- Consequence:
  - Session Hijacking, Malware Installs, Fraud (CSRF)

Cross-Site Scripting (XSS)

- **Reflected XSS**
- Attacker tricks the victim into sending the malicious payload themselves (e.g. Phishing email).
- Malicious JavaScript is sent/echoed back ***in the same transaction***

Reflected XSS Lesson



Reflected XSS – Mitigations

- Input Validation
  - Negative Security – Blacklist known XSS using the Core Rule Set Regular Expressions
  - Positive Security – Enforce expected input for “field1” parameter data
- Application Defect Identification/ Remediation
  - Identify if application does not properly output encode user supplied data
  - HTTPOnly flag missing from SessionIDs

Reflected XSS Mitigations

Input Validation

Negative security model: allow all, deny what's wrong

- Blacklist known XSS payloads using the ModSecurity Core Rule Set Regular Expressions

Positive security model: deny all, allow what's right

- Enforce expected input for “field1” parameter data

Core Rules - Negative Security

Generic XSS Detection

```
SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES "(?:\b(?:
(?:type\b\W*?\b(?:text\b\W*?\b(?:j(?:ava)?|ecma|vb)|
application\b\W*?\bx-(?:java|vb))script|
c(?:opyparentfolder|reatetextrange)|get(?:special|
parent)folder|iframe\b.{0,100}?\bsrc)\b|on(?:
(?:mo(?:use(?:o(?:ver|ut)|down|move|up)|ve)|
key(?:press|down|up)|c(?:hange|lick)|s(?:elec|
ubmi)t|(?:un)?load|dragdrop|resize|focus|blur)\b\W*?
=|abort\b)|(?:l(?:owsrc\b\W*?\b(?:(?:… \

"phase:
2,pass,capture,t:none,t:htmlEntityDecode,t:compressW
hiteSpace,t:lowercase,ctl:auditLogParts=
+E,log,auditlog,msg:'Cross-site Scripting (XSS)
Attack',id:'950004',tag:'WEB_ATTACK/XSS',logdata:'%
{TX.0}',severity:'2'"
```

Updated Core Rules

Targeted XSS for WebGoat

```
<Location /WebGoat/attack>
SecRule ARGS:field1 "(?:\b(?:(?:type\b\W*?\b(?:text
\b\W*?\b(?:j(?:ava)?|ecma|vb)|application\b\W*?\bx-
(?:java|vb))script|c(?:opyparentfolder|
reatetextrange)|get(?:special|parent)folder|iframe
\b.{0,100}?\bsrc)\b|on(?:(?:mo(?:use(?:o(?:ver|ut)|
down|move|up)|ve)|key(?:press|down|up)|c(?:hange|
lick)|s(?:elec|ubmi)t|(?:un)?load|dragdrop|resize|
focus|blur)\b\W*?=|abort\b)|(?:l(?:owsrc\b\W*?\b(?:
(?:… \
"phase:
2,deny,capture,t:none,t:htmlEntityDecode,t:compressW
hiteSpace,t:lowercase,ctl:auditLogParts=
+E,log,auditlog,msg:'Cross-site Scripting (XSS)
Attack',id:'1',tag:'WEB_ATTACK/XSS',logdata:'%{TX.
0}',severity:'2'"
</Location>
```

Why Negative Security Fails

Evasions

- Canonicalization/Obfuscation Problems
  - Unicode, HTML Encoding/Decoding
  - Too many variations that result in ***functionality equivalent code***
- Original form

```
<script>alert('XSS')</script>
```

- Using ActionScript inside Flash

```
a="get"; b="URL(\""; c="javascript:";
  d="alert('XSS');\")"; eval(a+b+c+d);
```

- DIV Background Image

```
<DIV STYLE="background-image:
  \0075\0072\006C\0028'\006a
  \0061\0076\0061\0073\0063\0072\0069\0070
  \0074\003a\0061\006c
  \0065\0072\0074\0028.1027\0058.1053\0053
  \0027\0029'\0029">
```

http://ha.ckers.org/xss.html

Reflected XSS

Positive Security Model

```
<Location /WebGoat/attack>
SecRule &ARGS_GET_NAMES:field1 “@ge 1” “phase:
2,t:none,log,deny,msg:’Field1 Parameter Found in
Query_String.’”
SecRule &ARGS_POST_NAMES:field1 “@eq 0” “phase:
2,t:none,log,deny,msg:’Field1 Parameter is Missing
from Post Payload.’”
SecRule &ARGS_POST_NAMES:field1 “@gt 1” “phase:
2,t:none,log,deny,msg:’Multiple Field1 Parameters
Found in Post Payload.’”
SeRule ARGS_POST:field1 “!^\d{3}$” “phase:
2,t:none,log,deny,msg:’Field1 Invalid Parameter
Data.”
</Location>
```

Only allow
3 digits

Reflected XSS

Application Defect

Failure to HTML Output Encode User Supplied Data

Request

- <IMG SRC="javascript:alert('XSS');">
- <IMG SRC="javascript:alert('XSS');">

Response

Correct HTML Output Encoding of User Supplied Data

Request

- <IMG SRC="javascript:alert('XSS');">
- <IMG SRC="javascript:alert('XSS');">

Response

Reflected XSS

ModSecurity Audit Log Entry

```
--1e58114a-A--
[05/Feb/2009:01:42:16 --0500] SYqKSH8AAQEAABUEAsgAAAAA
192.168.110.1 4134 192.168.110.133 80
--1e58114a-B--
POST /WebGoat/attack?Screen=49&menu=900 HTTP/1.1
--CUT--
--1e58114a-C--
QTY1=1&QTY2=1&QTY3=1&QTY4=1&field2=4128+3214+0002+1999&fiel
d1=<script>alert(document.cookie)</script>&SUBMIT=Purchase
--CUT--
--1e58114a-E--
--CUT--
<div id="message" class="info"><BR> * Congratulations. You
have successfully completed this lesson.<BR> * Whoops! You
entered <script>alert(document.cookie)</script> instead of
your three digit code.  Please try again.</div>
```

App Defect Rule Set

Dynamic Taint Propagation[1]

