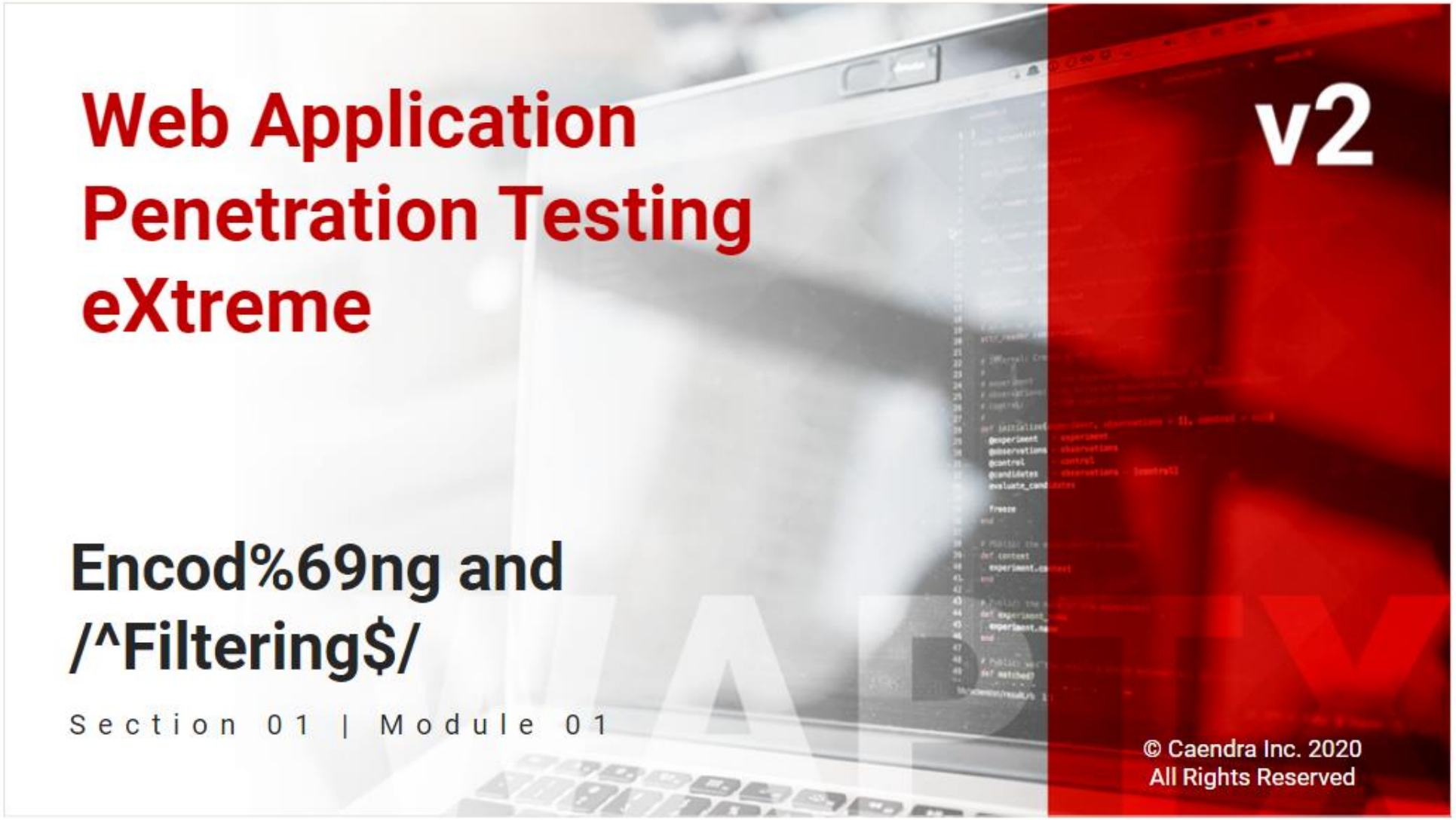
HideZeroOne
INE – Cyber Sec
www.hide01.ir



OUTLINE

Table of Contents

MODULE 01 | ENCOD%69NG AND /^FILTERING$/

1.1 Data Encoding Basics

1.2 Filtering Basics

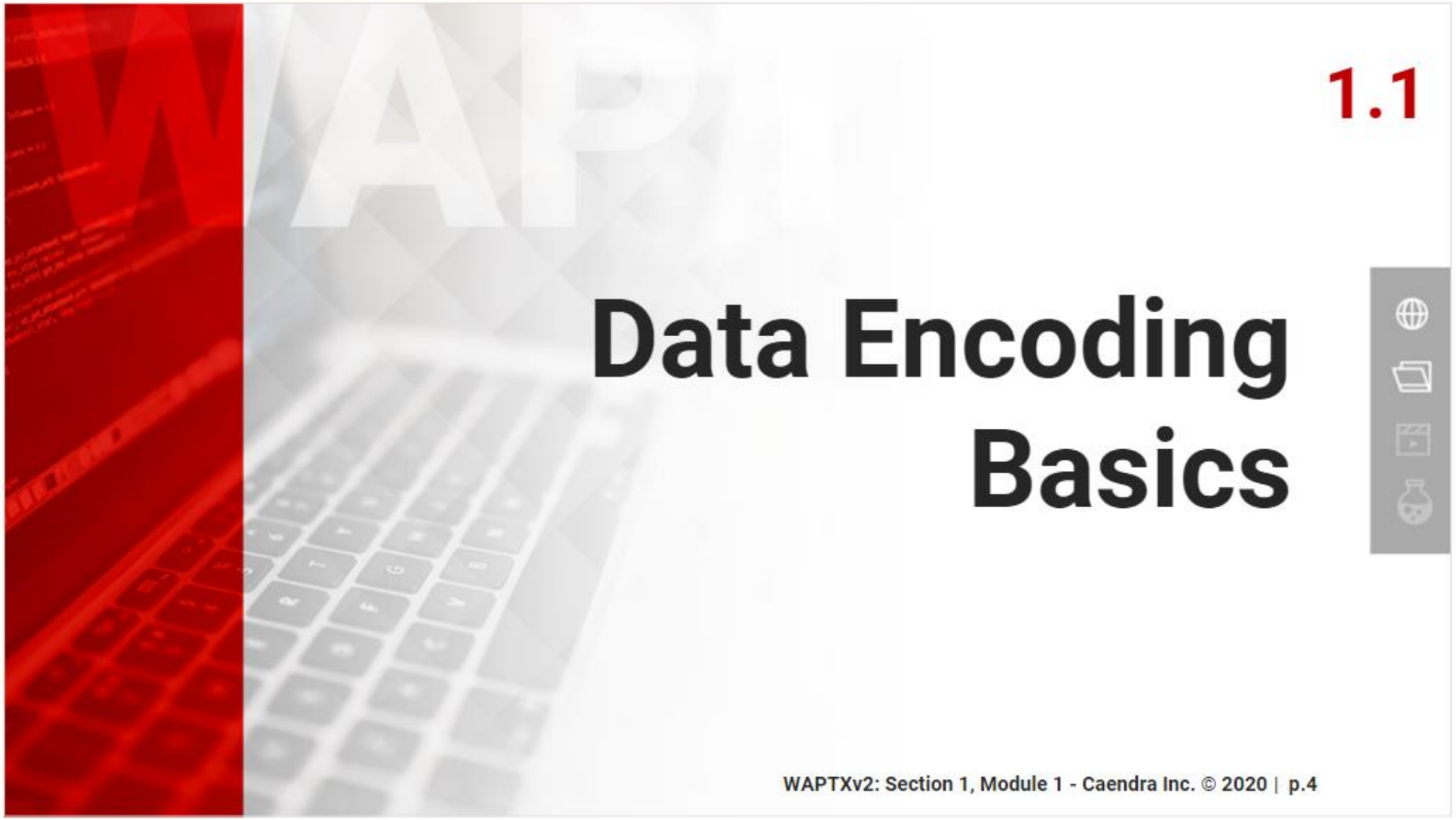
OUTLINE

Learning Objectives

In this module we will talk about different types of **data encoding**.

We will see how to recognize, encode, and decode several different formats as well as discuss **filters** and how they work.

OUTLINE



OUTLINE

1.1 Data Encoding Basics

Even though web applications have different purposes, technologies, etc., the use of data encoding is something that cannot be neglected.

From a penetration testing point of view, understanding what kind of data encoding is being used and how it works is fundamental in ensuring that the tests are performed as intended.

1.1.1 Dissecting Encoding Types

Let's briefly analyze the main types of data encoding used in web-oriented applications:

- **URL** encoding
- **HTML** encoding
- **Base** (**32**|**64**) encoding
- **Unicode** encoding

OUTLINE

1.1.1.1 URL Encoding

As stated in **RFC 3986**, URLs sent over the Internet must contain characters in the range of the US-ASCII code character set. If unsafe characters are present in a URL, encoding them is required.

The URL-encoding, or **percent-encoding,** replaces characters outside the allowed set with a "%" followed by the two hexadecimal digits representing the numeric value of the octet.

http://tools.ietf.org/html/rfc3986#section-2.1

1.1.1.1 URL Encoding

The table shown **here** is a simple character encoding chart that is useful in explaining which characters are "safe" and which characters should be encoded in URLs.

CLASSIFICATION	INCLUDED CHARACTERS	ENCODING REQUIRED?
Safe characters	Alphanumeric [0-9a-zA-Z], special characters $-_.+!*'(), and reserved characters used for their reserved purposes (e.g., question mark used to denote a query string)	NO
ASCII Control characters	Includes the ISO-8859-1 (ISO-Latin) character ranges 00-1F hex (0-31 decimal) and 7F (127 decimal.)	YES
Non-ASCII characters	Includes the entire "top half" of the ISO-Latin set 80-FF hex (128-255 decimal.)	YES
Reserved characters	$ & + , / : ; = ? @ (not including blank space)	YES*
Unsafe characters	Includes the blank/empty space and " < > # % { } | \ ^ ~ [ ] `	YES

* ***NOTE***: *Reserved characters only need encoding when not used for their defined, reserved purposes.*

http://perishablepress.com/stop-using-unsafe-characters-in-urls/

1.1.1.1 URL Encoding

Some commonly encoded characters are:

CHARACTER	PURPOSE IN URI	ENCODING
#	Separate anchors	%23
?	Separate query string	%3F
&	Separate query elements	%24
%	Indicates an encoded character	%25
/	Separate domain and directories	%2F
+	Indicates a space	%2B
<space>	Not recommended	%20 or +

1.1.1.2 HTML Encoding

Even in HTML, it is important to consider the information integrity of the URL's and ensure that user agents (browsers & co.) display data correctly.

There are two main issues to address: inform the user agent on which character encoding is going to be used in the document and preserve the real meaning of some characters that have special significance.

1.1.1.2 HTML Encoding

In order to generate potential attacks and test cases, you should not only know how this kind of encoding works, but also know how the decoding mechanism work.

OUTLINE

1.1.1.2.1 Document Character Encoding

There are several ways to instruct the user agent on which character encoding has been used in a given document.

These methods use the HTTP protocol and/or HTML directives.

1.1.1.2.1 Document Character Encoding

Define character encoding using HTTP

According to **HTTP 1.1 RFC**, documents transmitted via **HTTP** can send a charset parameter in the header to specify the character encoding of the document sent. This is the **HTTP** header: **Content-Type**.

If this header is sent, we will see something like this:

Content-Type: text/html; charset=utf-8

https://tools.ietf.org/html/rfc2616

1.1.1.2.1 Document Character Encoding

Define character encoding using HTTP

The **Content-Type** header indicates the media type of the body sent to the recipient. In the case of the **HEAD** method, it indicates the media type that would have been sent if the request had been a **GET**. If not defined, the RFC defines as the default charset the **ISO-8859-1**.

