WAF Virtual Patching Challenge: Securing WebGoat with ModSecurity

Ryan Barnett
Breach Security (www.breach.com)
rbarnett@breach.com

Revision 1 (January 20, 2009)

Abstract

In this paper, we present the technical details behind a *virtual patch,* which is a critical protection function provided by web application firewalls (WAFs). A virtual patch is a powerful, agile mitigation strategy to quickly help protect vulnerable web applications from remote compromise. During the course of this whitepaper, we will evaluate a number of example vulnerabilities from the OWASP WebGoat application. The context of these examples helps to quantify the significant research responsibilities of the virtual patch writer, and highlights how ModSecurity's rules language and advanced capabilities afford security consultants with a platform to mitigation complex vulnerabilities identified within a web application.

Introduction

Organizations are now realizing that traditional security products such as network firewalls, intrusion detection systems (IDS) and intrusion prevention systems (IPS) are not sufficient for protecting today's web applications from compromise. Network firewalls do not adequately analyze application layer protocol data for signs of attack, intrusion detection systems do not take any action to stop an attack that is detected and intrusion prevention systems suffer from a lack of understanding the nuances of the HTTP traffic.

Two Separate Approaches to Remediation

Since network defenses miss attacks against vulnerable web applications, efforts are therefore made to eliminate the risk by either securing the application with a patch from development or deploying a web application firewall (WAF) in front of the web application to protect it from attack.

Securing the Code

Many organizations have found that testing their applications for vulnerabilities and then understanding and prioritizing the results require a great deal of expertise. This is further expounded by the need to then communicate information about vulnerabilities to the application developers for eventual

remediation in the code. Organizations without the necessary expertise are often unable to fully understand the risk exposed by their applications and therefore do not give it much prioritization.

The best option for most organizations is to outsource their security assessment to a security service company. The service provides a complete assessment of applications for vulnerabilities and analysis of results by security experts. This ensures that results are free of false positives and properly prioritized. The experts also work with an organization's development team to ensure that the vulnerability and remediation steps are well understood and properly implemented.

All efforts to ensure that vulnerabilities are identified and then remediated by the development team still expose organizations to an unacceptable degree of risk. Remediating vulnerabilities in web applications is certainly not immediate as commercial software vendors infrequently patch applications and updates for internally developed applications require time for coding and testing. The end result is that the patching process takes a while and there is a significant window of opportunity where web applications are unprotected and the vulnerability can be exploited.

Web Application Firewalls

Web application firewalls (WAFs) eliminate the gap in network security defenses with specific technologies for applying application context-specific granular analysis of web traffic to block attacks and can revolutionize the approach taken for mitigating identified web application vulnerabilities. A WAF analyzes web requests before the traffic is sent to the web application and is able to block malicious web traffic; not just passively detect it.

WAFs include granular analysis capabilities specific to web applications that permit extremely precise validation of application communication. Web application firewalls inspect everything from user entry fields to URLs, and headers as well as monitor user sessions and cookies, and block any leakage of sensitive data. One significant feature offered by a WAF is a "virtual patch" functionality that protects web applications from a discovered vulnerability when normal software patches cannot be applied and the web application is at risk. A virtual patch is a powerful, scalable protection against compromise.

Web application firewalls can protect applications against most threats, but not everything. One particular weakness is business logic flaws. If an application is performing as designed, it is difficult for a WAF to determine that something wrong has occurred from a security perspective. WAFs often require customization to most efficiently protect more complex applications.

Organizations need a new security solution.

Virtual Patching – A Stronger Combined Solution

A better solution is to combine the results of an assessment with the virtual patching capability of a web application firewall to immediately remediate identified vulnerabilities in web applications. Organizations will be able to virtually eliminate their period of risk exposure between identifying and patching vulnerabilities.

Goals

The goal with this paper is to present a virtual patching framework that organizations can follow to maximize the timely implementation of virtual patches, as well as, to demonstrate how the ModSecurity web application firewall can be used to remediate a sampling of vulnerabilities in the OWASP WebGoat application.

What is a Virtual Patch?

The term virtual patching was originally coined by Intrusion Prevention System (IPS) vendors a number of years ago. It is not a web application specific term, and may be applied to other protocols however currently it is more generally used as a term for Web Application Firewalls (WAF). It has been known by many different names including both *External Patching* and *Just-in-time Patching*. Whatever term you choose to use is irrelevant. What is important is that you understand exactly what a virtual patch is. Therefore, I present the following definition:

> *A policy for an intermediary device (i.e. - Web Application Firewall - WAF) that is able to identify and block attempts to exploit a specific application vulnerability.*

The virtual patch works since the WAF analyzes transactions and intercepts attacks in transit, so malicious traffic never reaches the web application. The resulting impact of virtual patch is that, while the actual source code of the application itself has not been modified, the exploitation attempt does not succeed.

Why Not Just Fix the Code?

From a purely technical perspective, the number one remediation strategy would be for an organization to correct the identified vulnerability within the source code of the web application. This concept is universally agreed upon by both web application security experts and system owners. Unfortunately, in real world business situations, there arise many scenarios where updating the source code of a web application is not easy. Common roadblocks to source code fixes include:

Patch Availability

If a vulnerability is identified within a commercial application, the customer most likely will not be able to modify the source code themselves. In this situation, the customer is held at the mercy of the Vendor as they have to wait for an official patch to be released. Vendors usually have extremely rigid patch release dates, which mean that an officially supported patch may not be available for an extended period of time.

Installation Time

Even in situations where an official patch is available, or a source code fix could be applied to a custom coded application, the normal patching processes of most organizations is time consuming. This is usually due to the extensive regression testing required after code changes. It is not uncommon for these testing gates to be measured in months. For example, the Symantec Internet Threat Report [1] stated that the average time it took for organizations to patch their systems was 55 days, while the Whitehat Security Web Security Statistics Report [2] documented that their customers time-to-fix average was 138 days to remediate SQL Injection vulnerabilities found in their web applications. Now contrast this patching data with the fact that Symantec also reported that it only took an average of 6 days for exploit code to be released to the public and it becomes clear that traditional source code patching processes are not adequate.

Fixing Custom Code is Cost Prohibitive

Web assessments that include source code reviews, vulnerability scanning and penetration tests will most assuredly identify vulnerabilities in your web application. Identification of the vulnerability is only the first half of the battle with the second half being the remediation actions. What many organizations are finding out is that the cost associated with the identification of the vulnerabilities often pales in comparison to that of actually fixing the issues. This is especially true when vulnerabilities are not found early in the design or testing phases but rather after an application is already in production. In these situations, it is usually deemed that it is just too expensive to recode the application.