Follow untrusted data and identify points where they are misused

Use ModSecurity's built-in Transactional Collection (TX)

Use the "setvar:tx" action

Inspect Request Parameter Payloads

Monitor inbound payloads for meta-characters that could be used in an XSS attack	Set a TX variable that holds this data

Inspect *Current* Response Body Payload

Check outbound response data for **the exact same user-supplied data**

1- Fortify - B. Chess/J. West

Reflected XSS

App Defect Rule

```
SecRule ARGS "([\'\"\(\)\;<>#])" "chain,phase:
4,t:none,log,auditlog,deny,status:
403,id:'1',msg:'Potentially Malicious Meta-
Characters in User Data Not Properly Output
Encoded.',logdata:'%{tx.inbound_meta-
characters}'"
      SecRule MATCHED_VAR "^.{15,}$"
"chain,t:none,setvar:tx.inbound_meta-
characters=%{matched_var}"
            SecRule RESPONSE_BODY
"@contains %{tx.inbound_meta-characters}"
"ctl:auditLogParts=+E"
```

Rule and Debug Log Data (1)

```
SecRule ARGS "([\'\"\(\)\;<>#])" "chain,phase:
4,t:none,log,auditlog,deny,status:
403,id:'1',msg:'Potentially Malicious Meta-
Characters in User Data Not Properly Output
Encoded.',logdata:'%{tx.inbound_meta-characters}'"
```

```
Rule 81c0640: SecRule "ARGS" "@rx ([\\'\"\\(\\)\
  \;<>#])" "phase:
  2,log,auditlog,pass,chain,t:none,setvar:tx.inbound_met
  a-characters=%{matched_var}"
[4] Executing operator "rx" with param "([\\'\"\\(\\)\
  \;<>#])" against ARGS:field1.
[9] Target value: "<script>alert(document.cookie)</
  script>"
[9] Added regex subexpression to TX.0: <
[9] Added regex subexpression to TX.1: <
[4] Operator completed in 28 usec.
```

Rule and Debug Log Data (2)

```
SecRule MATCHED_VAR "^.{15,}$"
"chain,t:none,setvar:tx.inbound_meta-characters=%
{matched_var}"
```

```
[5] Rule 81d6a30: SecRule "MATCHED_VAR" "@rx ^.{15,}$" "phase:
  2,log,auditlog,pass,chain,t:none,setvar:tx.inbound_meta-
  characters=%{matched_var}"
[4] Transformation completed in 2 usec.
[4] Executing operator "rx" with param "^.{15,}$" against
  MATCHED_VAR.
[9] Target value: "<script>alert(document.cookie)</script>"
[4] Operator completed in 3 usec.
[9] Setting variable: tx.inbound_meta-characters=%{matched_var}
[9] Resolved macro %{matched_var} to
  "<script>alert(document.cookie)</script>"
[9] Set variable "tx.inbound_meta-characters" to
  "<script>alert(document.cookie)</script>".
[4] Rule returned 1.
```

Rule and Debug Log Data (3)

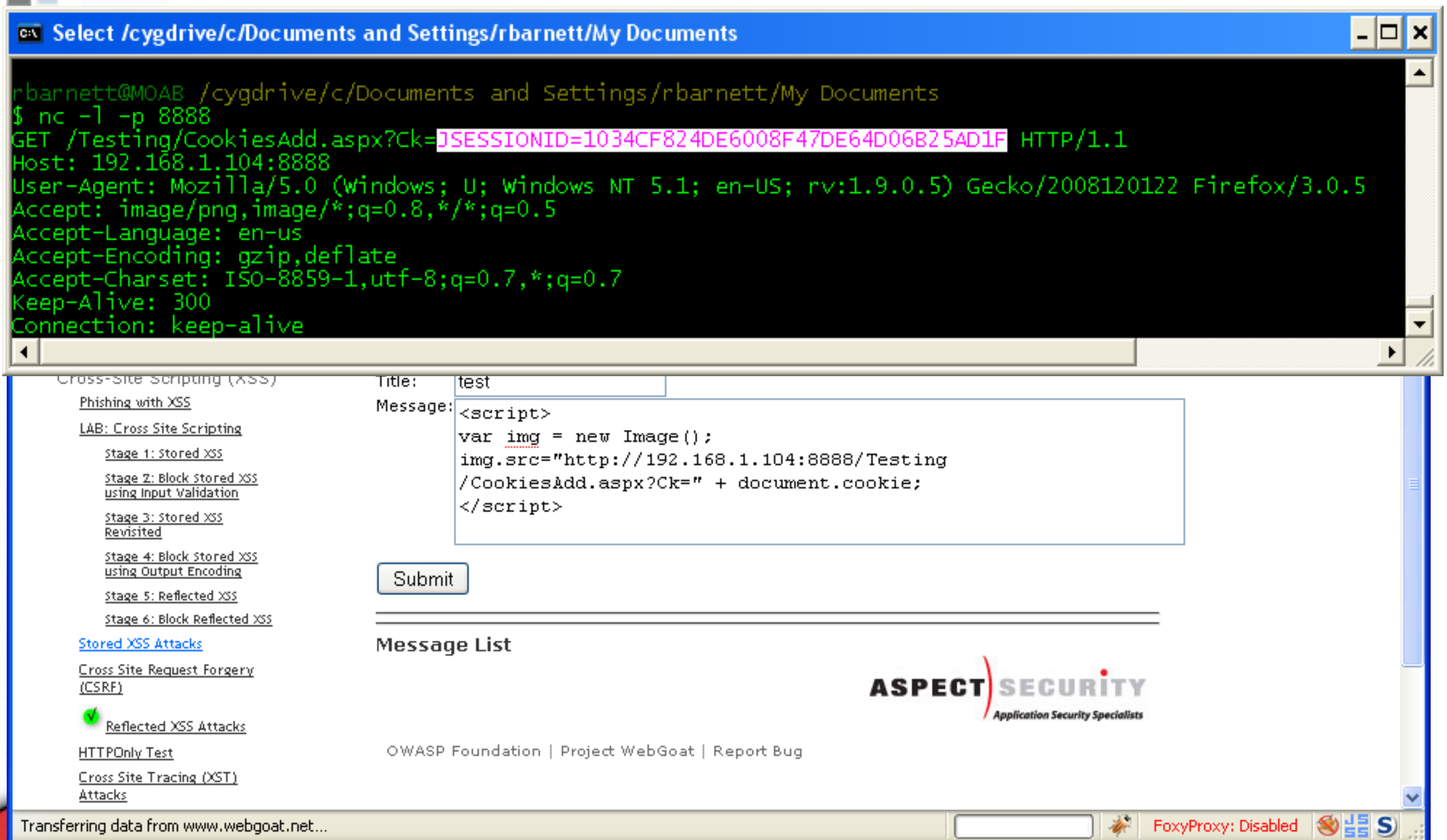
```
SecRule RESPONSE_BODY "@contains %
{tx.inbound_meta-characters}" "ctl:auditLogParts=
+E"
```

```
[4] Executing operator "contains" with param "%
  {tx.inbound_meta-characters}" against RESPONSE_BODY.
[9] Target value: "\r\n\r\n\r\n<!DOCTYPE html PUBLIC "-//W3C//
  DTD XHTML 1.0 Transitional//EN”
--CUT--
[9] Resolved macro %{tx.inbound_meta-characters} to
  "<script>alert(document.cookie)</script>‟
[4] Ctl: Set auditLogParts to ABIFHZE.
[2] Warning. String match "<script>alert(document.cookie)</
  script>" at RESPONSE_BODY. [file "/usr/local/apache/conf/
  core-rules_1.6.2/modsecurity_crs_15_customrules.conf"] [line
  "17"] [id "1"] [msg "Potentially Malicious Meta-Characters in
  User Data Not Properly Output Encoded."] [data
  "<script>alert(document.cookie)</script>"]
```