"8-bit single-byte coded graphic character sets" aka Latin 1

http://en.wikipedia.org/wiki/ISO/IEC_8859-1

1.1.1.2.1 Document Character Encoding

Define character encoding using HTTP

To make the server send out the appropriate charset information, it is possible to change the server settings or use the server-side scripting language.

Let's look at some examples in different programming languages.

1.1.1.2.1 Document Character Encoding

Define character encoding using HTTP

PHP> Uses the **header()** function to send a raw **HTTP** header:

```
header('Content-type: text/html; charset=utf-8');
```

ASP.Net> Uses the **response object**:

```
<%Response.charset="utf-8"%>
```

JSP> Uses the page directive:

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

http://www.php.net/header
http://msdn.microsoft.com/en-us/library/system.web.httpresponse

1.1.1.2.1 Document Character Encoding

Define character encoding using HTTP

It is also possible to set the character encoding using the **HTML** directive **META**. For example, this code is useful in specifying the character encoding of the current document to **UTF-8**:

```
<meta http-equiv="Content-Type" Content="text/html; charset=utf-8">
```

With **HTML5**, is also possible to write: **<meta charset="utf-8">**

1.1.1.2.2 Character References

In **HTML**, there are some special characters that can have multiple meanings. For example, the character **<** can represent the following:

- the beginning of a tag element

  Hello

- a comparison operator in JavaScript

  if (x < 7) {

- part of a text message

  "...less-than 3 would be written as < 3..."

1.1.1.2.2 Character References

To preserve the real meaning of characters, the HTML specification provides a way to escape these special characters so that they are not "confused" as HTML or other codes. The following links will take you to the respective HTML4 and HTML5 specifications.

CHR. REFERENCES

CHR. REFERENCES

http://www.w3.org/TR/1998/REC-html40-19980424/charset.html#h-5.3
http://www.w3.org/TR/html5/single-page.html#character-references

1.1.1.2.2 Character References

As the standard states, character references must start with a **U+0026 AMPERSAND** character (**&**) and following this, there are multiple ways to represent character references.

Let's see some examples.

1.1.1.2.2 Character References

We want to encode the character < *(less-than sign)*:

Character Reference	Rule	Encoded character
Named entity	& + named character references + ;	<
Numeric Decimal	& + # + D + ; D = a decimal number	<
Numeric Hexadecimal	& + #x + H + ; H = an hexadecimal number (case-insensitive)	< <

http://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references#Character_entity_references_in_HTML

1.1.1.2.2 Character References

Here we can see some interesting variations:

Character Reference	Variation	Encoded character
Numeric Decimal	No terminator (;)	&#60
	One or more zeroes before code	&#060 &#00000060
Numeric Hexadecimal	No terminator (;)	&#x3c
	One or more zeroes before code	&#0x3c &#00000x3c

1.1.1.3 Base (36|64) Encoding

Everyday, we see hexadecimal (aka Base16) numbers in network MAC addresses, X.509 certificates, Unicode characters, CSS colors, etc. This encoding scheme is frequently used in computer science.

In addition to Base16 encoding, there are other interesting **binary-to-text** encoding schemes: **Base36** and **Base64**

1.1.1.3.1 Base 36

Base 36 Encoding Scheme

Base36 is an interesting encoding scheme to play. It is the most compact, **case-insensitive**, alphanumeric numeral system using ASCII characters. In fact, the scheme's alphabet contains all digits [0-9] and Latin letters [A-Z].

The table on the next slide contains the conversions and comparisons with other well-known bases.

1.1.1.3.1 Base 36

Base 36 Encoding Scheme

binary	dec	hex	36
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	8
1001	9	9	9
1010	10	a	a
1011	11	b	b
1100	12	c	c

binary	dec	hex	36
1101	13	d	d
1110	14	e	e
1111	15	f	f
10000	16	10	g
10001	17	11	h
10010	18	12	i
10011	19	13	j
10100	20	14	k
10101	21	15	l
10110	22	16	m
10111	23	17	n
11000	24	18	o
11001	25	19	p

binary	dec	hex	36
11010	26	1a	q
11011	27	1b	r
11100	28	1c	s
11101	29	1d	t
11110	30	1e	u
11111	31	1f	v
100000	32	20	w
100001	33	21	x
100010	34	22	y
100011	35	23	z
100100	36	24	10

1.1.1.3.1 Base 36

Base 36 Encoding Scheme

For example, the number **1294870408610** in **Base10** is represented in **Base36** as **GIUSEPPE**.

Remember that it is case-insensitive. So, for example, the terms **XSS**, **xss**, **XsS**, etc… have the same representation in **Base10**, **43804**.

1.1.1.3.1 Base 36

Base 36 Encoding Scheme

Now the question is, "why should we know this encoding scheme?" The answer is easy, because Base36 is used in many real-world scenarios.

For example, **Reddit** uses it for identifying both post's and comments, while some URL shortening services like **TinyURL** use **Base36** integer as compact, alphanumeric identifiers.

http://www.reddit.com/
http://tinyurl.com/

http://tinyurl.com/jfvqr

OUTLINE

1.1.1.3.1 Base 36

Base 36 Encoding Scheme

Another example is if we want to convert **OHPE** from **Base36** to decimal, there are different implementations in many programming languages.

Let's see how to do this with **PHP** and **JavaScript**.

1.1.1.3.1 Base 36

Base 36 Encoding Scheme: PHP

PHP uses the **base_convert()** function to convert numbers:

OHPE in Base 10 is <?=base_convert("OHPE",36,10);?>

http://www.php.net/manual/en/function.base-convert.php

1.1.1.3.1 Base 36

Base 36 Encoding Scheme: JavaScript

JavaScript uses two functions:

- **(1142690).toString(36)**
- **1142690..toString(36)** **// encode**
  parseInt("ohpe",36) **// decode**

1.1.1.3.2 Base 64

Base 64 Encoding Scheme

Base64 is one of the most widespread binary-to-text encoding schemes to date. It was designed to allow binary data to be represented as **ASCII** string text.

It is an encoding scheme, not an encryption one. This is not clear to many developers who use **Base64** instead of encryption to store or transmit sensitive information.

1.1.1.3.2 Base 64

Base 64 Encoding Scheme

The alphabet of the **Base64** encoding scheme is composed of digits [**0-9**] and Latin letters, both upper and lower case [**a-zA-Z**], for a total of 62 values. To complete the character set to **64** there are the plus (**+**) and slash (**/**) characters.

Different implementations, however, may use other values for the latest two characters and the one used for padding (**=**). For a complete list look **here**.

http://en.wikipedia.org/wiki/Base64#Implementations_and_history

1.1.1.3.2 Base 64

Base 64 Encoding Scheme

To encode a message in **Base 64**, the algorithm divides the message into groups of **6 bits*** and then converts each group, with the respective **ASCII** character, following the conversion table.

That's why the allowed characters are 64 ($2^6 = 64$**).*

1.1.1.3.2 Base 64

Base 64 Encoding Scheme

Binary (dec)	Base 64	Binary (dec)	Base 64	Binary (dec)	Base 64	Binary (dec)	Base 64
000000 (0)	A	010000 (16)	Q	100000 (32)	g	110000 (48)	w
000001 (1)	B	010001 (17)	R	100001 (33)	h	110001 (49)	x
000010 (2)	C	010010 (18)	S	100010 (34)	i	110010 (50)	y
000011 (3)	D	010011 (19)	T	100011 (35)	j	110011 (51)	z
000100 (4)	E	010100 (20)	U	100100 (36)	k	110100 (52)	0
000101 (5)	F	010101 (21)	V	100101 (37)	l	110101 (53)	1
000110 (6)	G	010110 (22)	W	100110 (38)	m	110110 (54)	2
000111 (7)	H	010111 (23)	X	100111 (39)	n	110111 (55)	3
001000 (8)	I	011000 (24)	Y	101000 (40)	o	111000 (56)	4
001001 (9)	J	011001 (25)	Z	101001 (41)	p	111001 (57)	5
001010 (10)	K	011010 (26)	a	101010 (42)	q	111010 (58)	6
001011 (11)	L	011011 (27)	b	101011 (43)	r	111011 (59)	7
001100 (12)	M	011100 (28)	c	101100 (44)	s	111100 (60)	8
001101 (13)	N	011101 (29)	d	101101 (45)	t	111101 (61)	9
001110 (14)	O	011110 (30)	e	101110 (46)	u	111110 (62)	+
001111 (15)	P	011111 (31)	f	101111 (47)	v	111111 (63)	/

1.1.1.3.2 Base 64

Base 64 Encoding Scheme

If the total number of bits is not a multiple of 6, then null bits need to be added until the total is both a multiple of 6 and the result length a multiple of 4.

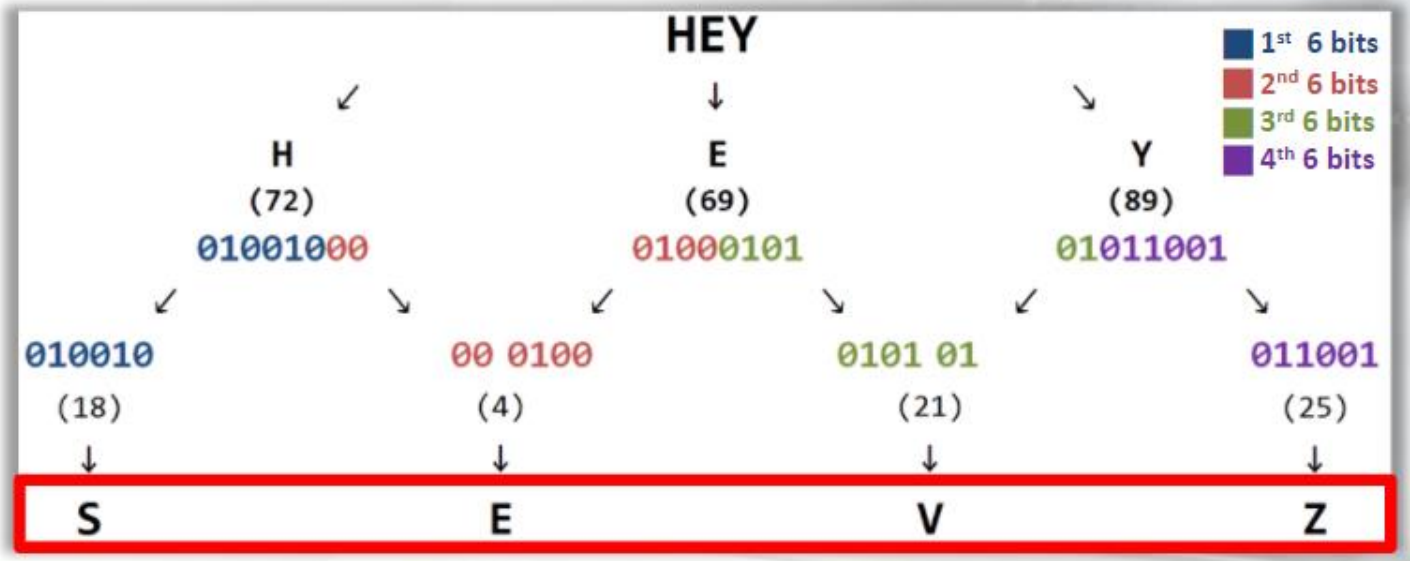
Then, if the **latest** group is 'null' (**000000**), the respective encoding value is **=** but, if the trailing "null groups" are two they will be encoded as **==**.