Legacy Code

An organization may be using a commercial application and the vendor has gone out of business, or they are using a version that is no longer supported by the vendor. In these situations, legacy application code can't be patched. An additional situation is when an organization is forced into using outdated vendor code due to in-house custom coded functionality being added on top of the original vendor code. This functionality is tied to a mission critical business application and prior upgrade attempts broke functionality.

Outsourced Code

As more and more businesses opt to outsource their application development, they are finding that executing vulnerability fixes would require an entirely new project.

Many organizations are facing the harsh reality that poor contractual language oftentimes does on cover “secure coding” issues but only functional defects.

Value of Virtual Patching

When you consider the numerous situations when organizations can’t simply immediately edit the source code, the value of virtual patching becomes apparent. From an organizations perspective, the benefits are:

- It is a scalable solution as it is implemented in few locations vs. installing patches on all hosts.
- It reduces risk until a vendor-supplied patch is released or while a patch is being tested and applied.
- There is less likelihood of introducing conflicts as libraries and support code files are not changed.
- It provides protection for mission-critical systems that may not be taken offline.
- It reduces or eliminates time and money spent performing emergency patching.
- It allows organizations to maintain normal patching cycles.

From a web application security consultant’s perspective, virtual patching opens up another avenue for providing services to your clients. Traditionally, if source code could not be updated for any of the reasons previously specified, there wasn’t much else a consultant could do to help. Now, a consultant can offer to create virtual patches to externally address the issues outside of the application code.

Why ModSecurity?

While there are other web application firewall applications, ModSecurity is uniquely qualified as the premier option. This is mainly attributed to two factors. First, ModSecurity is an open source, free web application firewall. The fact that there is no cost associated with its use is primarily why it is the most widely installed WAF with more than 10,000 installations woldwide. Second, it boasts a robust rules language and has a number of unique capabilities (outlined below) which allows it to mitigate complex vulnerabilities.

Robust HTTP and HTML Parsing

ModSecurity employs an HTTP and HTML parser to analyze the input stream. The parser is able to understand specific protocol features including content encoding such as chunked encoding or multipart/form-data encoding, request and response compression and even XML payload.

In addition the parser is flexible as the environment protected as many headers and protocol elements are not used according to RFC requirements. For example, while the RFC requires a single space between the method and the URI in the HTTP request line, Apache allows any sequence of whitespace between them. Another example is PHP unique use of parameters: in PHP leading and trailing spaces are removed from parameter names.

In a proxy deployment a stricter parsing may be acceptable, but ModSecurity is deployed a manner in which only a copy of the data inspected, the WAF has to be at least as flexible as the web server in order to prevent evasion. IDS/IPS systems that fail to do so can be easily evaded by attackers. [3]

Protocol Analysis

Based on the parsed info, ModSecurity must break up the HTTP stream into logical entities that can be inspected, such as headers, parameters and uploaded files. Each element is inspected separately not just for its content, but also for its length and count. In addition ModSecurity must correctly divide the network stream when keep-alive HTTP connections are used to unique request and replies, and correctly match requests and replies.

Anti-Evasion Capabilities

Modern protocols such as HTTP and HTML allow the same information to be presented in multiple ways. As a result signature based detection of attacks must inspect the attack vector in any form it may be in. Attackers evade detection systems by using a less common presentation of the attack vector. Some common evasion techniques include using different character encodings for the attack vector or using none canonized path names. In order to prevent evasion ModSecurity transforms the request to a normalized form before inspection.

While modern IDPS systems may support anti-evasion techniques, those are limited to well defined parts of the request such as the URI. ModSecurity can selectively employ normalization functions for different input fields for each inspection performed. For example, while an IDPS would normalize the URI, ModSecurity can normalize an HTML form field that accepts path names as input.

Rules instead of Signatures

Virtual patches must implement complex logic, as it cannot rely solely on signatures and requires a more robust rules language to define the tests. For example, the following features exist in the ModSecurity rules language:

- Operators and logical expressions – can check an input field for attributed other than its content, such as its size or character distribution. Additionally ModSecurity can combine such atoms to create more complex conditions using logical operators. For example, it may inspect if a field length is too long only for a specific value of another field, or alternatively check if two different fields are empty.

- Selectable anti-evasion transformation functions – as discussed above, each rule can employ specific transformation function.

- Variables, sessions & state management – since the protocols inspected keep state, the rules language needs to include variables. Such variables can persist for a single transaction, for the life of a session, or globally. Using such variables enables ModSecurity to aggregate information and therefore detect an attack based on multiple indications during the life span of a transaction or a session. Attacks that require such mechanisms to detect are brute force attacks, application layer denial of service attacks and business logic flaws.

- Control structures – the ModSecurity rules language includes control structures such as conditional execution. Such structures enable ModSecurity to perform different rules based on transaction content. For example, if the transaction payload is XML, an entirely different set of rules may be used.

A Virtual Patching Methodology

Virtual Patching, like most other security processes, is not something that should be approached haphazardly. Instead, a consistent, repeatable process should be followed that will provide the best chances of success. The following virtual patching workflow mimics the industry accepted practice for conducting IT Incident Response and consists of the following phases: *Preparation, Identification, Analysis, Virtual Patch Creation, Implementation/Testing, and Recovery/Follow T Up*.

Preparation Phase

The importance of properly utilizing the preparation phase with regards to virtual patching cannot be overstated. The idea is that you need to do a number of things to setup the virtual patching processes and framework prior to actually having to deal with an identified vulnerability, or worse yet, react to a live web application intrusion. The point is that during a live compromise is not the ideal time to be proposing installation of a web application firewall and the concept of a virtual patch. Tension is high during real incidents and time is of the essence, so lay the foundation of virtual patching when the waters are calm and get everything in place and ready to go when an incident does occur. Here are a few critical items that should be addressed during the preparation phase:

- Ensure that you are signed up for on all vendor alert mail-lists for commercial software that you are using. This will ensure that you will be notified in the event that the vendor releases vulnerability information and patching data.
- Virtual Patching Pre-Authorization – Virtual Patches need to be implemented quickly so the normal governance processes and authorizations steps for standard software patches need to be expedited.

Since virtual patches are not actually modifying source code, they do ***not*** require the same amount of regression testing as normal software patches. I have found that categorizing virtual patches in the same group as Anti-Virus updates or Network IDS signatures helps to speed up the authorization process and minimize extended testing phases.