Cross-Site Scripting (XSS)

- **Stored XSS**
- Attacker is the one who sends the malicious payload to the application.
- Victim views the malicious payload at another time.
- Malicious JavaScript is sent/echoed back ***in different transactions***
- Negative/Positive Security rules presented for Reflective XSS still work to block the inbound attack

Stored XSS Lesson – Stealing



App Defect Rule Set

Stored XSS

Use ModSecurity's Global Persistent Collection (GLOBAL)

Use the "initcol:global=" and "setvar:global." actions

Leverage the Reflected XSS Rules

Set a variable in the GLOBAL collection that holds this data across transactions

Inspect *ALL* Response Body Payloads

Check outbound response data for **the exact same user-supplied data**

Stored XSS

App Defect Rule

```
SecAction "phase:1,nolog,pass,initcol:global=xss_list"
…Reflected XSS Rules Here…
SecRule GLOBAL:'/XSS_LIST_.*/' "@streq "phas%
{tx.inbound_meta-characters}" e:
4,t:none,nolog,pass,skip:1"
SecRule TX:INBOUND_META-CHARACTERS ".*" "phase:
4,t:none,nolog,pass,setvar:global.xss_list_%
{time_epoch}=%{matched_var}"

SecRule GLOBAL:'/XSS_LIST_.*/' "@within %
{response_body}" "phase:
4,t:none,log,auditlog,pass,msg:'Potentially Malicious
Meta-Characters in User Data Not Properly Output
Encoded',tag:'WEB_ATTACK/XSS'"
```

Viewing Persistent Collections

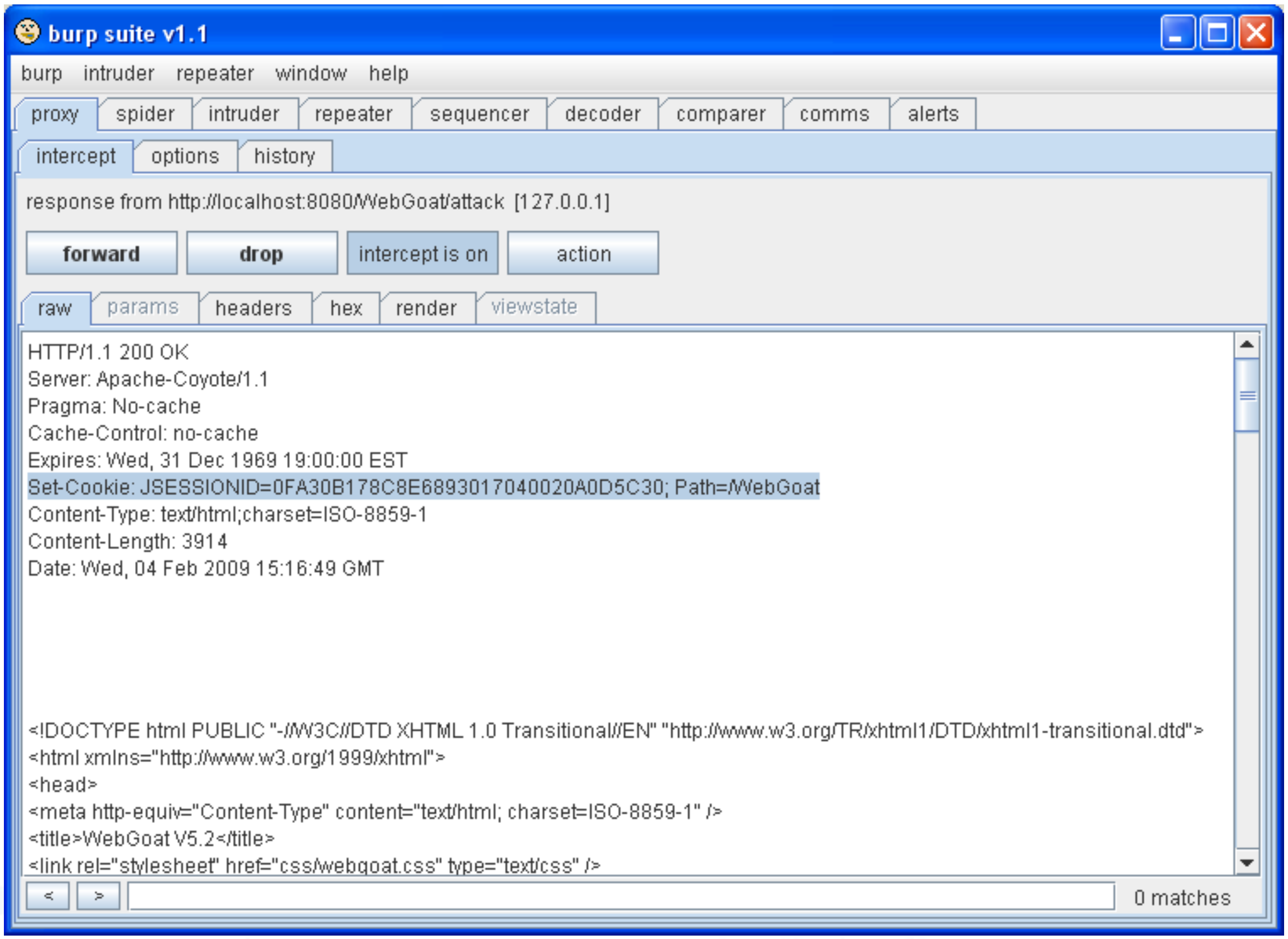
```
# java -cp /root/org.jwall.tools.jar
org.jwall.tools.CollectionViewer /tmp/

Collection global, last read @ Thu Feb 05 02:01:18 EST 2009

Created at Thu Feb 05 01:42:16 EST 2009

global[xss_list].xss_list_1233816136 =
<script>alert(document.cookie)</script>

global[xss_list].xss_list_1233817131 =
<SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>

global[xss_list].xss_list_1233817276 = <META HTTP-
EQUIV="refresh" CONTENT="0;
URL=http://;URL=javascript:alert('XSS');">

global[xss_list].xss_list_1233817198 = <BASE
HREF="javascript:alert('XSS');//">

global[xss_list].TIMEOUT = 3600

This collection expires in 59m 57.242s
```