Let's check out some examples.

1.1.1.3.2 Base 64

Base 64 Encoding Scheme

To encode the term "**HEY**" we have:

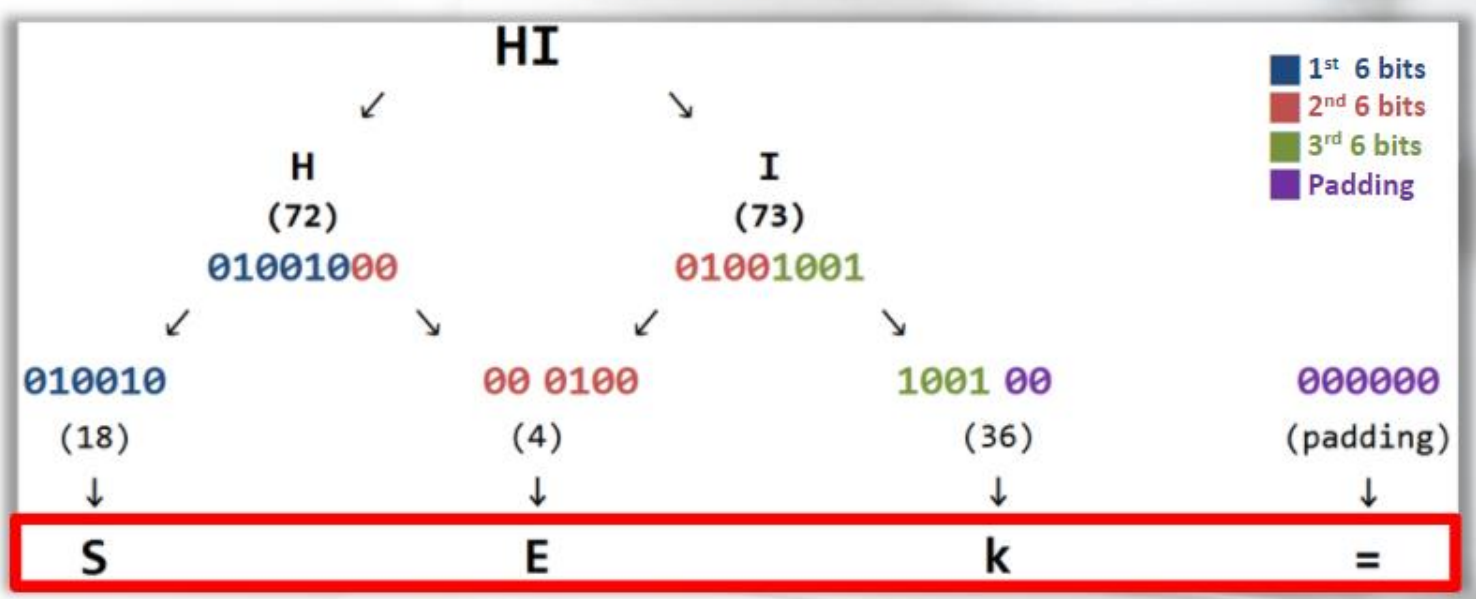


OUTLINE

1.1.1.3.2 Base 64

Base 64 Encoding Scheme

To encode the term "**HI**" we have:

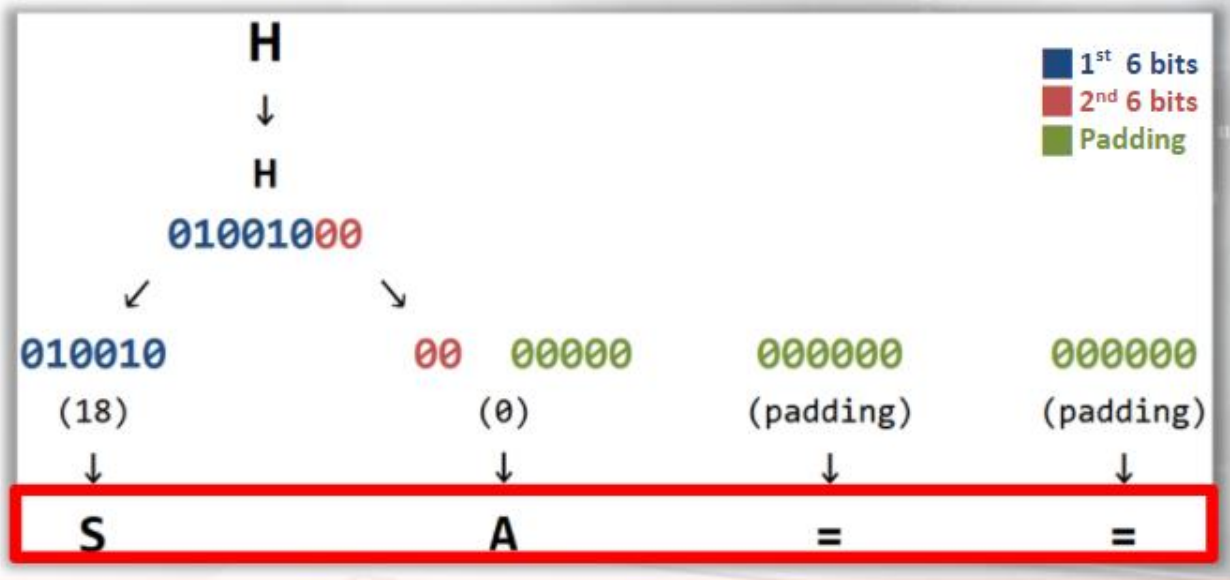


OUTLINE

1.1.1.3.2 Base 64

Base 64 Encoding Scheme

To encode the char "**H**" we have:



OUTLINE

1.1.1.3.2 Base 64

Base 64 Encoding Scheme

Naturally, due to its popularity, there are many encoding / decoding implementations of **Base64** in a variety of different programming languages.

Let's see some of them.

OUTLINE

1.1.1.3.2 Base 64

Base 64 Encoding Scheme: PHP

PHP uses **base64_encode** and **base64_decode** functions to encode/decode data based on **MIME Base 64** implementation:

<?=base64_encode('encode this string')?> //Encode

<?=base64_decode('ZW5jb2RlIHRoaXMgc3RyaW5n')?> //Decode

https://www.php.net/base64_encode
https://www.php.net/base64_decode

1.1.1.3.2 Base 64

Base 64 Encoding Scheme: JavaScript

Many browsers can handle **Base64** natively through functions **btoa** and **atob**:

window.btoa('encode this string'); //Encode

window.atob('ZW5jb2RlIHRoaXMgc3RyaW5n'); //Decode

https://developer.mozilla.org/en-US/docs/Web/API/Window.btoa

OUTLINE

1.1.1.3.2 Base 64

Base 64 Encoding Scheme

It is important to notice that if we want to handle **Unicode** strings, then we should encode them before using **Base64** functions. For example, in **JavaScript** this is possible as follows:

*The escapes and encodings are required to avoid exceptions with characters out of range. Learn more **here**.*

```
function utf8_to_b64( str ) {
    return window.btoa(encodeURIComponent( escape( str )));
}

function b64_to_utf8( str ) {
    return unescape(decodeURIComponent(window.atob( str )));
}

// Usage:
utf8_to_b64('✓ à la mode'); // JTI1dTI3MTMlMjUyMCUyNUUwJTI1MjBsYSUyNTIwbW9kZQ==
b64_to_utf8('JTI1dTI3MTMlMjUyMCUyNUUwJTI1MjBsYSUyNTIwbW9kZQ=='); // "✓ à la mode"

utf8_to_b64('I \u2661 Unicode!'); // SSUyNTIwJTI1dTI2NjElMjUyMFVuaWNvZGUlMjUyMQ==
b64_to_utf8('SSUyNTIwJTI1dTI2NjElMjUyMFVuaWNvZGUlMjUyMQ=='); // "I ♡ Unicode!"
```

https://developer.mozilla.org/en-US/docs/Web/API/Window.btoa

OUTLINE

1.1.1.4 Unicode Encoding

Unicode (aka **ISO/IEC 10646** Universal Character Set) is the character encoding standard created to enable people around the world to use computers in any language. It supports all the world's writing systems.

Because **Unicode** contains such a large number of characters, glyphs, numbers, etc., from a security point of view, it is fascinating because incorrect usage can expose web applications to possible security attacks. One such example, is that it can be useful to bypass filters.

1.1.1.4 Unicode Encoding

We are not going to cover the Unicode specifics, but if you want to have a better background on the argument, character sets and related topics, the following link is a great starting point:

http://www.joelonsoftware.com/articles/Unicode.html

1.1.1.4 Unicode Encoding

There are three ways to map Unicode character points:

- **UTF-8**
- **UTF-16**
- **UTF-32**

UTF means **U**nicode **T**ransformation **F**ormat and the trailing number indicates the number of bits to represent code points.

1.1.1.4 Unicode Encoding

Thus, each UTF has a different representation and it is important to understand how to handle these in our tests. The following table shows a sample message encoded in the three different UTF formats.

CHARACTER	REPRESENTATION			
	Unicode	UTF-8 code point	UTF-16 code point	UTF-32 code point
I	U+0049	49	0049	00000049
♥	U+2665	E2 99 A5	2665	00002665
🍻	U+1F37B	F0 9F 8D BB	D83C DF7B	0001F37B

1.1.1.4 Unicode Encoding

It is also useful to know how Unicode characters are handled through different implementations like URLs, HTML, JavaScript, etc. We can see some of them below.

CHARACTER	REPRESENTATION					
	Unicode Code Point	URL-Encoding (UTF-8)	HTML Entity Named	HTML Entity Decimal	HTML Entity Hexadecimal	JavaScript, JSON, Java (UTF-16)
I	U+0049 (hex 49, dec 73)	%49	-	I	I	\u0049
♥	U+2665 (hex 2665, dec 9829)	%E2%99%A5	♥	♥	♥	\u2665
🥂	U+1F37B (hex 1F37B, dec 127867)	%F0%9F%8D%BB	-	🍻	🍻	\uD83C\uDF7B

1.1.1.4 Unicode Encoding

Besides the representation of Unicode characters in multiple encoding types, another interesting aspect is the interpretation that humans and different implementations give to some characters.