- Deploy ModSecurity In Advance - As time is critical during incident response, it would be a poor time to have to get approvals to install new software. You can install ModSecurity in embedded mode on your Apache servers, or an Apache reverse proxy server. The advantage with this deployment is that you can create fixes for non-Apache back-end servers. Even if you do not use ModSecurity under normal circumstances, it is best to have it "on deck" ready to be enabled if need be.
- Increase Audit Logged – The standard Common Log Format (CLF) utilized by most web servers does not provide adequate data for conducting proper incident response. Consider the following Apache access_log entry:

```
80.87.72.6 - - [22/Apr/2007:18:55:53 --0400] \
"POST /xmlrpc.php HTTP/1.1" 200 293
```

We see that this request uses a POST Request Method. This means that critical data such as the Request Body (where the client is passing parameter data) is not logged. Without the full request payloads, it is next to impossible to accurately confirm either an attack attempt or a successful compromise. Fortunately, ModSecurity has a robust audit logging engine that is able to capture the entire request and response payloads. The following audit log entry is for the same xmlrpc.php request we showed from the Apache access_log file.

```
--ddb9bf17-A--
[22/Apr/2007:18:55:53    --0400]    dGgsYX8AAAEAABJkpY8AAACG
80.87.72.6 41376 192.168.1.133 80
--ddb9bf17-B--
POST /xmlrpc.php HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: www.example.com
User-Agent: libwww-perl/5.805
Content-Length: 201
--ddb9bf17-C--
<?xml
version="1.0"?><methodCall><methodName>test.method</methodNa
me><params><param><value><name>',''));echo'_begin_';echo
`id;ls/;w`;echo
'_end_';exit;/*</name></value></param></params></methodCall>
```

As you can see, now that we can see the request body contents, we are able to identify that the client is attempting to exploit the php application and is attempting to execute OS command injection.

Identification Phase

The Identification Phase occurs when an organization becomes aware of a vulnerability within their web application. There are generally two different methods of identifying vulnerabilities: Proactive and Reactive.

Proactive Identification

This occurs when an organization takes it upon themselves to assess their web security posture and conducts the following tasks:

- Vulnerability assessment (internal or external) and penetration tests
- Source code reviews

These tasks are extremely important for custom coded web applications as there would be external entity that has the same application code.

Reactive Identification

There are three main reactive methods for identifying vulnerabilities:

- Vendor contact (e.g. pre-warning) - Occurs when a vendor discloses a vulnerability for commercial web application software that you are using.
- Public disclosure - Public vulnerability disclosure for commercial/open source web application software that you are using. The threat level for public disclosure is increased as more people know about the vulnerability.
- Security incident – This is the most urgent situation as the attack is active. In these situations, remediation must be immediate. Normal network security response measures include blocking the source IP of the attack at a firewall or edge security device. This technique does not work as well for web application attacks as you may prevent legitimate users from accessing the application. A virtual patch is more flexible as it is not necessarily ***where*** an attacker is coming from but ***what*** they are sending.

Analysis Phase

There are a number of tasks that must be completed during the analysis phase.

What is the name of the vulnerability?

This means that you need to have the proper CVE name/number identified by the vulnerability announcement, vulnerability scan, etc…

What is the impact of the problem?

It is always important to understand the level of criticality involved with a web vulnerability. Information leakages may not be treated in the same manner as an SQL Injection issue.

What versions of software are affected?

You need to identify what versions of software are listed so that you can determine if the version(s) you have installed are affected.

What configuration is required to trigger the problem or how to tell if you are affected by the problem?

Some vulnerabilities may only manifest themselves under certain configuration settings.

Is proof of concept exploit code available?

Many vulnerability announcements have accompanying exploit code that shows how to demonstrate the vulnerability. If this data is available, make sure to download it for analysis. This will be useful later on when both developing and testing the virtual patch.

Is there a work around available without patching or upgrading?

This is where virtual patching actually comes into play. It is a temporary work-around that will buy organizations time while they implement actual source code fixes.

Is there a patch available?

Unfortunately, vulnerabilities are often announced without an accompanying patch. This leaves organizations exposed and is why virtual patching has become an invaluable tool. If there is a patch available, then you initiate the proper patch management processes and simultaneously create a virtual patch.

Virtual Patch Creation Phase

The process of creating an accurate virtual patch is bound by two main tenants:

1. *No false positives*. Do not ever block legitimate traffic under any circumstances. This is always the top priority.
2. *No false negatives*. Do not miss attacks, even when the attacker intentionally tries to evade detection. This is a high priority.[4]

The virtual patch creator must keep these priorities, and their relative ordering, in mind at all times. A key distinction between virtual patch construction philosophies (log-only mode vs. a blocking mode) lies in the relative ranking of these two goals. The art of creating blocking virtual patches is generalizing the detection logic as much as possible to rigorously meet rule #2, without ever violating rule #1.

Deriving a Zero False Negative Virtual Patch

When performing technical vulnerability research, the virtual patch writer must first search for all of the necessary conditions for an attack to succeed. The researcher starts by obtaining technical data that triggers the vulnerability remotely (perhaps from proof of concept exploit code). The writer then varies or fuzzes all the “interesting-looking” parts of the attack. Changes are made one at a time, in steps, keeping careful notes. (Strings, length values, character encoding, white

space… the list goes on. All are good things to vary.) If the attack succeeds even when a particular variable is set to a random value, *that variable is not important for the virtual patch criteria*. Eventually the researcher can identify the complete set of variables that are important to the attack's success, and arrive at a set of criteria that must be collectively satisfied for any attack to succeed. If there are multiple distinct attack vectors, the researcher must perform this analysis on each one separately.

Given a set of criteria that must be satisfied for an attack to succeed, it is possible to describe virtual patching logic that has zero false negatives. That is, an attack simply cannot succeed unless the associated web application attack traffic has exactly the characteristics that the virtual patch is looking for.

Deriving a Zero False Positive Virtual Patch

Given a zero false negative virtual patch as previously described, the writer must also evaluate the accuracy of patch in terms of false positives. At this stage, the writer attempts to identify at least one characteristic that would never occur in normal web traffic. If a characteristic exists that is both anomalous compared to normal traffic and critical to the attack's success, then the zero false negative virtual patch is also a zero false positive signature.

Negative Security Virtual Patches

A negative security model (or misuse based detection) is based on a set of rules that detect specific known attacks rather than allow only valid traffic. It is important to note that the differentiation between negative and positive security models is subjective and reflects how tight the security envelope around the application is. A good example would be limiting the characters allowed in an input field. Since the character set is a closed set, providing a white list of permitted characters is actually similar to providing a black list of forbidden characters including the characters complementing the 1st group.

Positive Security Virtual Patches

Positive security model is a comprehensive security mechanism that provides an independent input validation envelope to an application. The model specifies the characteristics of valid input (character set, length, etc…) and denies anything that does not conform. By defining rules for every parameter in every page in the application the application is protected by an additional security envelop independent from its code.

Which Method is Better for Virtual Patching – Positive or Negative Security?

A virtual patch may employ either a negative or positive security model. Which one you decide to use depends on the situation and a few different considerations. For example, negative security rules can usually be implemented more quickly, however the possible evasions are more likely.