Application Defect Mitigation

Missing Cookie Protections – HTTPOnly Flag

- Defect:
  - Application does not use the HttpOnly Cookie Option
- Vulnerability:
  - The HttpOnly cooking flag option helps to prevent client-side code from access the cookie data within the browser
- Technique:
  - If attackers are able to insert XSS code, they may be able to steal SessionID credentials
- Consequence:
  - Session Hijacking

Missing HTTPOnly Flag



Missing HTTPOnly Rule Set

ModSecurity + Apache

Use ModSecurity to Inspect Outbound Set-Cookie Data

Check for SessionIDs that are missing the HTTPOnly flag

Use ModSecurity's "setenv" action

ENV data holds the initial Set-Cookie data

Apache can access/update this data using the Header directive

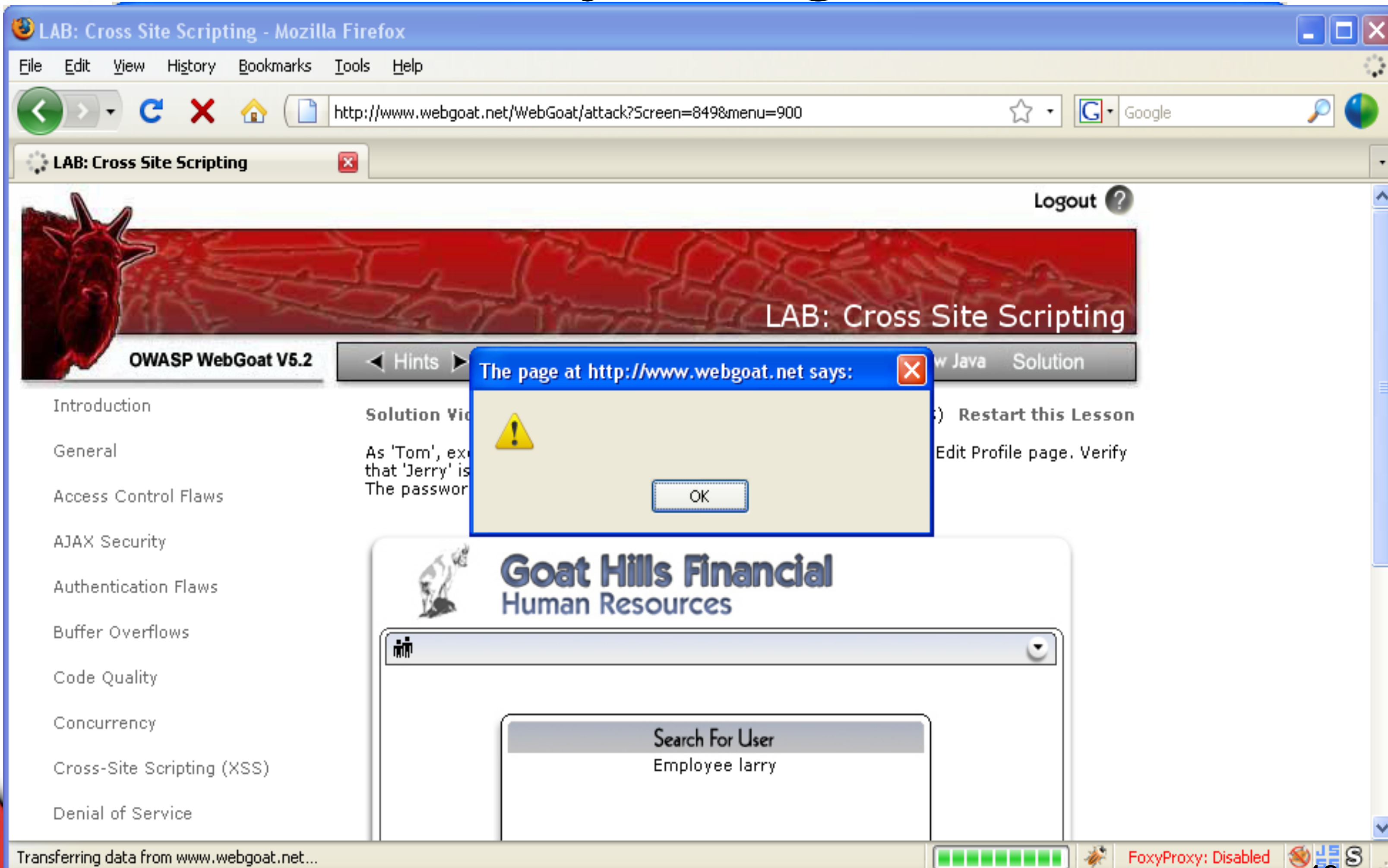
Issue a new Set-Cookie header that appends HTTPOnly to the end