OUTLINE

1.1.1.4 Unicode Encoding

Homoglyph | Visual Spoofing

*"In typography, a **Homoglyph** is one or two or more characters, or glyphs, with shapes that either appear identical or cannot be differentiated by quick visual inspection."* [Wikipedia]

An additional classification is:

HOMOGRAPH > a **word** that looks the same as another word

HOMOGLIPH > a look-alike **character** used to create homographs

1.1.1.4 Unicode Encoding

Homoglyph | Visual Spoofing

One of the possible attacks with Unicode is called:

Visual Spoofing

1.1.1.4 Unicode Encoding

Homoglyph | Visual Spoofing

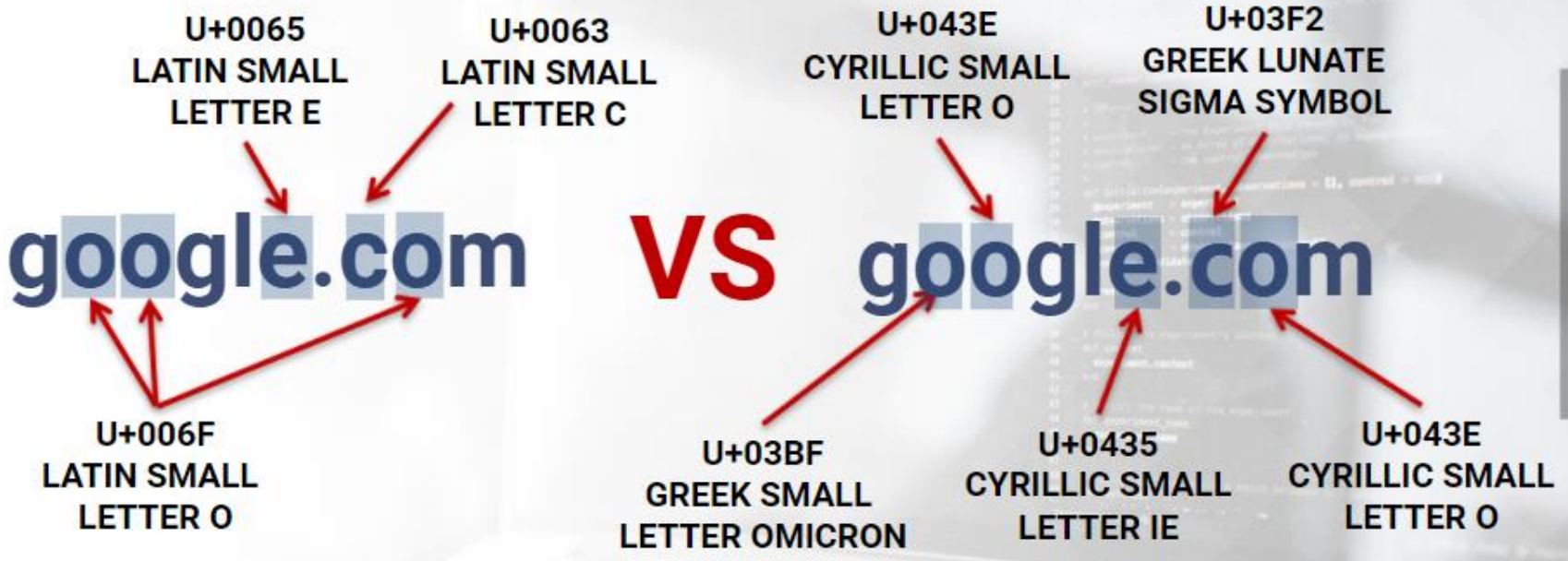
If we analyze the characters code points of the string, the differences between the **o** and the **o** are evident, but for a human this is not so obvious.

These kind of characters, also known as a **confusable**, received special attention from the Unicode Consortium (**TR39**). So much so, that they have provided a **utility**, whereby given an input string you can see the combinations that are confusable with it.

http://www.unicode.org/reports/tr39/
http://unicode.org/cldr/utility/confusables.jsp

1.1.1.4 Unicode Encoding

Homoglyph | Visual Spoofing - Example: google.com



OUTLINE

1.1.1.4 Unicode Encoding

Homoglyph | Visual Spoofing

To speed the homographs generation, rather than searching for look-alike characters in Unicode, there is an interesting application made by Adrian “Irongeek” Crenshaw:

Homoglyph Attack Generator

http://www.irongeek.com/homoglyph-attack-generator.php

1.1.1.4 Unicode Encoding

Homoglyph | Visual Spoofing

This attack generator tool is part of a really interesting paper where the author explains the abuse of Unicode characters in order to obfuscate phishing attacks through the use of Homoglyph and Punycode.

Punycode and Homoglyph Attacks to Obfuscate URLs for Phishing

http://www.irongeek.com/i.php?page=security/out-of-character-use-of-punycode-and-homoglyph-attacks-to-obfuscate-urls-for-phishing

1.1.1.4 Unicode Encoding

Computer Interpretations

Another interesting aspect is related to string and character "evolutions," which occur during normal software processes transformations.

An example of this is upper and lower casing transformations, which are described in the upcoming slides.

OUTLINE

1.1.1.4 Unicode Encoding

Computer Interpretations - Example: Censured Feedback

In a feedback page, the application layer performs a censorship check before storing data in a DB.

There is an input filter that blocks the term **EVIL**, then transform the string to lowercase and store it in DB.

1.1.1.4 Unicode Encoding

Computer Interpretations - Example: Censured Feedback

The input flow could be as follow:

- A user sends the following message:

```
Evİl intent, as usual!
```

- The filter checks for evil strings, but without success.

```
Evİl != evil
```

U+0130 (İ)
LATIN CAPITAL LETTER I
WITH DOT ABOVE

OUTLINE

1.1.1.4 Unicode Encoding

Computer Interpretations - Example: Censured Feedback

- The casing operation is performed [to lowercase]:

```
evil intent, as usual!
```

U+0130(İ) to lowercase is
U+0069
LATIN SMALL LETTER I

- **CENSURED BYPASSED**

1.1.1.4 Unicode Encoding

Computer Interpretations - Example: Censured Feedback

This happened because a casing operation is performed in the application flow **AFTER** a security check.

Of course, it also works by upper casing characters like this:

ſ to upper case is **S**

U+017F
LATIN SMALL LETTER LONG S

U+0053
LATIN CAPITAL LETTER S

OUTLINE

1.1.1.4 Unicode Encoding

Computer Interpretations- Example: Censured Feedback

It turns out, that this type of vulnerable implementation may allow an attacker to bypass filters. For example, they can bypass anti cross-site scripting and SQL injection filters and so forth.

These are things that ~~never~~ happen in real world!
Check them out here:

Creative usernames and Spotify account hijacking

http://labs.spotify.com/2013/06/18/creative-usernames/

1.1.1.4 Unicode Encoding

Computer Interpretations

There are other ways in which characters and strings can be transformed by software processes, such as normalization, canonicalization, best fit mapping, etc.

These are brilliantly summarized and explained by Chris Weber in his:

Unicode Security Guide

http://websec.github.io/unicode-security-guide/

1.1.1.4 Unicode Encoding

Computer Interpretations: Mixed Examples

Normalization:

ⓓⓡⓞⓟ ⓣⓐⓑⓛⓔ **becomes** `drop table`

Canonicalization:

‹ (U+2039)
﹤ (U+FE64)
＜ (U+ff1c)

} **becomes** < (U+003C)

1.1.1.4 Unicode Encoding

If you want to play a bit with Unicode characters, you can visit **Unicode utilities** to get information about a character or search confusable characters.

In addition to this tool, there are other interesting resources such as **codepoints.net**, **txtn.us** and **Unicode Text Converter**.

http://unicode.org/cldr/utility/
http://codepoints.net/
http://txtn.us/
http://www.panix.com/~eli/unicode/convert.cgi

1.1.2 Multiple (De|En) Codings

As we have just seen in the previous slides, data encoding is fundamental for communication in web applications.

Sometimes, encoding is required by the chosen channel. Other times it is used intentionally by developers but, sometimes it is used **multiple times**, intentionally and not. It is also common to **ab**use multiple encodings to bypass security measures

1.1.2 Multiple (De|En) Codings

Example: URL-Encoding > URL

A simple scenario could be a URL sent over URL, like the common URL redirects we see daily for surfing web sites:

```
http://mywebsite/login.php?redirectURL=FORW-URL?is_ok=yes
```

In this case, the forwarding URL will be URL-encoded in order to respect the URL rules:

```
http://mywebsite/login.php?redirectURL=FORW-URL%3Fis_ok%3Dyes
```

1.1.2 Multiple (De|En) Codings

Example: URL-Encoding > URL

Of course, even if a parameter sent is not a URL, encoding is still required:

`http://mywebsite/login.php?param=I` ❤ 🤯

In this case, the parameter contains Unicode characters, hence the URL-encoding will be:

`http://mywebsite/login.php?param=I%E2%99%A5%F0%9F%8D%BB`

1.1.2 Multiple (De|En) Codings

Example: URL-Encoding > URL

Multiple encodings may also occur if the parameter sent is previously encoded, like the following:

http://mywebsite/login.php?param=Rk9SVy1VUkw/Y2F0PWNsb3ducw==

In this case, we have a Base64 data encoded to send over URL; thus, the result will be:

http://mywebsite/login.php?param=Rk9SVy1VUkw%2FY2F0PWNsb3ducw%3D%3D

1.1.2 Multiple (De|En) Codings

Example: URL-Encoding > URL

Sometimes, a simple parameter can be a structured parameter. For example, the following cookie value:

```
SESSION = dXNlcm5hbWU6Y2xvd247cGFzc3dvcmQ6dGhlQ2xvd24h
```

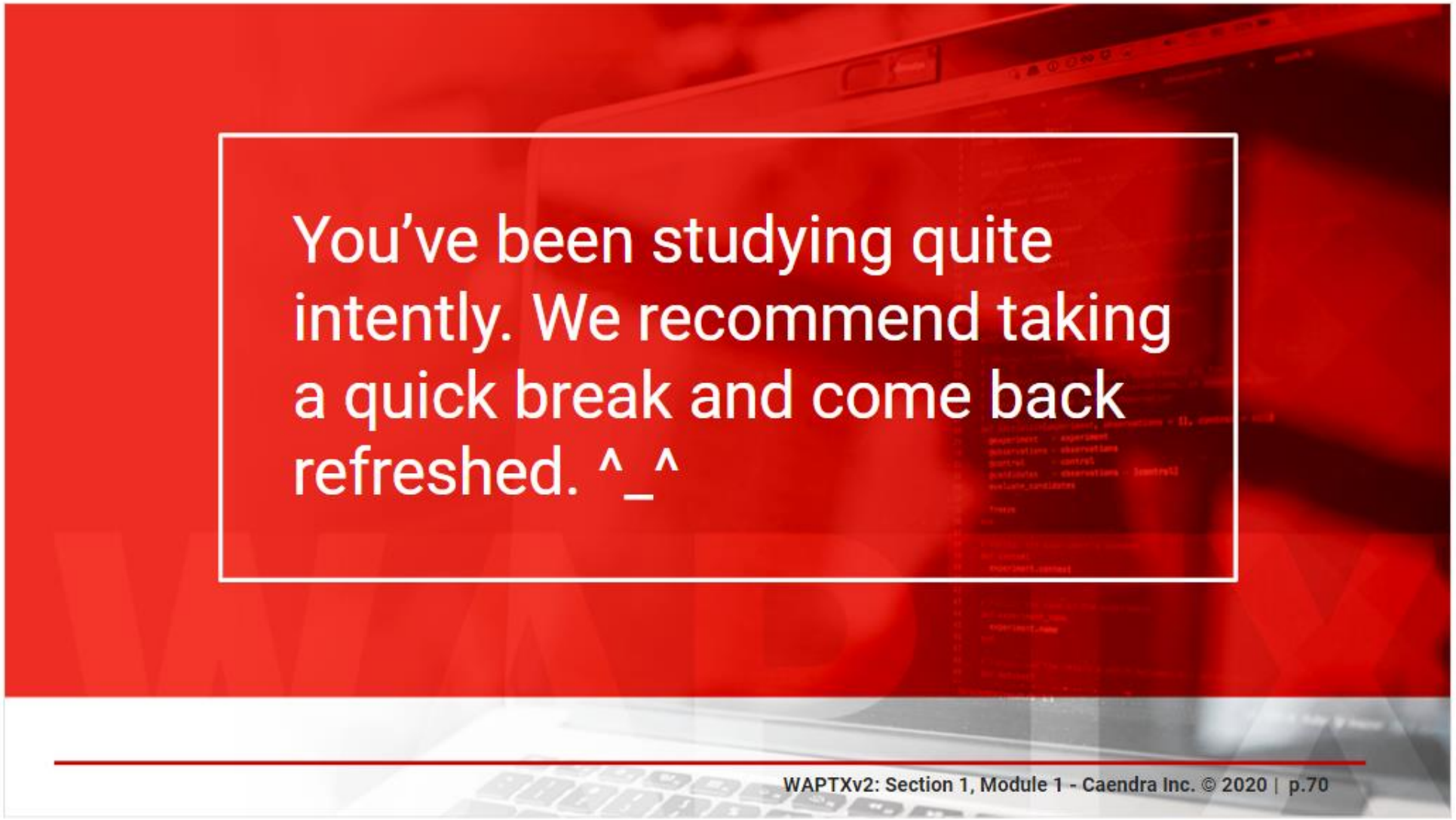
...may appear like a 'random' value used to identify the user session, but it decodes to:

```
SESSION = username:clown;password:theClown!
```

1.1.2 Multiple (De|En) Codings

It turns out that we can construct several examples based on multiple encoding/decoding scenarios, but the topic here is to understand that identifying different data encoding types is an important skill that may help you to detect and exploit multiple scenarios.