Positive security rules, only the other hand, provides better protection however it is often a manual process and thus is not scalable and difficult to maintain for

large/dynamic sites. While manual positive security rules for an entire site may not be feasible, a positive security model can be selectively employed when a vulnerability alert identifies a specific location with a problem.

Beware of Exploit-Specific Virtual Patches

You want to resist the urge to take the easy road and quickly create an exploit-specific rule. While it may provide some immediate protection, its long term value is significantly decreased. A case study of this concept in the IDS world is "bleeding edge" snort signature for Bugtraq vulnerability #21799. This vulnerability in the Cacti open source graphing software was picked quite at random. The exploit references on Bugtraq vulnerabilities archive is:

```
/cacti/cmd.php?1+1111)/**/UNION/**/SELECT/**/2,0,1,1,127.0.0
.1,null,1,null,null,161,500, proc,null,1,300,0, ls -la >
./rra/suntzu.log,null,null/**/FROM/**/host/*+11111
```

And the Snort signature is:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(
msg:"BLEEDING-EDGE WEB Cacti cmd.php Remote Arbitrary
SQL Command Execution Attempt"; flow:to_server,established;
uricontent:"/cmd.php?"; nocase;
uricontent:"UNION"; nocase;
uricontent:"SELECT"; nocase;
reference:cve,CVE-2006-6799; reference:bugtraq,21799;
classtype: web-application-attack; sid:2003334; rev:1;
)
```

While snort has some anti-evasion techniques such as case insensitivity and URI decoding, this signature still falls short of detecting an exploit of the vulnerability. It is gears only towards detecting the specific attack vector shown above. Any other exploit such as blind SQL injection would not be detected. It also searches for the keywords only in the request line, while many development environments would allow for parameters to be provided both in the POST and GET payload.

Additionally this signature is prone to false positives as both select and union are common English words and since the signature do not require any word delimiters the signature will also be satisfied by the words "Selection" and "Reunion". In many cases such a signature has to be turned off.

For examples of poorly written ModSecurity rules, let's look at the following GotRoot rule:

```
SecDefaultAction "log,deny,phase:2,status:500,t:urlDecodeUni, \
t:htmlEntityDecode,t:lowercase"

# WEB-CGI csSearch.cgi arbitrary command execution attempt
SecRule REQUEST_URI "/csSearch\.cgi\?" chain
SecRule REQUEST_URI "\`"
```

In the first line, the SecDefaultAction is specifying the use of the "t:lowercase" transformation function. This is often used to normalize input data for anti-

evasion. When this is used, care should be taken to specify only lowercase letters in the operator payload section. In this rule example, however, the rule writer mistakenly used mixed-case and thus this rule would not trigger (false negative).

Implementation/Testing Phase

In order to accurately test out the newly created virtual patches, it may be necessary to use an application other than a web browser. Some useful tools are:

- Command line web clients such as Curl and Wget.
- Local Proxy Servers such as WebScarab (://www.owasp.org/index.php/Category:OWASP_WebScarab_Project) and Burp Proxy (://www.portswigger.net/suite/).
- ModSecurity AuditViewer (://www.jwall.org/web/audit/viewer.jsp) – which allows you to load a ModSecurity audit log file, manipulate it and then re-inject the data back into any web server.

These tools will allow you to manipulate the request data in any way desired.

ModSecurity's Debug Log File

In order to verify exactly how your new rule is working, you should review the ModSecurity SecDebugLog file data. The Debug log provides extensive details on the rule processing order and in many cases is the only true way to verify that the rule is working exactly as you expected. You will most likely need to increase the SecDebugLogLevel directive setting to get enough detail to validate the patch processing. You can selectively increase the logging based on source IP address so that you don't impact performance on the entire web server. Below is an excerpt of the debug log data during rule processing (some data deleted for readibility):

```
Recipe: Invoking rule 82211d8.
Executing operator !rx with param "^(POST)$" against
REQUEST_METHOD.
Target value: POST
Operator completed in 17 usec.
Rule returned 0.
No match, not chained -> mode NEXT_RULE.
Recipe: Invoking rule 82214b0.
Rule returned 0.
No match, not chained -> mode NEXT_RULE.
Recipe: Invoking rule 82360d0.
Executing operator !rx with param "^(\w{0,32})$" against
ARGS:username.
Target value:
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000
Operator completed in 13 usec.
Rule returned 1.
```

```
Match, intercepted -> returning.
Access denied with code 501 (phase 2). Match of "rx ^(\w{0,32})$"
against "ARGS:username" required. [id "1"] [msg "Postparameter
username failed validity check. Value domain: Username."]
[severity "ERROR"]
```

Recovery/Follow-Up Phase

Although you may need to expedite the implementation of virtual patches, you should still track them in your normal Patch Management processes. This means that you should create proper change request tickets, etc… so that their existence and functionality is documented.

You should also have periodic re-assessments to verify if/when you can remove previous virtual patches if the web application code has been updated with the real source code fix. I have found that many people opt to keep virtual patches in place due to better identification/logging vs. application or db capabilities.

Securing WebGoat with ModSecurity

In the summer of 2008, Stephen Craig Evans lead an OWASP Summer of Code (SoC) Project entitled: Securing WebGoat with ModSecurity (://www.owasp.org/index.php/OWASP_Securing_WebGoat_using_ModSecurity_Project). The goal of the project was stated as:

> *The purpose of this project is to create custom ModSecurity rule sets 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.*

I was one of the official project reviewers. This seemed to me to be simultaneously a great challenge (as there are many vulnerabilities within WebGoat that are complex to address externally) and extremely practical. Providing a full description of every one of the approximately 50 lessons and their mitigations is beyond the scope of this whitepaper, however I will be presenting a selected few that I feel highlight some rather cutting-edge solutions.

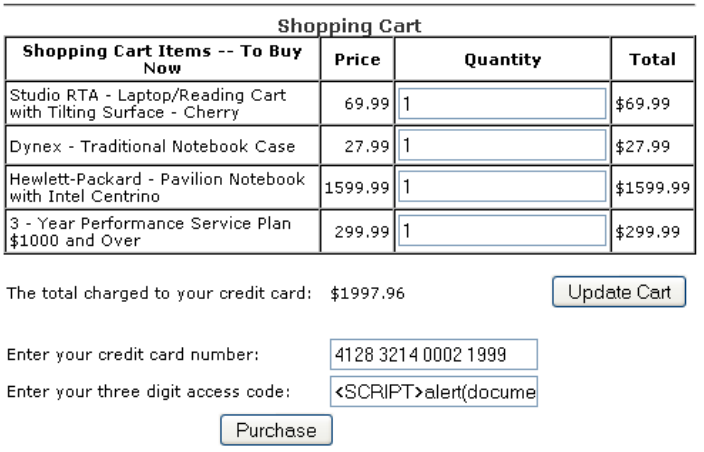
Cross-site Scripting (XSS)

Improper html output entity encoding of user supplied data, which exposes clients to Cross-site Scripting (XSS) attacks, is pretty much universally seen as the the #1 security vulnerability facing web applications. Just take a look at such resources as the Top Ten, Web Application Security Statistics Project or the Sla.ckers " so it begins" mail-list thread for evidence of the widespread existence of sites that are vulnerable to XSS attacks. Depending on the web application language your site is using, it most likely has some form of output encoding capabilities that can be configured to help mitigate the issue. Ivan Ristic also recently outlined some high level coding concepts to help address XSS.

The purpose of this section is to outline how to use ModSecurity to help address not only XSS attacks but to also address the underlying vulnerability, which is to detect web applications that aren't properly output encoding data. We will also show some offensive techniques aimed at short circuiting XSS SessionID stealing by fixing any cookies that are missing the HTTPOnly flag.

Blocking Inbound Reflected XSS Attacks

In the Cross-Site Scripting (XSS) -> LAB: Reflected XSS Attack, an attacker can send malicious javascript in the "Enter your three digit access code" field -



The subsequent request payload would look similar to the following:

```
POST /WebGoat/attack?Screen=185&menu=900 HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
rv:1.9.0.5) Gecko/2008120122 Firefox/3.0.5
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
Connection: keep-alive
Referer: http://localhost:8080/WebGoat/attack?Screen=185&menu=900
Cookie: JSESSIONID=D7B05470E93E267EFA86A13E31A293F8
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
Content-Type: application/x-www-form-urlencoded
Content-Length: 134

QTY1=1&QTY2=1&QTY3=1&QTY4=1&field2=4128+3214+0002+1999&field1=%3CS
CRIPT%3Ealert%28document.cookie%29%3B%3C%2FSCRIPT%3E&SUBMIT=Purcha
se
```

XSS - Negative Security

The Core Rule set, which is available for free from the ModSecurity website, includes a robust negative security rule set for XSS detection. The current version

of the rule set uses complex logic combining two different operators; @pm set-based pattern matching used for fast pre-qualification of data to identify the existence of key XSS strings, and then @rx regular expression rules to apply advanced checks to both confirm XSS logic and exclude false positives. For this particular issue, we can create a new chained, targeted rule set that applies the checks to the correct location and parameter.

```
<Location /WebGoat/attack>

SecRule ARGS:field1 "@pm jscript onsubmit copyparentfolder
javascript meta onmove onkeydown onchange onkeyup activexobject
expression onmouseup ecmascript onmouseover vbscript: <![cdata[
http: settimeout onabort shell: .innerhtml onmousedown onkeypress
asfunction: onclick .fromcharcode background-image: .cookie
ondragdrop onblur x-javascript mocha: onfocus javascript:
getparentfolder lowsrc onresize @import alert onselect script
onmouseout onmousemove background application .execscript
livescript: getspecialfolder vbscript iframe .addimport onunload
createtextrange onload <input" \

"chain,t:urlDecodeUni,t:htmlEntityDecode,t:compressWhiteSpace,t:lo
wercase,ctl:auditLogParts=+E,log,auditlog,msg:'Cross-site
Scripting (XSS)
Attack',id:'950004',tag:'WEB_ATTACK/XSS',logdata:'%{TX.0}',severit
y:'2'"

      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|f
      ocus|blur)\b\W*?=|abort\b)|(?:l(?:owsrc\b\W*?\b(?:(?:java|vb
      )script|shell|http)|ivescript)|(?:href|url)\b\W*?\b(?:(?:jav
      a|vb)script|shell)|background-
      image|mocha):|s(?:(?:tyle\b\W*=.*\bexpression\b\W*|ettimeout
      \b\W*?)\(|rc\b\W*?\b(?:(?:java|vb)script|shell|http):)|a(?:c
      tivexobject\b|lert\b\W*?\(|sfunction:))|<(?:(?:body\b.*?\b(?
      :backgroun|onloa)d|input\b.*?\btype\b\W*?\bimage)\b|
      ?(?:(?:script|meta)\b|iframe)|!\[cdata\[)|(?:\.(?:(?:execscr
      ip|addimpor)t|(?:fromcharcod|cooki)e|innerhtml)|\@import)\b)
      " \

      "capture,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase
      "

</Location>
```

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

XSS - Positive Security

Here in the example: the “field1” parameter should ***only*** accept 3 digits in length.

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

```
<Location /WebGoat/attack>

SecRule &ARGS_POST_NAMES:field1 "@eq 0"
"phase:2,t:none,t:urlDecodeUni,t:normalisePathWin,t:lowercas
e,deny,log,auditlog,msg:'Input Validation Alert – Field1
Argument Missing in Post Payload',logdata:'%{MATCHED_VAR}'"

SecRule &ARGS_POST_NAMES:field1 "@gt 1"
"phase:2,t:none,t:urlDecodeUni,t:normalisePathWin,t:lowercas
e,deny,log,auditlog,msg:'Input Validation Alert – Multiple
Field1 parameters.',logdata:'%{MATCHED_VAR}'"

SecRule ARGS_POST:field1 "!^\d{3}$"

"phase:2,t:none,t:urlDecodeUni,t:normalisePathWin,t:lowercas
e,deny,log,auditlog,msg:'Input Validation Alert – Data not
in the correct format.',logdata:'%{MATCHED_VAR}'"
```

This rule set will help to prevent evasion attempts by ensuring that there is only 1 argument called “field1”, that it is only present within the post_payload data and that it has the proper format and length. Keep in mind that this type of input validation should also be incorporated within the application itself. The main reasons for implementing this type of positive security filter at the web application firewall layer are for general security-in-depth and also for those web applications where updating the code is either not possible or will take a very long time.

Application Defect Identification – Missing Output Encoding

ModSecurity does not currently manipulate inbound or outbound data so it can not, by itself, be used to entity encode user data that is returned in output. While this is true, ModSecurity can be utilized to identify when web applications are failing to properly html entity encode user data in output.

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

```
#
# XSS Detection - Missing Output Encoding
#
SecAction "phase:1,nolog,pass,initcol:global=xss_list"