Missing HTTPOnly Flag

App Defect Rule

```
SecRule RESPONSE_HEADERS:/Set-Cookie2?/ "!(?i:
\;? ?httponly;?)" "chain,phase:
3,t:none,pass,nolog"
  SecRule MATCHED_VAR "(?i:(j?sessionid|(php)?
sessid|(asp|jserv|jw)?session[-_]?(id)?|cf(id|
token)|sid))" "t:none,setenv:http_cookie=%
{matched_var}"


Header set Set-Cookie "%{http_cookie}e;
HTTPOnly" env=http_cookie
```

HTTPOnly Flag Added



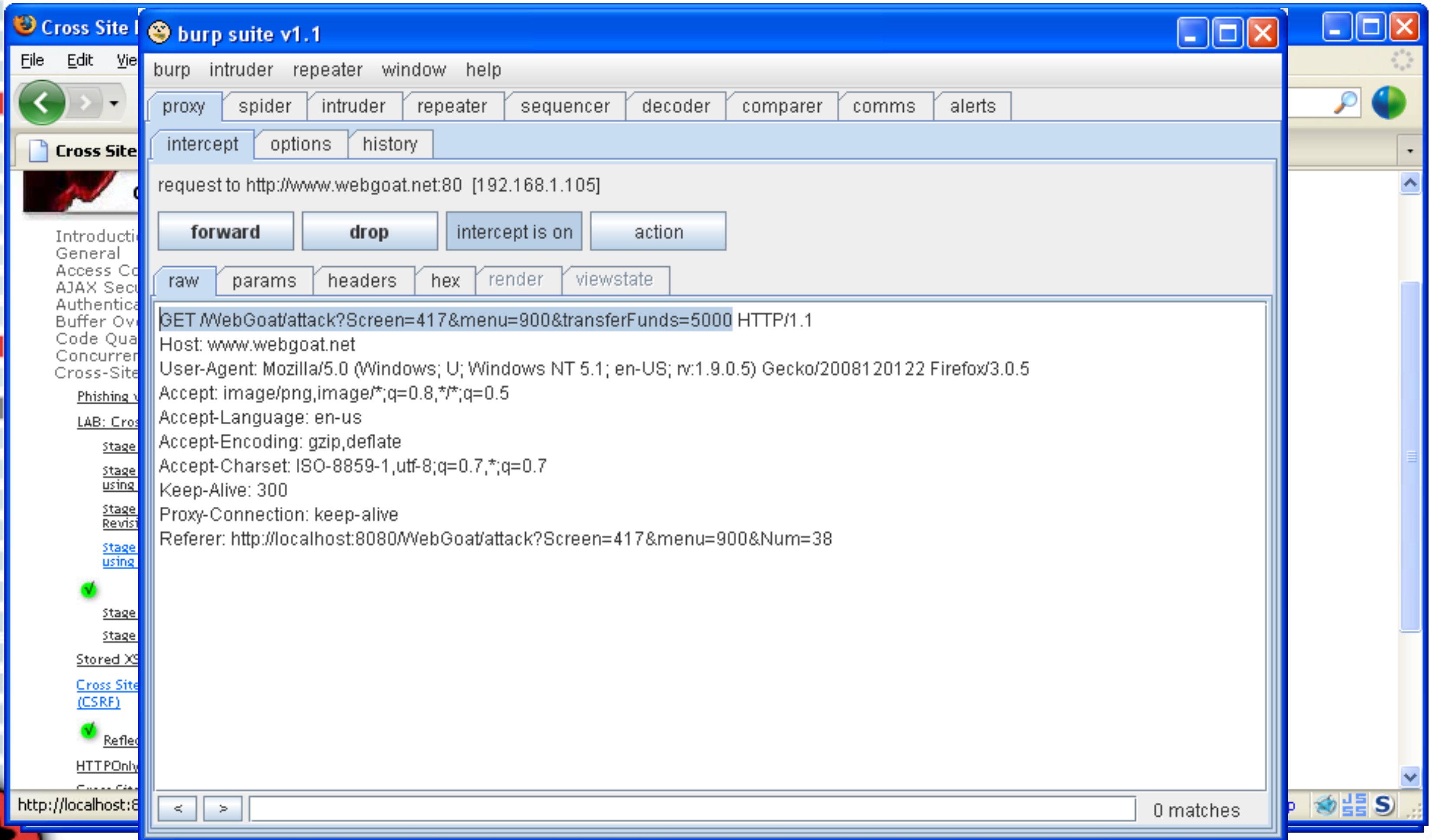
Black Hat Briefings

SOLUTION EXAMPLE:
CROSS-SITE REQUEST FORGERY (CSRF)

Cross Site Request Forgery (CSRF)

- Defect:
  - Application uses Implicit Authentication based on SessionID Cookie data
  - Also known as Session Riding, One-Click Attacks, etc…
- Vulnerability:
  - Web browsers automatically send SessionID data with requests
- Technique:
  - An attack that tricks the victim into loading a page that contains a malicious request.
  - In a forum, the attack may direct the user to invoke a logout function
  - Can be combined with XSS
- Consequence:
  - Fraud

WebGoat CSRF Lesson



CSRF Mitigation

Adding Unique Tokens via Content Injection

Use ModSecurity's Session Persistent Collection

Data is saved for each SessionID

Create/Inject Unique Token Value into Response Data

Use "t:sha1" action to capture hash of JSESISONID	Use "append" Content Injection action	Uses csrf.js script from OWASP CSRFGuard