All in all, in order to respect the application requirements and properly test a web application, we must detect and consider multiple encoding and decoding operations.



OUTLINE

OUTLINE

1.2 Filtering Basics

A common, yet often recommended, best practice to protect web applications against malicious attacks is the use of specific **input filtering** and **output encoding** controls.

These kinds of controls may range from naive blacklists to experienced and highly restrictive whitelists. What about in the real world? We are somewhere in the middle!

1.2 Filtering Basics

Controls can be implemented at different layers in a web application. They can a be represented as either **libraries and APIs** (by naive developers) or, in the best case, by internal specialists or external organizations, like **ESAPI by OWASP**.

Security controls are also inside most common browsers.

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

1.2 Filtering Basics

Sometimes, because of a multitude of complications and standards, it is not possible, or it is too problematic to implement internal solutions. These solutions may be in the form of libraries or APIs; however, the key is in the adoption of external solutions.

Generally, these solutions fall into the **IDS** and **IPS** world, but for web applications, the most chosen are the **W**eb **A**pplication **F**irewall (**WAFs**).

1.2 Filtering Basics

Even with WAFs, it is possible to choose between several implementations.

These range from not only commercial and very expensive, but also free and open source solutions like the well known **ModSecurity**.

http://www.modsecurity.org/

1.2 Filtering Basics

The instructions on what a WAF, or generally, what a filter must block/allow are defined as **rules** (also referred as **filters**).

Before we analyze different filter implementations, let's see the *de facto* standard used to write rules.

1.2 Filtering Basics

Regular **E**xpressions (**RE** or **RegEx**) represents the official way used to define the filter rules. Mastering RegEx is fundamental to understand how to bypass filters because RE are extremely powerful!

The upcoming slides contain a brief introduction to the subject.

1.2.1 Regular Expressions

NOTE:

This is a brief introduction to Regular Expressions. For a comprehensive introduction and much more, in the references you will find some interesting resources.

1.2.1 Regular Expressions

A regular expression is a special sequence of characters used for describing a **search pattern**.

For the sake of clarity, in the next slides we will see the following notation:

- regular expression > `regex`
- pattern matched > `match`

1.2.1 Regular Expressions

Many programming languages, text processors, etc., support regular expressions. The implementation system of regex functionality is often called **regular expression engine**. Basically, a regex "engine" tries to match the pattern (regex) to the given string.

There are two main types of regex **engines**: DFA and NFA, also referred to as **text-directed** and **regex-directed** engines. The key difference between the two engines is a notational convenience in the construction of the FA (Finite Automaton).

http://en.wikipedia.org/wiki/Deterministic_finite_automaton
http://en.wikipedia.org/wiki/Nondeterministic_finite_automaton

1.2.1 Regular Expressions

The DFA engine is faster than NFA because of its deterministic approach, but it does not support useful features like **lazy quantifiers** and **backreferences**. Additionally, NFA works the way humans tend to do: "regex-directed". It's no surprise that the NFA engine is more popular.

Here is a table of notable programs that use DFA or NFA engines:

ENGINE	PROGRAM
DFA	awk, egrep, MySQL, Procmail
NFA	.NET languages, Java, Perl, PHP, Python, Ruby, PCRE library, vi, grep, less, more

1.2.1 Regular Expressions

The syntax and behavior of a particular engine is called a **regular expression flavor**. Since the engine is a piece of software, there are different versions and of course they are not fully compatible with each other.

Thus, expect to find multiple flavor based on the library you are using/testing!

1.2.1 Regular Expressions

For a complete comparison list of regular expression engines/flavors visit the following links:

Comparison of regular expression engines

- http://en.wikipedia.org/wiki/Comparison_of_regular_expression_engines

Regular Expression Flavor Comparison

- http://www.regular-expressions.info/refflavors.html

1.2.1 Regular Expressions

Since regular expression is a "symbolic language", to master this tool we must know the symbols of the language. They are few and have specific meanings.

Let's look at some tables that collect these symbols and their meanings.

1.2.1.1 Metacharacters

Char	Name	Meaning
^	Caret	The position at the START of the line.
$	Dollar	The position at the END of the line.
()	Opening/Closing parenthesis	Start/close a characters group.
[]	Opening/Closing square bracket	Start/close a characters class.
?	Question mark	One or zero (optional) of the immediately-preceding item (char or group).
+	Plus	One or more of the immediately-preceding item (char or group).
*	Star or asterisk	Any number, including none, of the immediately-preceding item (char or group).
.	Period or dot	Shorthand for a character class that matches any character.
\	Backslash	Escape special characters.
|	Vertical bar or pipe	It means OR. Combines multiple expressions in one that matches any of single ones.
{}	Opening/Closing curly brace	Start/close repetitions of a characters class.

1.2.1.2 Shorthand Character Classes

Since there are some character classes frequently used, there are also related shorthand classes that are useful to decrease the size and increase the readability of a regex.

The table on the next slide has the most common shorthand classes.

1.2.1.2 Shorthand Character Classes

SHORTHAND	NAME	MEANING
^	Caret	If at the beginning of the character class, it means to reverse the matching for the class.
\d	Digit	Matches any digit character. The same as [0-9]
\D	Non-digit	The complement of \d. The same as [^\d]
\w	Part-of-word character	Matches any alphanumeric character or an underscore. The same as [a-zA-z0-9_] In some flavors the underscore is omitted.
\W	Non-word character	The complement of \w. The same as [^\w]
\s	Whitespace character	Matches any whitespace character. The same as [\f\n\r\t\v]
\S	Non-whitespace character	The complement of \s. The same as [^\s]

1.2.1.3 Non-Printing Characters

Oftentimes, to evade bad filters and obfuscate the payload it is common to use **non-printing characters**. These are control characters used mainly to control the format of displayed/printed information.

The most used characters are represented in the table on the next slide.

http://en.wikipedia.org/wiki/Control_character

1.2.1.3 Non-Printing Characters

SHORTHAND	NAME (Symbol, Unicode)	MEANING
\0	NUL (␀ U+0000)	NUL Byte, in many programming languages marks the end of a string.
\b	Backspace (␈ U+0008)	Within a character class represent the backspace character, while outside \b matches a word boundary.
\t	Horizontal tab (␉ U+0009)	Generated by the Tab key on a standard keyboard.
\n	Line feed (␊ U+000A)	New line.
\v	Vertical tab (␋ U+000B)	Vertical tabulation.
\f	Form feed (␌ U+000C)	Form feed.
\r	Carriage return (␍ U+000D)	In HTTP, the \r\n sequence is used as the end-of-line marker.
\e	Escape (␛ U+001B)	Escape character (Only for GNU Compiler Collection).

1.2.1.4 Unicode

Regular expression flavors that work with Unicode use specific meta-sequences to match code points.

The sequence is `\u`*`code-point`*, where *`code-point`* is the hexadecimal number of the character to match. There are regex flavors like PCRE that do not support the former notation, but use an alternative sequence `\x{`*`code-point`*`}` in its place.

1.2.1.4 Unicode

Match Unicode Code Point

For example, the regex \u2603 matches the snowman character  in .NET, Java, JavaScript and Python.

If we want to match the same character to the PCRE library in Apache and PHP, we must use the other notation:
\x{2603}

1.2.1.4 Unicode

Instead of matching a single Unicode code point, it is possible to match if a character has a specific "quality". This is possible with the use of **Unicode properties**.

Unicode defines for each character, properties or qualities, such as: "*is this character uppercase*" or "*is this character a punctuation*" and so on, in order to match these qualities with regex exists specific meta-sequences.

1.2.1.4 Unicode

The characters that have a specific quality are matched with the meta-sequence `\p{quality-id}`. To match the characters that do not have the quality, the meta-sequence is `\P{quality-id}`.

Some general Unicode character qualities are reported in the table on the next slide.

1.2.1.4 Unicode

CHARACTER QUALITY	DESCRIPTION
\p{L} or **\p{Letter}**	All the letters, from any language.
\p{Ll} or **\p{Lowercase_Letter}**	Lowercase letters that have the respective uppercase quality.
\p{Z} or **\p{Separator}**	Characters used to separate, but without visual representation.
\p{S} or **\p{Symbol}**	Currency symbols, math symbols, etc...
\p{N} or **\p{Number}**	All the numeric characters.
\p{Nd} or **\p{Decimal_Digit_Number}**	Numbers from zero to nine in multiple scripts except Chinese, Japanese, and Korean.
\p{P} or **\p{Punctuation}**	All the punctuation characters.

1.2.1.4 Unicode

Match Unicode Category

For example, to match the lowercase characters in this string:

Ğ **ī ű** Ŝ **ê p** P Ẽ

the regex is `\p{Ll}` and the characters matched are

ī ű ê p

1.2.1.4 Unicode

Match Unicode Category

To match the string with all the case variations (lower, upper and title), this regex does the job:

```
[\p{Ll}\p{Lu}\p{Lt}]
```

As a shorthand, some regex flavors implement this solution:

```
\p{L&}
```

1.2.2 Web Application Firewall

We have briefly seen Regular Expressions, which is the main method used to define the rules of how a WAF should behave. They define which input is good or bad for the web application and respectively what the WAF should allow or block.

The meaning given to the rules defines the mode by which a WAF should behave. It can be whitelisted or blacklisted. Basically, a WAF in whitelisting mode allows only what is explicitly defined in the rules; however, in contrast, blacklisting mode allows anything except what is explicitly denied in the rules.

OUTLINE

1.2.2 Web Application Firewall

It turns out that whitelisting mode is the best solution to protect a web application; on the other hand, to customize the rules is not an easy task. This requires a deep knowledge of the application to protect and, obviously, of the WAF solution.

Furthermore, the whitelisting mode is prone to false positives, which is the reason it is very common to find WAFs deployed in blacklisting mode rather than whitelisting mode.

1.2.2 Web Application Firewall

The blacklisting mode is a collection of well-known attacks. WAF producers put together a list of rules to protect a web application against various attack vectors that are used to exploit the most common vulnerabilities.

OUTLINE

1.2.2 Web Application Firewall

The main problem with this kind of approach is that there are multiple ways to reach the same goal; therefore, every small change of the attack payload must be added to the blacklist, otherwise you have a WAF bypass!

Predicting or keeping track of each payload tweak is very hard, that's why we frequently read the expression:

All WAFs can be bypassed!

1.2.2 Web Application Firewall

Over the years, security researchers have discovered several "*alternative vectors*" (i.e. **WAF bypasses)** and a lot of well-known names were involved.

The following examples are just simple rules to follow to deceive ingenuous WAFs.