#
```

```
# Identifies Reflected XSS
# If malicious input (with Meta-Characters) is echoed back in the
# reply non-encoded.
SecRule &ARGS "@gt 0" \
"chain,phase:4,t:none,log,auditlog,deny,status:403,id:'1',msg:'Pot
entially Malicious Meta-Characters in User Data Not Properly
Output Encoded.',logdata:'%{tx.inbound_meta-characters}'"
      SecRule ARGS "([\'\"\(\)\;<>/])" \
      "chain,t:none,capture,setvar:global.xss_list_%{time_epoch}=%
{matched_var},setvar:tx.inbound_meta-characters=%{matched_var}"
            SecRule RESPONSE_BODY "@contains %{tx.inbound_meta-
      characters}" "ctl:auditLogParts=+E"
```

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

```
#
# Identifies Stored XSS
# If malicious input (with Meta-Characters) is echoed back on any
# page non-encoded.
SecRule GLOBAL:'/XSS_LIST_.*/' "@within %{response_body}" \
"phase:4,t:none,log,auditlog,pass, msg:'Potentially Malicious M
eta-Characters in User Data Not Properly Output
Encoded',tag:'WEB_ATTACK/XSS'"
```

Missing HTTPOnly Cookie Flags

If you are unfamiliar with what the HTTPOnly cookie flag is or why your web apps should use it, please refer to the following resources –

- Mitigating Cross-site Scripting With HTTP-only Cookies - ://msdn.microsoft.com/en-us/library/ms533046.aspx
- OWASP HTTPOnly Overview - ://www.owasp.org/index.php/HTTPOnly

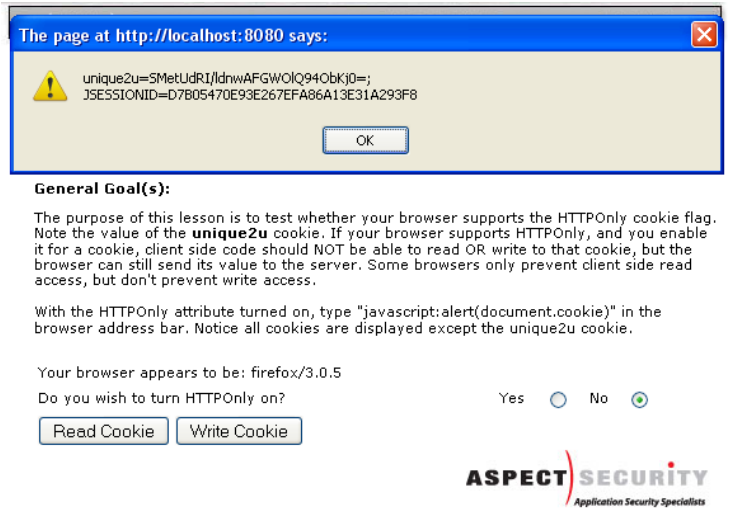
The bottom line is this - while this cookie option flag does absolutely nothing to prevent XSS attacks, it does significanly help to prevent the #1 XSS attack goal which is stealing SessionIDs. While HTTPOnly is not a "silver bullet" by any means, the potential ROI of implement it is quite large. Notice I said "potential" as in order to provide the intended protections, two key players have to work together -

- Web Applications - whose job it is to append the "HTTPOnly" flag onto all Set-Cookie response headers for SessionIDs, and
- Web Browsers - whose job it is to identify and enforce the security restrictions on the cookie data so that javascript can not access the contents.

The current challenges to realizing the security benefit of the HTTPOnly flag is that universal adoption in both web apps and browsers is still not there yet. For example, depending on your web app platform, you may not have an

easy mechanism to implementing this feature. For example - in Java you could following the example provided here on the OWASP site - ://www.owasp.org/index.php/HTTPOnly#Using_Java_to_Set_HTTPOnly, however this doesn't work well for the JSESSIONID as it is added by the framework. Jim Manico has been fighting the good fight to try and get Apache Tomcat developers to implement his patch to add in HTTPOnly support - ://manicode.blogspot.com/2008/08/httponly-in-tomcat-almost.html. The point is that with so many different web app development platforms, it isn't going to be easy to find support for this within every web app that you have to secure...

In the HTTPOnly Test lesson, the user can toggle the use of the HTTPOnly flag on cookie values.



The issue is that the web application did not set the HTTPOnly flag when issuing the Set-Cookie and thus client side javascript has access to the document.cookie data.

```
HTTP/1.x 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 19:00:00 EST
Set-Cookie: JSESSIONID=42B2EEB960E859CBEF77597FF9D525DF;
Path=/WebGoat
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 3914
Date: Fri, 30 Jan 2009 23:28:15 GMT
```

One of my pet peevs with the web application security space is the stigma that is associated with a WAF. Most everyone only focuses in on the negative security and blocking of attacks aspects of the typical WAF deployment and they fail to realize that WAFs are a highly specialized tool for HTTP. Depending on your circumstances, you may not ever intend to do blocking. There are many other use-cases for WAFs and how they can help, in this case as a tactical response tool to help address an underlying vulnerability In this case, we could monitor when back-end/protected web applications are handing out SessionIDs that are missing

the HTTPOnly flag. This could raise an alert that would notify the proper personnel that they should see if editing the web language code is possible to add this feature in. A rule to do this with ModSecurity would look like this -

```
# Identifies SessiondIDs without HTTPOnly flag
#
SecRule RESPONSE_HEADERS:/Set-Cookie2?/ "!(?i:\;? ?httponly;?)"
"chain,phase:3,t:none,pass,log,auditlog,msg:'AppDefect: Missing
HttpOnly Cookie Flag.'"
  SecRule MATCHED_VAR
"(?i:(j?sessionid|(php)?sessid|(asp|jserv|jw)?session[-
_]?(id)?|cf(id|token)|sid))" "t:none"
```

While this rule is pretty useful for identifying and alerting of the issue, many organizations would like to take the next step and try and fix the issue. If the web application does not have a way to add in the HTTPOnly cookie flag option internally, you can actually leverage ModSecurity+Apache for this purpose.

ModSecurity has the ability to set environmental data that Apache reads/acts upon. In this case, we can modify our previous rule slightly to use the "setenv" action and then we add an additional Apache "header" directive that will actually overwrite the data with new Set-Cookie data that includes the HTTPOnly flag -

```
# Identifies SessiondIDs without HTTPOnly flag and sets the
# "http_cookie" ENV Token for Apache to read
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}"