Validate CSRF Token Data on Subsequent Requests

Check that CSRF Token exists and data matches the saved CSRF Hash data

CSRF Rules (1)

Storing/Injecting Unique Tokens

```
SecContentInjection On

SecRule RESPONSE_HEADERS:/Set-Cookie2?/ "(?i:jsessionid=([a-f0-9]+)\;\s?)" "chain,phase:3,t:none,pass,nolog,capture,setsid:%{TX.1},setvar:session.sessionid=%{TX.1}"
      SecRule SESSION:SESSIONID "(.*)" "t:none,capture,t:sha1,t:hexEncode,setvar:session.csrf_token=%{TX.1}"

SecRule REQUEST_FILENAME "/WebGoat/attack" "phase:4,t:none,nolog,pass,append:'<script language=\"JavaScript\"> \
var tokenName = \'MODSEC_CSRF_TOKEN\'; \
var tokenValue = \'%{session.csrf_token}\'; \
\
--CUT--
</script>'"
```

CSRF Rules (2)

Validating Tokens

```
SecRule &ARGS "@ge 1" "chain,phase:
2,t:none,deny,log,ctl:auditLogParts=+E,msg:'CSRF
Attack Detected - Missing CSRF Token.'"
      SecRule &ARGS:MODSEC_CSRF_TOKEN "!@eq 1"

SecRule &ARGS "@ge 1" "chain,phase:
2,t:none,deny,log,msg:'CSRF Attack Detected -
Invalid Token.'"
      SecRule ARGS:MODSEC_CSRF_TOKEN "!@streq %
{SESSION.CSRF_TOKEN}"
```

CSRF Content Injection

POST request to http://www.webgoat.net/WebGoat/attack?Screen=118&menu=900

previous | next | action

request | modified request | response

raw | params | headers | hex | viewstate

```
POST /WebGoat/attack?Screen=118&menu=900 HTTP/1.1
Host: www.webgoat.net
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.6) Gecko/2009011913 Firefox/3.0.6
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://www.webgoat.net/WebGoat/attack?Screen=118&menu=900&MODSEC_CSRF_TOKEN=4ffc9823f93eb12000378159c7536096592d41fd
Cookie: JSESSIONID=44652680ECCB7A6CAC039DDD137FF01E
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
Content-Type: application/x-www-form-urlencoded
Content-Length: 123

title=Nice+Site%21&message=I+like+this+site+%3A%29&SUBMIT=Submit&MODSEC_CSRF_TOKEN=4ffc9823f93eb12000378159c7536096592d41fd
```

< > 0 matches

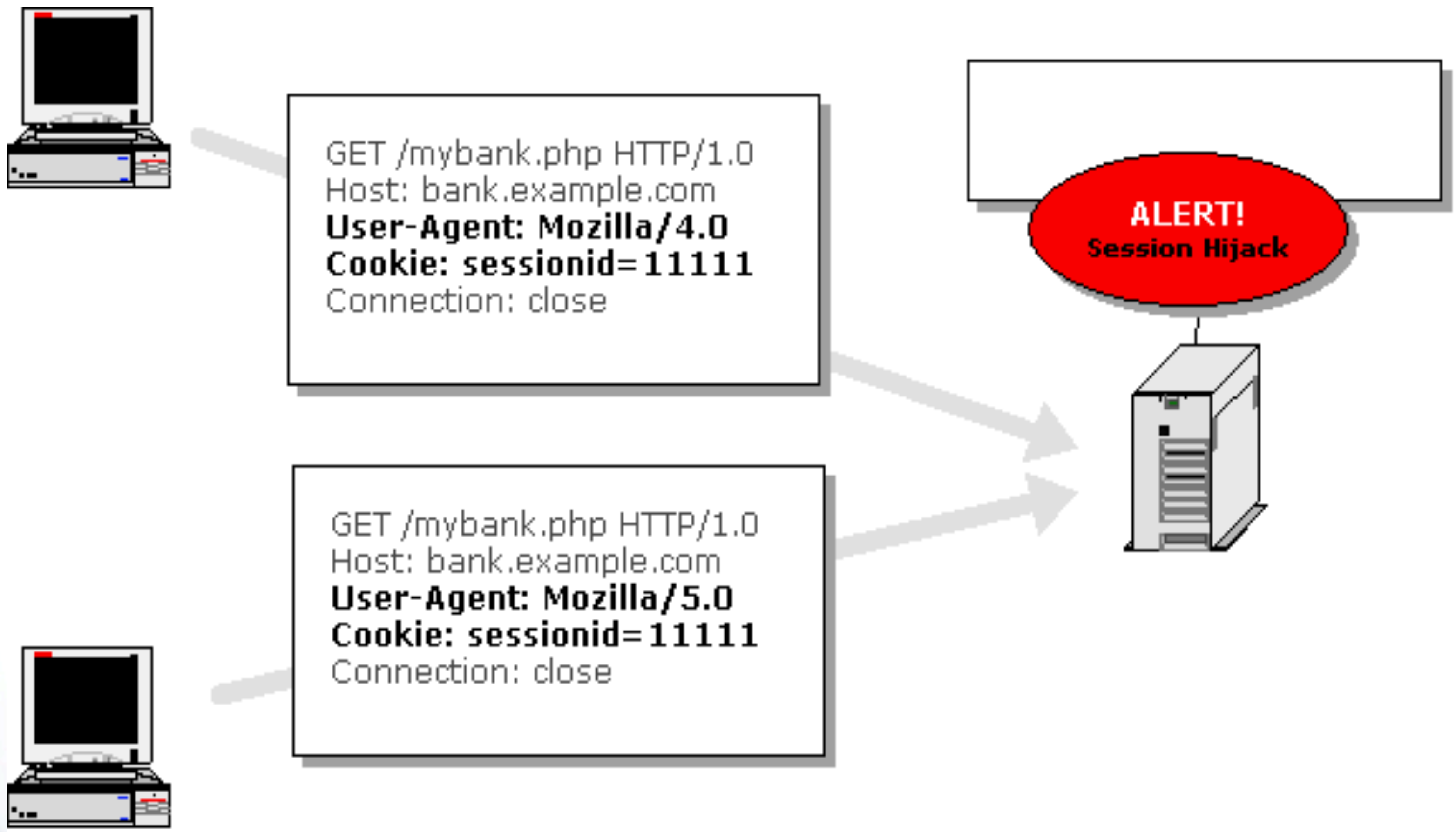
SOLUTION EXAMPLE:

SESSION MANAGEMENT FLAWS

Session-Based Attacks

- Defect:
  - The web application does not “remember” who it issued the SessionID to.
  - Clients submit SessionIDs that the web application did not issue (via Set-Cookie)
- Vulnerability:
  - Attacker can use SessionIDs that belong to other users.
- Technique:
  - Brute Force Guessing SessionIDs
  - Session Hijacking
  - Session Fixation
- Consequence:
  - Session Hijacking

Identifying Session Attacks



Session Flaw Mitigation

Tracking SessionIDs

Use ModSecurity's Session Persistent Collection

Data is saved for each SessionID

Capture Hash Values of Meta-Data and Save in Session Collection

Valid Session Token	IP Network Block Hash	User-Agent Hash

Validate SessionID Data on Subsequent Requests

Valid Session Token	IP Network Block Hash	User-Agent Hash

Session Rules (1)

Storing SessionID Meta-Data

```
SecRule RESPONSE_HEADERS:/Set-Cookie2?/ "(?
i:jsessionid=([a-f0-9]+)\;\s?)" "phase:
3,t:none,pass,log,capture,msg:'Captured session id
from response cookie: %{TX.1}',setsid:%{TX.
1},setvar:session.sessionid=%{TX.1},setvar:tx.ip=%
{remote_addr},setvar:tx.ua=%{request_headers.user-
agent},setvar:session.valid=1"


SecRule TX:IP "^(\d{1,3}\.\d{1,3}\.\d{1,3}\.)"
"phase:
3,capture,t:none,t:sha1,t:hexEncode,nolog,pass,setvar
:session.ip=%{tx.1}"

SecRule TX:UA "(.*)" "phase:
3,capture,t:none,t:sha1,t:hexEncode,nolog,pass,setvar
:session.ua=%{tx.0}"
```

Session Rules (2)

Validate SessionID Meta-Data

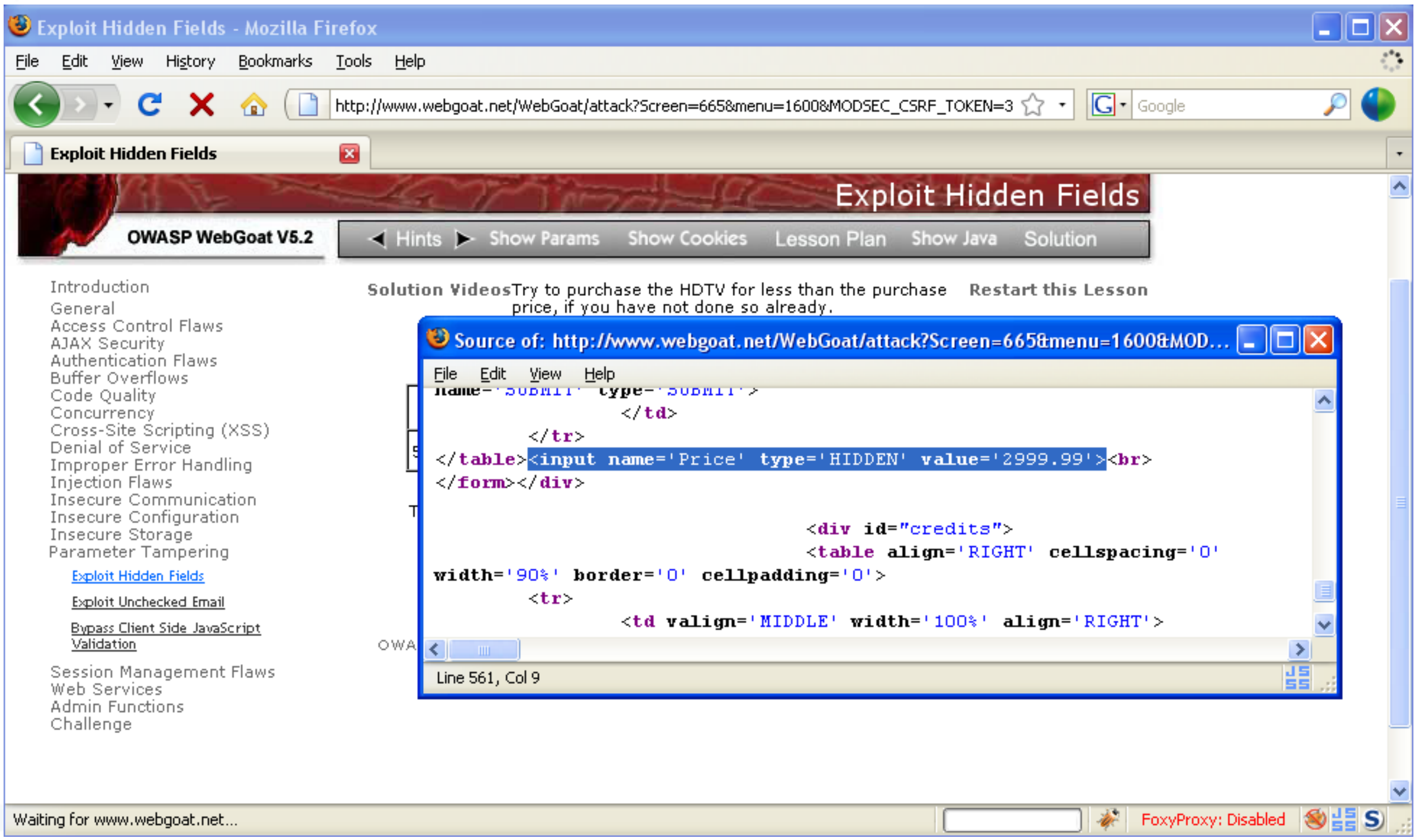
```
SecRule REQUEST_COOKIES:JSESSIONID "!^$" "phase:1,t:none,pass,nolog,setsid:%
{request_cookies.jessionid},setvar:session.sessionid=%
{request_cookies.jsessionid},setvar:tx.ip=%{remote_addr},setvar:tx.ua=%
{request_headers.user-agent}"

SecRule &SESSION:VALID "!@eq 1" "phase:1,t:none,deny,log,msg:'Invalid SessionID
Submitted.'"

SecRule TX:IP "^(\d{1,3}\.\d{1,3}\.\d{1,3}\.)" "phase:
2,capture,t:none,t:sha1,t:hexEncode,nolog,pass,setvar:tx.ip_hash=%{tx.1}"

SecRule TX:UA "(.*)" "phase:
2,capture,t:none,t:sha1,t:hexEncode,nolog,pass,setvar:tx.ua_hash=%{tx.0}"

SecRule TX:IP_HASH "!@streq %{SESSION.IP}" "phase:
2,t:none,pass,log,setvar:tx.sticky_session_anomaly=+1,msg:'Warning - Sticky
SessionID Data Changed - IP Address Mismatch.'"

SecRule TX:UA_HASH "!@streq %{SESSION.UA}" "phase:
2,t:none,pass,log,setvar:tx.sticky_session_anomaly=+1,msg:'Warning - Sticky
SessionID Data Changed - User-Agent Mismatch.'"

SecRule TX:STICKY_SESSION_ANOMALY "@eq 2" "phase:2,t:none,deny,log,msg:'Warning -
Sticky SessionID Data Changed - IP Address and User-Agent Mismatch.'"
```