1.2.2.1 Simple Rules to Bypass WAFs

Cross-Site Scripting

Instead of using:

- alert('xss')
- alert(1)

DO

The best choice is:

- `prompt('xss')`
- `prompt(8)`
- `confirm('xss')`
- `confirm(8)`
- `alert(/xss/.source)`
- `window[/alert/.source](8)`

1.2.2.1 Simple Rules to Bypass WAFs

Cross-Site Scripting

Instead of using:

- alert(document.cookie)

The best choice is:

- with(document)alert(cookie)
- alert(document['cookie'])
- alert(document[/cookie/.source])
- alert(document[/coo/.source+/kie/.source])

DO

OUTLINE

1.2.2.1 Simple Rules to Bypass WAFs

Cross-Site Scripting

Instead of using:

- <img src=x onerror=alert(1);>
- javascript:alert(document.cookie)

The best choice is:

- <svg/onload=alert(1)>
- <video src=x onerror=alert(1);>
- <audio src=x onerror=alert(1);>
- data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4=

DO

OUTLINE

1.2.2.1 Simple Rules to Bypass WAFs

Blind SQL Injection

Instead of using:

- ' or 1=1

The best choice is:

- ' or 6=6
- ' or 0x47=0x47
- or char(32)=' '
- or 6 is not null

DO

1.2.2.1 Simple Rules to Bypass WAFs

SQL Injection

Instead of using:

▼ **UNION SELECT**

The best choice is:

▲ **UNION ALL SELECT**

OUTLINE

1.2.2.1 Simple Rules to Bypass WAFs

Directory Traversal

Instead of using:

- /etc/passwd

The best choice is:

- /too/../etc/far/../passwd
- /etc//passwd
- /etc/ignore/../passwd
- /etc/passwd.....

DO

1.2.2.1 Simple Rules to Bypass WAFs

Web Shell

Instead of using:

- c99.php
- r57.php
- shell.aspx
- cmd.jsp
- CmdAsp.asp

DO

The best choice is:

- augh.php

OUTLINE

1.2.2.2 WAF Detection and Fingerprinting

Usually WAFs work in **passive** mode, **reactive** mode, or sometimes both. It depends on the period at which they are installed. For example, once deployed, they can be in passive mode, reducing the number of false positives and avoiding blocking the application; however, once in production, most are reactive.

Before testing a web application, it is extremely useful to know if there is a WAF on the other side and what kind it is. WAF systems leave several footprints of their presence, which allow us to detect which WAF is in place. Let's check out some techniques.

1.2.2.2 WAF Detection and Fingerprinting

Cookie Values

Some WAF systems reveal their presence through cookies. They release their own cookie during the HTTP communications.

1.2.2.2 WAF Detection and Fingerprinting

Cookie Values:

Citrix Netscaler uses some different cookies in the HTTP responses like `ns_af` or `citrix_ns_id` or `NSC_`

F5 BIG-IP ASM (Application Security Manager) uses cookies starting with `TS` and followed with a string that respect the following regex:

```
^TS[a-zA-Z0-9]{3,6}
```

1.2.2.2 WAF Detection and Fingerprinting

Cookie Values:

Barracuda uses two cookies `barra_counter_session` and `BNI__BARRACUDA_LB_COOKIE`.

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 8543
Content-Type: text/html
Expires: Tue, 08 Apr 2014 08:56:45 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Date: Tue, 08 Apr 2014 08:57:44 GMT
Set-Cookie: BNI__BARRACUDA_LB_COOKIE=0000000000000000000000000c400000a0000bb20; Path=/
```

1.2.2.2 WAF Detection and Fingerprinting

Header Rewrite

Some WAFs rewrite the HTTP headers. Usually these modify the `Server` header to deceive the attackers.

For example, they either rewrite the header if the request is malicious or, depending on the malicious request, remove the HTTP `Server` header in the response.

1.2.2.2 WAF Detection and Fingerprinting

Header Rewrite: Example – Rewrite Server Header

HTTP response for **non-hostile** request

```
HTTP/1.1 200 OK
Date: Mon, 7 Apr 2014 10:10:50 GMT
Server: Apache (Unix)
Content-Type: text/html
Content-Length: 2506
```

HTTP response for **hostile** request

```
HTTP/1.1 404 Not Found
Date: Mon, 7 Apr 2014 10:11:06 GMT
Server: Netscape-Enterprise/6.1
Content-Type: text/html;
Content-Length: 158
```

1.2.2.2 WAF Detection and Fingerprinting

HTTP Response Code

Some WAFs modify the HTTP response codes if the request is hostile; for example:

mod_security > `406 Not Acceptable`

AQTRONIX WebKnight > `999 No Hacking`

1.2.2.2 WAF Detection and Fingerprinting

HTTP Response Body

It is also possible to detect the presence of the WAF plainly in the response body.

1.2.2.2 WAF Detection and Fingerprinting

HTTP Response Body: mod_security

```
HTTP/1.1 406 Not Acceptable
Date: Mon, 7 Apr 2014 11:10:50 GMT
Server: Apache
Content-Length: 226
Keep-Alive: timeout=10, max=30
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1
<head><title>Not Acceptable!</title></head><body><h1>Not Acceptable!</h1>
<p>An appropriate representation of the requested resource could not be found on this server.
This error was generated by Mod_Security.</p></body></html>
```

1.2.2.2 WAF Detection and Fingerprinting

HTTP Response Body: AQTRONIX WebKnight

WebKnight Application Firewall Alert

Your request triggered an alert! If you feel that you have received this page in error, please contact the administrator of this web site.

What is WebKnight?
AQTRONIX WebKnight is an application firewall for web servers and is released under the GNU General Public License. It is an ISAPI filter for securing web servers by blocking certain requests. If an alert is triggered WebKnight will take over and protect the web server.

For more information on WebKnight:
http://www.aqtronix.com/WebKnight/

AQTRONIX WebKnight

1.2.2.2 WAF Detection and Fingerprinting

HTTP Response Body: dotDefender

OUTLINE

1.2.2.2 WAF Detection and Fingerprinting

Close Connection

An interesting feature supported by some WAFs is `close connection`.

It is useful in dropping the connection in the case the WAF detects a malicious request.

1.2.2.2 WAF Detection and Fingerprinting

Close Connection: mod_security

Here is a possible implementation with mod_security to detect a brute force attack:

```
SecAction phase:1,id:109,initcol:ip=%{REMOTE_ADDR},nolog

SecRule ARGS:login "!^$"
"nolog,phase:1,id:110,setvar:ip.auth_attempt=+1,deprecatevar:ip.auth_attempt=20/120"

SecRule IP:AUTH_ATTEMPT "@gt 25"
"log,drop,phase:1,id:111,msg:'Possible Brute Force Attack'"
```

Source and Meaning > https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual#drop

1.2.2.2 WAF Detection and Fingerprinting

Many penetration testing tools have features to detect the presence of a WAF. These features are both used as a first step to understand how to craft payloads and if it is needed.

An example would be to obfuscate the attack vector or use a specific bypass.

1.2.2.2 WAF Detection and Fingerprinting

The most well-known tool made by Sandro Gauci and Wendel G. Henrique is called **wafw00f**.

Wafw00f is a tool written in python that can detect up to 20 different WAF products.

https://code.google.com/p/waffit/

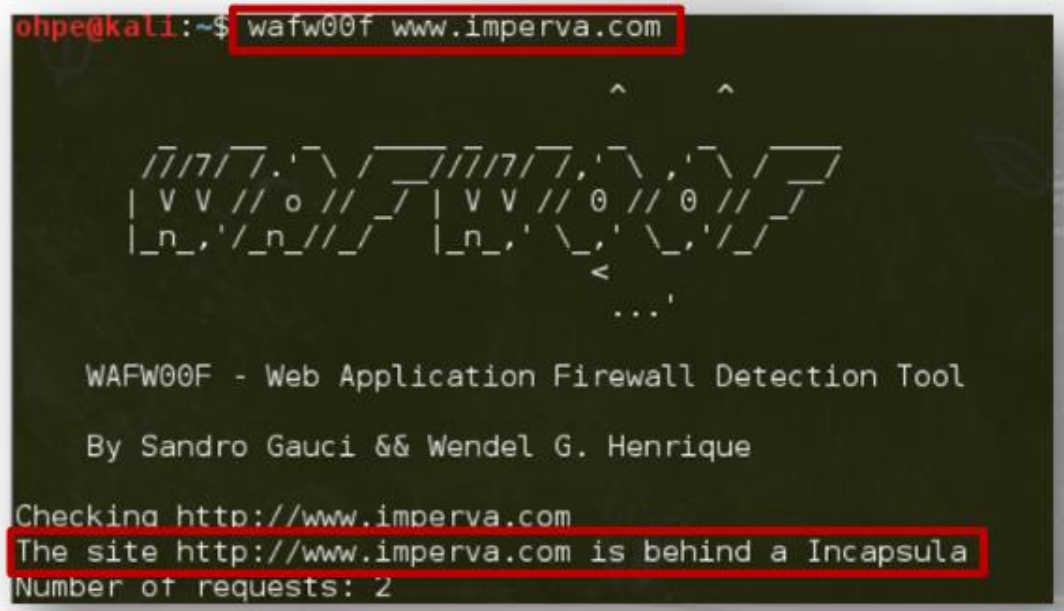
1.2.2.2 WAF Detection and Fingerprinting

The techniques used to detect a WAF are similar to those we have seen previously:

1. Cookies
2. Server Cloaking
3. Response Codes
4. Drop Action
5. Pre-Built-In Rules

1.2.2.2 WAF Detection and Fingerprinting

Scanning a website with wafw00f is very simple, and the following image confirms it:



OUTLINE

1.2.2.2 WAF Detection and Fingerprinting

As an addition to wafw00f you might want to use Nmap. It contains a script that tries to detect the presence of a web application firewall, its type and version.

The script file is **http-waf-fingerprint** and is authored by Hani Benhabiles.

http://nmap.org/nsedoc/scripts/http-waf-fingerprint.html

1.2.2.2 WAF Detection and Fingerprinting

Scanning a website with nmap is as simple as running wafw00f. We just require the script name to be in the command:

```
ohpe@kali:/$ nmap --script=http-waf-fingerprint www.imperva.com -p 80

Starting Nmap 6.40 ( http://nmap.org ) at 2014-04-08 14:20 CEST
Nmap scan report for www.imperva.com (185.11.125.104)
Host is up (0.045s latency).
PORT   STATE SERVICE
80/tcp open  http
| http-waf-fingerprint:
|   Detected WAF
|_    Incapsula WAF

Nmap done: 1 IP address (1 host up) scanned in 11.57 seconds
```

1.2.2.2 WAF Detection and Fingerprinting

Another interesting resource is **imperva-detect** by Lamar Spells. This utility is 100% focused on the detection of an Imperva WAF and it runs 6 tests, one baseline and five additional:

```
#    Test 0 - Baseline to establish expected behavior
#    Test 1 - "Web Leech" blocking
#    Test 2 - "E-mail Robot" blocking
#    Test 3 - BlueCoat Proxy Manipulation blocking
#    Test 4 - Web Worm blocking
#    Test 5 - XSS blocking
```

https://code.google.com/p/imperva-detect/

1.2.2.2 WAF Detection and Fingerprinting

The following image is an example of how to run imperva-detect test scripts:

```
ohpe@kali:/$ imperva-detect.sh blog.imperva.com

--- Testing [blog.imperva.com] for presence of application firewall ---

Test 0 - Good User Agent...
  -- HTTP Return Code = 200
  -- Content Size Downloaded = 79749
Test 1 - Web Leech User Agent...
  -- HTTP Reutrn Code  = 200 & downloaded content size is the same -- application firewall not detected
Test 2 - E-mail Collector Robot User Agent Blocking...
  -- HTTP Reutrn Code  = 200 & downloaded content size is the same -- application firewall not detected
Test 3 - BlueCoat Proxy Manipulation Blocking...
  -- HTTP Return Code = 302 -- expected 404 -- application firewall possibly present
Test 4 - Web Worm Blocking...
  -- HTTP Return Code [404] encountered - application firewall possibly present
Test 5 - XSS Blocking...
  -- HTTP Return Code = 302 -- while checking XSS blocking