# Now we use the Apache Header directive to set the new data
Header set Set-Cookie "%{http_cookie}e; HTTPOnly" env=http_cookie
```

The end result of this ruleset is that ModSecurity+Apache can transparently add on the HTTPOnly cookie flag on the fly to any Set-Cookie data that you define. Thanks goes to Brian Rectanus from Breach for working with me to get the Header directive syntax correct.

One note of warning *- make sure that you understand how the web application is handling setting SessionIDs meaning if they are created server-side vs. client-side (in javascript). This rule set will work fine if the SessionIDs are generated server-side. If they are created client-side, however, this will disrupt session management.*

Hopefully the data presented here will help people who would like to have the security benefit of this flag however are running into challenges with implementing it within the app.

Cross-Site Request Forgery

The scenario for this lesson is that the user forum where the attacker is injecting the CSRF code is on the same site/domain as the CSRF targeted web application. So this means that initially blocking the CSRF injection would be feasible with the XSS rules and/or positive security for the newsgroup form submission page.

What is more challenging would be to assume that this newsgroup could possibly be hosted on a totally different website (possibly even hacker controlled). The attacker is hoping that the victim would happen to be currently logged into the target website at the same time they viewed the CSRF code page. Obviously, the likelihood of this increases significantly if the CSRF vector is hosted on the same site as the target app.

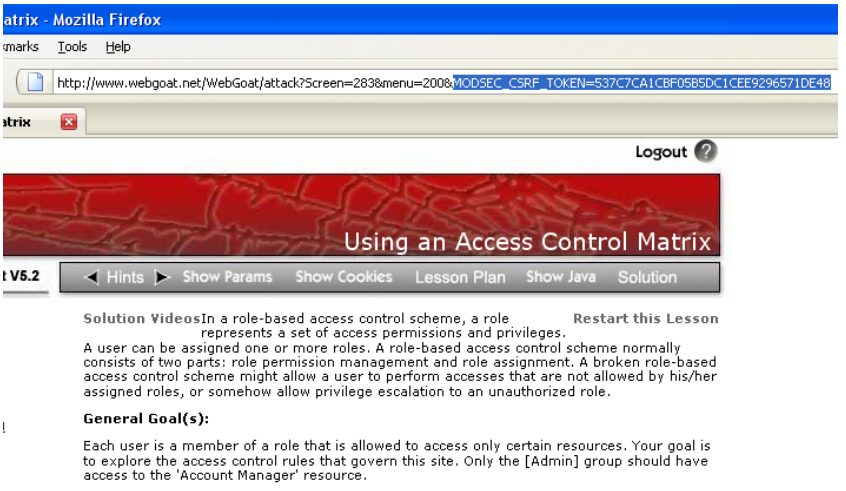
Assuming that the CSRF code is hosted on a separate site, the challenge from the web application's (and ModSecurity's) point of view is that we may *only* see the final request. It is for this reason that Anti-CSRF tokens are used. ModSecurity has a Content Injection feature that allows it to either prepend or append data to the response bodies sent to the client from the web application. We can use this feature to inject our own JavaScript code that will update the relevant links and form pages within the client's browser to add a "MODSEC_CSRF_TOKEN" value.

Instead of creating our own unique token value, we can instead utilize the applications SessionID data that it hands out to clients in Cookies. We can inject the same SessionID string into the CSRF tokens and then when subsequent requests are submitted, we can enforce that the tokens exist and that the values match that of the submitted Cookie data within the Request Header.

```
#
# CSRF Protections
# Enable Content Injection
SecContentInjection On

#
# We need to create a session collection based on the Set-Cookie data
# We will use the sessionid data later in macro expansion when we
# inject the csrf token javascript
#
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}

#
# Enforce that requests have the csrf token and that it matches the
# JSESSIONID data
#
SecRule &ARGS "@eq 1" "chain,phase:2,t:none,pass,nolog,skip:2"
 SecRule ARGS_NAMES "^MODSEC_CSRF_TOKEN$" "t:none"

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 %{request_cookies.jsessionid}"
```

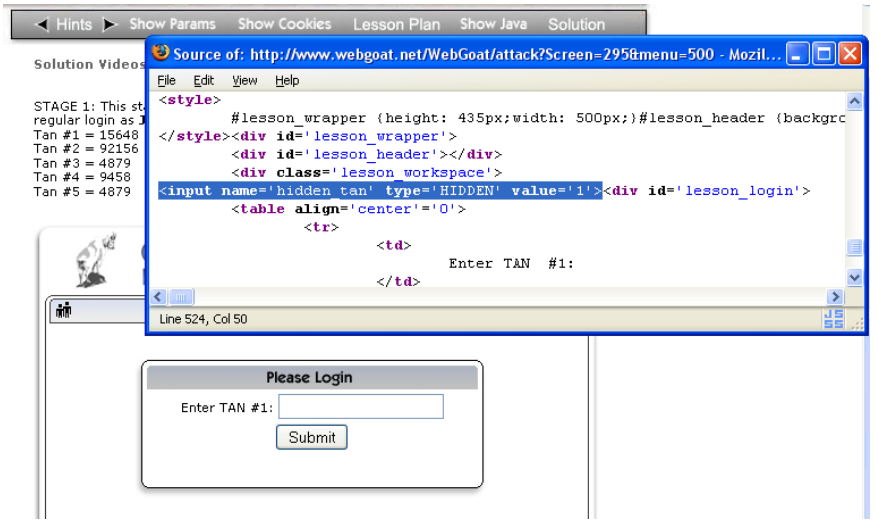
```
#
# Content Injection Section
# We inject the javascript and use macro expansion for the session.sessionid
# data
#
SecRule REQUEST_FILENAME "/WebGoat/attack"
"phase:4,t:none,nolog,pass,append:'<script language=\"JavaScript\"> \
\
var tokenName = \'MODSEC_CSRF_TOKEN\'; \
var tokenValue = \'%{session.sessionid}\'; \
\
function updateTags() { \
\
        var all = document.all ? document.all :
document.getElementsByTagName(\'*\'); \
        var len = all.length; \
\
        for(var i=0; i<len; i++) { \
                var e = all[i]; \
                \
                updateTag(e, \'src\'); \
                updateTag(e, \'href\'); \
        } \
} \
\
--CUT--
updateTags(); \
updateForms(); \
\
</script>'"
```

After these rules are applied, the WebGoat links and forms now include the MODSEC_CSRF_TOKEN data.



Authentication Flaws – Hidden Parameter Manipulation

In both lessons, the attacker alters a value of a hidden field during a login process. The lesson is that the application is keeping track of hidden data that the user can manipulate.



For this particular lesson, one method of fixing this issue is to parse the response html for hidden field data and saving it for later inspection. It is important to understand that accurate parsing of outbound html is challenging. Yes, storing this type of data in hidden fields is a bad idea; however the real problem is that the back-end should be tracking the use/reuse of this data. With that in mind, it is possible to skip parsing the response html payloads and simply focus on when the hidden_tan and parameters are first submitted. When this happens, you can easily store this data in a Session collection (tied once again to the JSESSIONID) and then check on future requests that these values are not re-submitted.

For Multi-level Login 2 – all you have to do is store the original username submitted (user2) and then ensure that it matches the hidden_user argument submitted later. If it doesn't match then you can deny.

Here are some example rules:

```
#
# Initiate the session collection based on the JSESSIONID
#
SecAction "phase:1,t:none,pass,nolog,setsid:%{REQUEST_COOKIES.JSESSIONID}, \
setuid:%{session.username}"

#
# Capture the submitted username during login for tracking/display in Mod audit
# logs
#
SecRule ARGS:'/^user/' ".*" "phase:2,t:none,pass,nolog,capture, \
setvar:session.username=%{TX.0},setuid:%{TX.0}"

#
# If this is the first time we have seen the "hidden_tan" and "tan" parameters,
# we store the data in the session collection and skip the security checks.
# We have to give the session variables unique names so that we have unique values
#
SecRule &SESSION:'/(hidden_tan|tan)/' "@eq 0" \
"chain,phase:2,t:none,pass,nolog,skip:2"
        SecRule ARGS:hidden_tan ".*" "chain,capture, \
setvar:session.hidden_tan_%{time_epoch}=%{TX.0}"
                SecRule ARGS:tan ".*"
        "capture,setvar:session.tan_%{time_epoch}=%{TX.0}

#
# If we get here, then we have saved parameter data in the session collection to
```

```
# check against the currently submitted data.  We can use the wildcarding RegEx
# capabilities of the session collection variable to allow us to inspect all
# of the saved unique parameter names.
# If any of the submmited hidden_tan/tan data matches what was already saved,
# then this is session replay attack.
#

SecRule SESSION:'/HIDDEN_TAN_*/' "@streq %{ARGS.HIDDEN_TAN}" \
"phase:2,t:none,log,auditlog,deny,msg:'Previous Hidden Tan Data Used.'"

SecRule SESSION:'/TAN_*/' "@streq %{ARGS.TAN}"  \
"phase:2,t:none,deny,log,auditlog,msg:'Previous Hidden Tan Data Used.'"

#
# Verify that the hidden_user data matches the username when they first logged in.
#
SecRule &ARGS:HIDDEN_USER "@eq 1" "chain,phase:2,t:none,deny,log, \
auditlog,msg:'Hidden User Parameter Manipulation.'"
        SecRule SESSION:'/^user/' "!@streq %{ARGS.HIDDEN_USER}"
```

Virtual patching is an interesting use-case concept, however it is best if we can "generically" attempt to address the underlying vulnerability. In the case of these two lessons, it is relatively easy to implement some virtual patches to address the vulnerability once you have knowledge about the exact attack vector parameter names. What would be great is to try and achieve the same level of protection without knowing the names of these parameters…

In a real-world application, the approach of parsing the response bodies for hidden data values, storing it and then comparing it on subsequent requests has merit. As I stated previously, it is not needed for the context of these two WebGoat lessons (as the issue is not with the first submittal of data but ones that come later), however in real applications there exists issues with altering hidden fields that go beyond replay attacks and can be a problem if the first person manipulates them. In this case, you need to have some knowledge of outbound response body data so that you can enforce it when it comes back in. Keep in mind that this technique is difficult to get right mainly due to the combination of needing a good parser along with the free text coding style of today's Web 2.0 apps.

Following is an attempt to implement a generic outbound response body inspection rule to identify/save any HIDDEN data elements and then recheck on subsequent requests for the existence of the hidden parameter value name and ensure it matches what was originally sent out.

```
SecRule SESSION:HIDDEN_ARG_NAME "!^$" "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"

SecRule RESPONSE_BODY "<input.*name=[\"']?([\w\s]*)[\"']?[\s>]type=[\"']? \
(hidden)[\"']?[\s>]value=[\"']?(\w*)[\"']\s?>" "phase:4,t:none,t:lowercase, \
pass,nolog, capture,setvar:session.hidden_arg_name=%{tx.1}, \
setvar:session.hidden_arg=%{tx.1}=%{tx.3}"
```

This has not been tested rigorously for evasions, etc… but it seems to work well for the 4.4 and 4.5 lessons. One possible limitation with these rules is that it may not work correctly if there were multiple outbound HIDDEN elements. It is for this reason that the use of of the Lua API within ModSecurity could be utilized.

The main advantage of using Lua scripts is that you can employ advanced operator functions such as for/while loops. For instance, the following section of Lua code for this same WebGoat lesson shows this type of for loop logic where it is inspecting the response body contents and extracting out hidden data for later inspection:

```
--
-- now read from file, grab hidden fields, & write back to data file
--
  local outstr -- this is used later to build output string

  -- open file first and put into string buffer
  fh = io.open("/etc/modsecurity/data/output3.txt", 'r')
--  local fh = assert(io.open(fname, "r"))
  local tbuff = fh:read("*a")
  fh:close()

  local fh2 = io.open("/etc/modsecurity/data/lessons1.data", "w+")

  for a in string.gmatch(tbuff, "<input .->") do
               t = {}

               for k, v in string.gmatch(a, "(%w+)='(.-)'") do
                     t[k]=v
               end

               -- can modify this for other input types; remove for now
--             if t.type:lower() == "hidden" then

               -- write table to file in Entry format as described in chapters
'10.1 - Data Description' & '12 - Data Files and Persistence' in the Programming in
Lua online manual

-- Format:
-- Entry{
--    name = "...",
--    type = "...",
--    value = "..."
--  }
                     if t.value == nil then -- e.g. for types such at TEXT and
PASSWORD
                            t.value = ''
                     end

                     -- for <input type='BUTTON' onclick='validate();'
value='Submit'>
                     if t.name == nil then
                            t.name = ''
                     end

                     if fh2 then
                            outstr = string.format("Entry{\n  name = \"%s\",\n
type = \"%s\",\n  value = \"%s\"\n}\n\n", t.name, t.type, t.value)
                            fh2:write(outstr)
                            m.log(9, "Luascript: now writing hidden values to
file lessons1.data")
                  end
--             end
       end
```

The main advantage that Lua would have vs. using a standard ModSecurity SecRule would be that Lua would be able to more accurately parse multiple hidden data fields within a response body.

Conclusions

Virtual Patching for web applications is an indispensable remediation process as it is able to provide protections that either wouldn't be available through traditional source code fixes or the time-to-fix length is just too long. While the ideal scenario for vulnerability remediation is to actually fix the issues within the code, ModSecurity's robust rules language and advanced features (such as Content Injection and Lua) offer an impressive platform for externally addressing web application vulnerabilities.

References

[1] Symantec Internet Threat Report, H3, 2007.

[2] Whitehat Security Web Security Statistics, March 2008

[3] Ofer Shezaf, "ModSecurity Core Rule Set": An Open Source Rule Set for Generic Detection of Attacks against Web Applications, OWASP AppSec Conference 2007

[4] Victoria Irwin, The Science of Vulnerability Filters: A Virtual Software Patch, TippingPoint Technologies, March 2004.