SOLUTION EXAMPLE:
HIDDEN PARAMETER TAMPERING

Hidden Parameter Tampering

- Defect:
  - The web application keeps track of session state data by adding on “HIDDEN” form parameters
- Vulnerability:
  - Attackers can manipulate this data.
- Technique:
  - Attack can edit page source or use a local web proxy to intercept the response data and change data
- Consequence:
  - Session Hijacking, Business Logic Flaws

Hidden Parameter Tampering



Parameter Manipulation Mitigation

Use ModSecurity's Session Persistent Collection

Data is saved for each SessionID

Inspect Response Body Payload for HIDDEN Parameter Data

Save HIDDEN data in Session Collection

Validate Parameter Data on Subsequent Requests

- Check if saved Hidden Parameter Name Exists in Request
- Ensure the Hidden Parameter Data is Unaltered

Parameter Manipulation Rules (1)

Capture Outbound HIDDEN Data

```
SecRule RESPONSE_BODY "(<input\s.*type=[\"']?
hidden[\"']?[\s>][^<]*>)" "chain,phase:
4,t:none,t:lowercase,pass,nolog,capture,setvar:t
x.hidden_data=%{tx.1}"
      SecRule TX:HIDDEN_DATA "<input
\s.*name=[\"']?([\w\s]*)[\"']?[\s>]"
"chain,capture,setvar:session.hidden_arg_name=%
{tx.1}"
            SecRule TX:HIDDEN_DATA "<input
\s.*value=[\"']?([\w\s\.]*)[\"']?[\s>]"
"capture,setvar:session.hidden_arg=%
{session.hidden_arg_name}=%{tx.1}"
```

Parameter Manipulation Rules (1)

Validate Inbound Parameter Data

```
SecRule &SESSION:HIDDEN_ARG_NAME "@gt 0"
"chain,phase:
2,t:none,log,auditlog,deny,msg:'Hidden
Parameter Manipulation.'"
      SecRule ARGS_POST_NAMES
"@contains %{SESSION.HIDDEN_ARG_NAME}"
"chain"
            SecRule REQUEST_BODY "!
@contains %{SESSION.HIDDEN_ARG}"
"t:none,t:lowercase"
```

Viewing Hidden Collection Data

```
# java -cp /root/org.jwall.tools.jar
org.jwall.tools.CollectionViewer /tmp/
Reading collections from /tmp
Collection default_SESSION, last read @ Fri Jan 30 04:08:39
EST 2009
Created at Fri Jan 30 04:05:56 EST 2009
default_SESSION[].sessionid =
D798FE268D317B360020B9D797EFF2A1
default_SESSION[].hidden_arg = price=2999.99
default_SESSION[].ip =
d9df736088f7a4a919e4de2634d4b53d487a3b26
default_SESSION[].ua =
467cbdf2fcbeef118adf68237f3ead3a5c7b1670
default_SESSION[].hidden_arg_name = price
default_SESSION[].TIMEOUT = 3600
This collection expires in 59m 32.943s
```

The Need for Lua

- ModSecurity Rules Language Limitations
  - Allows for easy “and” logic but it is difficult to do if/then/or structures
  - RegEx parsing has problems with capturing multiple individual elements (e.g. – more than 1 HIDDEN parameter)
- ModSecurity has a Lua API
  - User creates scripts that use advanced programming logic
- Stephen Evans created many example Lua scripts for WebGoat mitigations
- Content Injection (Javascript) + Lua is a powerful virtual patching combination
  - Not constrained by pre-packaged, WAF GUI functionality

www.lua.org

Lua Scripts

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

```
local tbuff = m.getvar("RESPONSE_BODY", "none")
for a in string.gmatch(tbuff, "<input .->") do
  t = {}
  for k, v in string.gmatch(a, "(%w+)='(.-)'") do
    t[k]=v
  end
  if t.type:lower() == "hidden" then
    -- write t.type, t.name and t.value to file
end
Entry{
  name = "hidden_tan",
  type = "HIDDEN",
  value = "2"
```

Conclusion

- A WAF is more than an “attack blocking device.”
  - Can also identify/correct Application Defects.
  - Can be used as an HTTP Auditing device.
- There is a tremendous need for Virtual Patching:
  - Expedite the implementation of mitigations.
  - Provide protection for apps that can’t be updated.
- ModSecurity is an excellent, tactical tool to use for mitigation strategies
  - Robust rules language
  - Content Injection + Lua is powerful

Questions?

- Thank you!

Business: Ryan.Barnett@breach.com

Personal: RCBarnett@gmail.com