--- Tests Finished on [blog.imperva.com] -- 3 out of 5 tests indicate Imperva application firewall present ---
```

1.2.3 Client-Side Filters

Web Application Firewalls and libraries are filtering solutions used to block web attacks, **server-side** at the heart of web applications. Over the years, this has become the "classic" and consolidated approach.

However, in the last ten years another approach has arisen. The concept is to block web attacks **client-side** within web browsers. These browsers are the primary mean used to address attacks.

1.2.3 Client-Side Filters

The goal of client-side defenses is to protect users against vulnerabilities in web applications. Of course this approach is not simple, and defenses need to be generic enough to always be enabled. If they are not, they can become a blocker for the browsers themselves and to their respective users.

From an attacker's point of view, we want to understand these mechanisms and how to bypass them. Our aim is the target users who would otherwise be protected.

1.2.3.1 Browser Add-ons

The first browser protection began in the open source community.

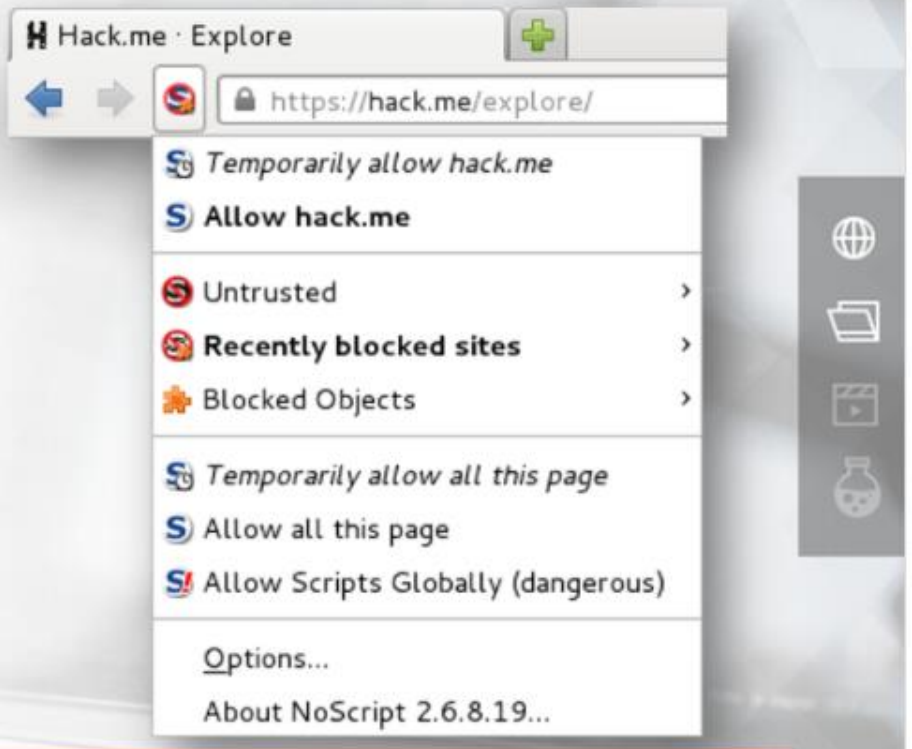
The pioneer of the first valid solution was Giorgio Maone, in late 2005, with the **NoScript Security Suite** extension for Firefox.

https://addons.mozilla.org/it/firefox/addon/noscript/

OUTLINE

1.2.3.1 Browser Add-ons

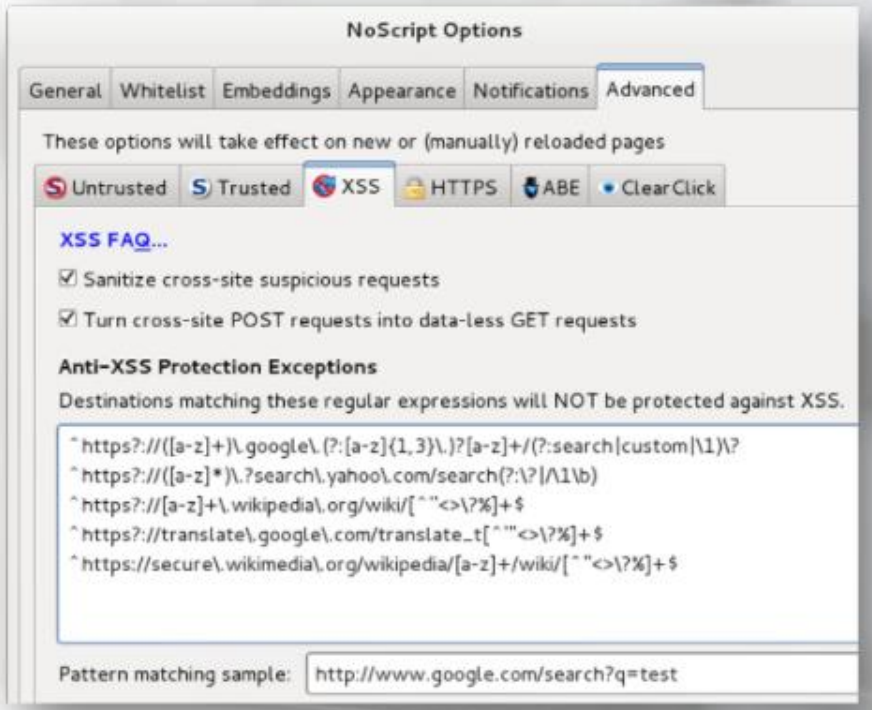
NoScript is a whitelist-based security tool that basically disables all the executable web content (JavaScript, Java, Flash, Silverlight, ...) and lets the user choose which sites are "trusted", thus allowing the use of these technologies.



OUTLINE

1.2.3.1 Browser Add-ons

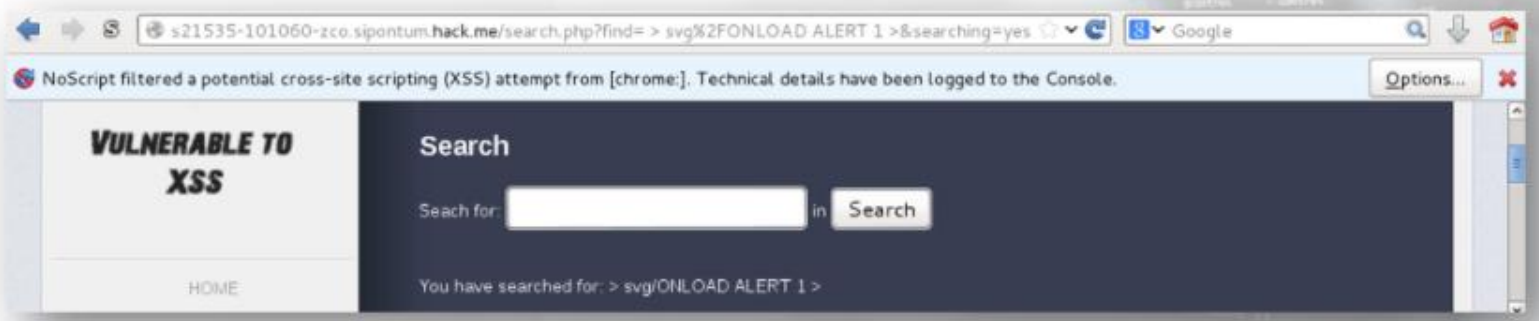
NoScript is easy enough to use; however, the strongest point of this extension is the extensive list of security features supported.



OUTLINE

1.2.3.1 Browser Add-ons

Among the features, the strong and powerful **anti-XSS protection** is probably one of the most effective browser-based solutions to prevent targeted malicious Web attacks.



http://noscript.net/features#xss

OUTLINE

1.2.3.2 Native Browser Filters

History

The first attempt at blocking malicious requests "*natively*" (i.e. internally in the browser), was made by Microsoft and introduced in Internet Explorer 8 as **XSS Filter**.

This filter attempts to block reflected XSS attacks by applying regular expressions to response data.

http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx

1.2.3.2 Native Browser Filters

History

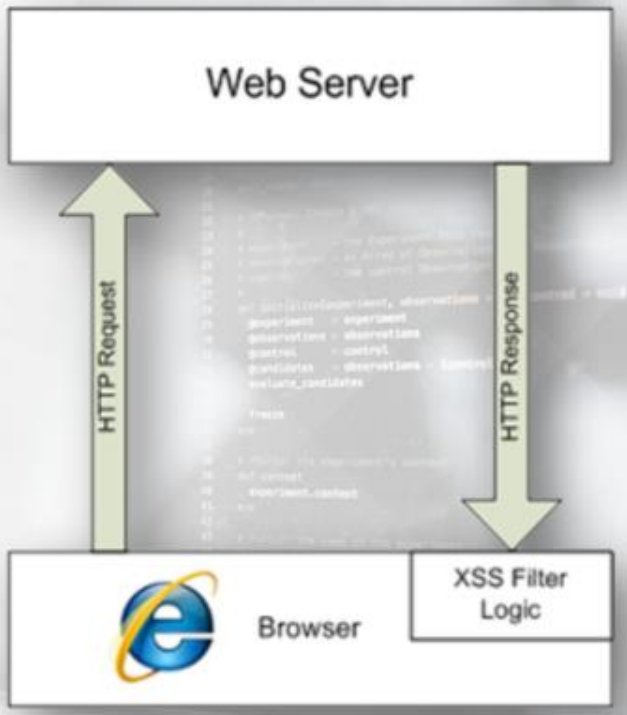
After Microsoft, Google Chrome introduced their own cross-site scripting filters, **XSS Auditor**.

This filter is slightly different from IE's XSS Filter and NoScript. Instead of being layered on top of the browser, it is integrated into WebKit/Blink, which are the rendering engines that support XSS Auditor.

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer)

The architecture and implementation of XSS Filter in Internet Explorer is explained in this **blog post**.



http://blogs.technet.com/b/srd/archive/2008/08/18/ie-8-xss-filter-architecture-implementation.aspx

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer)

The XSS Filter rules are **hardcoded** in the `c:\windows\system32\mshtml.dll` library. We have multiple ways to inspect them using the following:

- Hex editors like WinHex, or Notepad++ with TextFX plugin
- IDAPro
- MS-DOS commands! ← **Faster solution!**

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer)

The command is:

```
findstr /C:"sc{r}" \WINDOWS\SYSTEM32\mshtml.dll | find "{"
```

If you want a more "human-readable" version, then export the result to a file and use a text editor to read the content.

```
findstr /C:"sc{r}" \WINDOWS\SYSTEM32\mshtml.dll | find "{" > savepath
```

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer)

Here are few extracted rules in Internet Explorer 11:

```
{<EM{B}ED[ /+\t].*?((src)|(type)).*?=}
{[ /+\t\"\'`]{o}n\c\c\c+?[ +\t]*?=.}
{[i]?f{r}ame.*?[ /+\t]*?src[ /+\t]*=}
{<fo{r}m.*?>}
{<sc{r}ipt.*?[ /+\t]*?((src)|(xlink:href)|(href))[ /+\t]*=}
{<BA{S}E[ /+\t].*?href[ /+\t]*=}
{<LI{N}K[ /+\t].*?href[ /+\t]*=}
{<ME{T}A[ /+\t].*?http-equiv[ /+\t]*=}
```

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer): Filter Examination

This filter detects the string **javascript:**

```
((j|(&#x?0*((74)|(4A)|(106)|(6A));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|(tab;)|(newline;
))))*(a|(&#x?0*((65)|(41)|(97)|(61));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|(tab;)|(newli
ne;))))*(v|(&#x?0*((86)|(56)|(118)|(76));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|(tab;)|(n
ewline;))))*(a|(&#x?0*((65)|(41)|(97)|(61));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|(tab;)
|(newline;))))*(s|(&#x?0*((83)|(53)|(115)|(73));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|(t
ab;)|(newline;))))*(c|(&#x?0*((67)|(43)|(99)|(63));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)
|(tab;)|(newline;))))*{(r|(&#x?0*((82)|(52)|(114)|(72));?))}([\t]|(&((#x?0*(9|(13)|(10)|A
|D);?)|(tab;)|(newline;))))*(i|(&#x?0*((73)|(49)|(105)|(69));?))([\t]|(&((#x?0*(9|(13)|(1
0)|A|D);?)|(tab;)|(newline;))))*(p|(&#x?0*((80)|(50)|(112)|(70));?))([\t]|(&((#x?0*(9|(13
)|(10)|A|D);?)|(tab;)|(newline;))))*(t|(&#x?0*((84)|(54)|(116)|(74));?))([\t]|(&((#x?0*(9
|(13)|(10)|A|D);?)|(tab;)|(newline;))))*(:|(&((#x?0*((58)|(3A));?)|(colon;)))).}
```


OUTLINE

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer): Filter Examination

This filter detects the string **vbscript:**

```
{(v|(&#x?0*((86)|(56)|(118)|(76));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|(tab;)|(newline;
))))*(b|(&#x?0*((66)|(42)|(98)|(62));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|(tab;)|(newli
ne;))))*(s|(&#x?0*((83)|(53)|(115)|(73));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|(tab;)|(n
ewline;))))*(c|(&#x?0*((67)|(43)|(99)|(63));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|(tab;)
|(newline;))))*{(r|(&#x?0*((82)|(52)|(114)|(72));?))}([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|
(tab;)|(newline;))))*(i|(&#x?0*((73)|(49)|(105)|(69));?))([\t]|(&((#x?0*(9|(13)|(10)|A|D)
;?)|(tab;)|(newline;))))*(p|(&#x?0*((80)|(50)|(112)|(70));?))([\t]|(&((#x?0*(9|(13)|(10)|
A|D);?)|(tab;)|(newline;))))*(t|(&#x?0*((84)|(54)|(116)|(74));?))([\t]|(&((#x?0*(9|(13)|(
10)|A|D);?)|(tab;)|(newline;))))*(:|(&((#x?0*((58)|(3A));?)|(colon;)))).}
```


OUTLINE

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer):

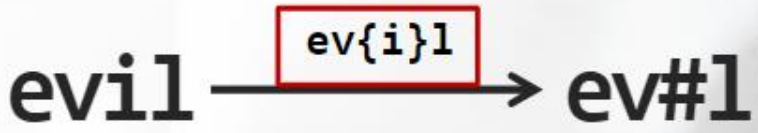
In the previous slides, you probably noticed the `red` characters highlighted between curly braces. They are also known as '**neutering**' characters.

The use of neutering characters is the approach that the IE team decided to use to neutralize detected attacks, which is a kind of trade-off between usability and effectiveness.

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer):

Basically, once a malicious injection is detected, the XSS Filter modifies the evil part of the payload by adding the # (*pound*) character in place of the neuter character, defined in the rules.



Let's look at some examples.

OUTLINE

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer): Neutering in Action

The XSS attack:

```
<svg/onload=alert(1)>
```

is transformed to:

```
<svg/#nload=alert(1)>
```

Filter rule

```
{[ /+\t\"\'`]{o}n\c\c\c+?[ +\t]*?=.}
```

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer): Neutering in Action

The XSS attack:

```
<isindex/onmouseover=alert(1)>
```

is transformed to:

```
<is#ndex/#nmouseover=alert(1)>
```

Filter rules

```
{<is{i}ndex[ /+\t>]}
{[ /+\t\"\'`]{o}n\c\c\c+?[ +\t]*?=.}
```

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer):

The XSS Filter has a few rules, just 25, but are well constructed and difficult to attack.

Over the years, several bypasses have been discovered; however, the latest versions *'seem'* stronger than past versions (unless you have a 0-day ☺).

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer):

XSS Filter is enabled by default in the Internet, Trusted, and Restricted security zones, but an interesting feature was introduced **to disable the filter**.

The main reason was because some sites may depend on the reflected values that the filter searches for.

http://msdn.microsoft.com/en-us/library/dd565647(v=vs.85).aspx#remediation

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer):

Web sites that chose to opt-out of this protection can use the HTTP response header:

```
X-XSS-Protection: 0
```

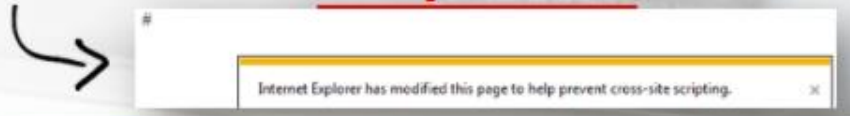
1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer):

Later, the IE team added support to a new token in the `X-XSS-Protection` header:

```
X-XSS-Protection: 1; mode=block
```

With this token, if a potential reflected XSS attack is detected, the browser, rather than attempting to sanitize the page, will render a simple `#`. Here is a **simple test**.

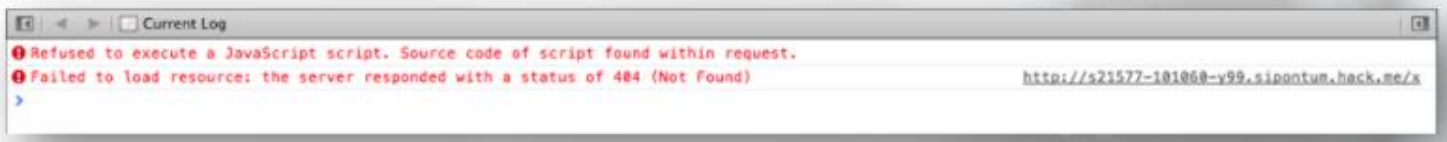


http://www.enhanceie.com/test/xss/BlockMode.asp

1.2.3.2 Native Browser Filters

XSS Filter (Internet Explorer):

Even if the `X-XSS-Protection` header was initially introduced by Internet Explorer, today other browsers based on WebKit and Blink support it.



Safari

OUTLINE

1.2.3.2 Native Browser Filters

XSSAuditor (WebKit/Blink)

In the footsteps of Internet Explorer, some researchers developed their own set of client-side XSS filters, also known as **XSSAuditor**.

The implementation is only for the **Blink/WebKit** rendering engines. This is enabled by default in browsers such as Google Chrome, Opera and Safari.

http://www.adambarth.com/papers/2010/bates-barth-jackson.pdf

1.2.3.2 Native Browser Filters

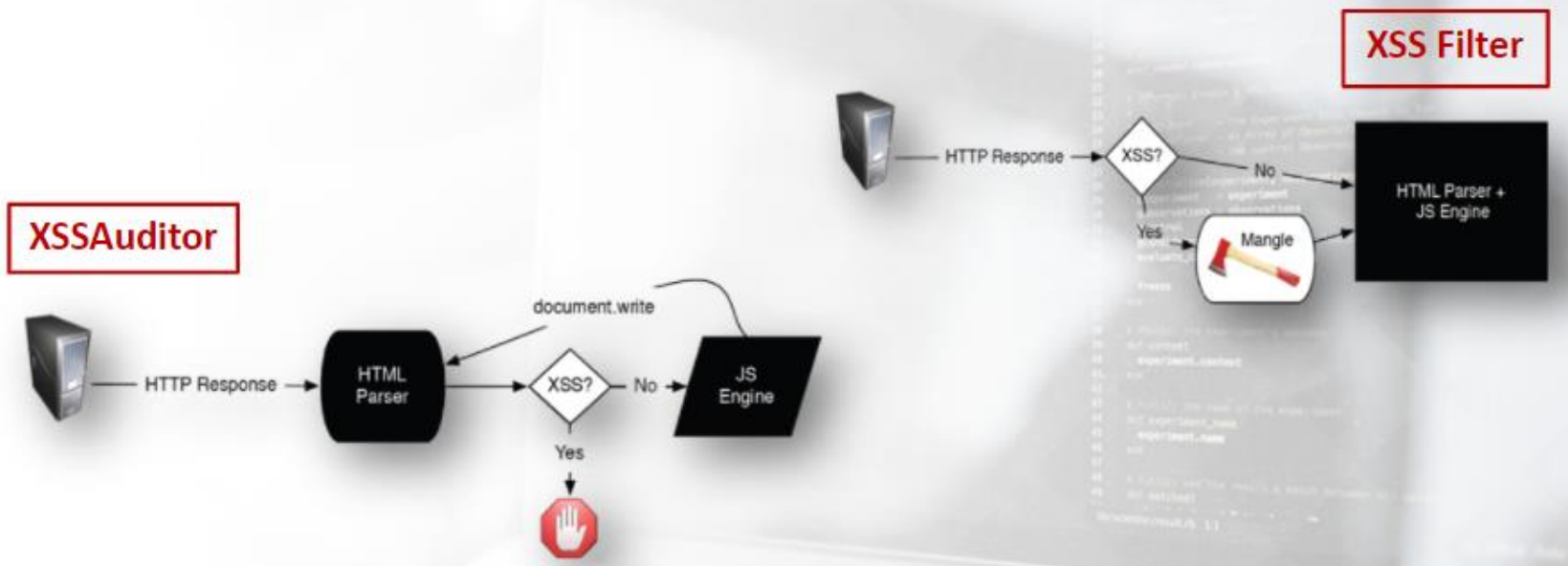
XSSAuditor (WebKit/Blink)

Despite the IE XSS Filter, XSSAuditor adopts a different approach to the problem. As the authors claims, the new filter design is both effective and highly precise. To do this, they placed XSSAuditor in between the HTML Parser and JS engine.

The image on the following slide will clarify the differences.

1.2.3.2 Native Browser Filters

XSSAuditor (WebKit/Blink): XSS Filter vs XSSAuditor



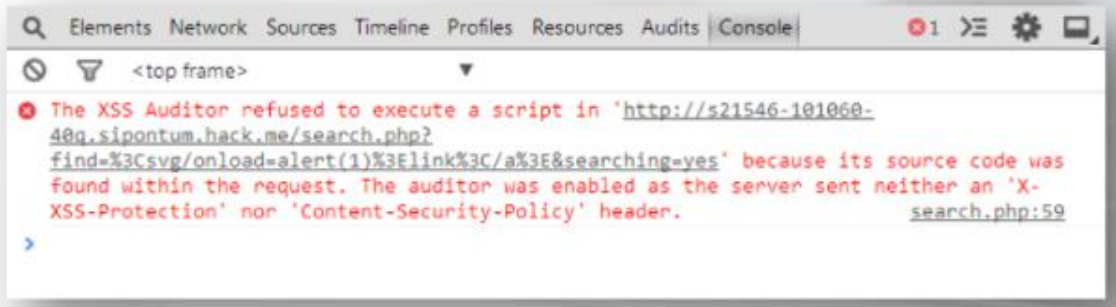
Images taken from http://www.adambarth.com/papers/2010/bates-barth-jackson.pdf

OUTLINE

1.2.3.2 Native Browser Filters

XSSAuditor (WebKit/Blink)

The filter analyzes both the inbound requests and the outbound. If, in the parsed HTML data, it finds executable code within the response, then it stops the script and generates a console alert similar to the following:



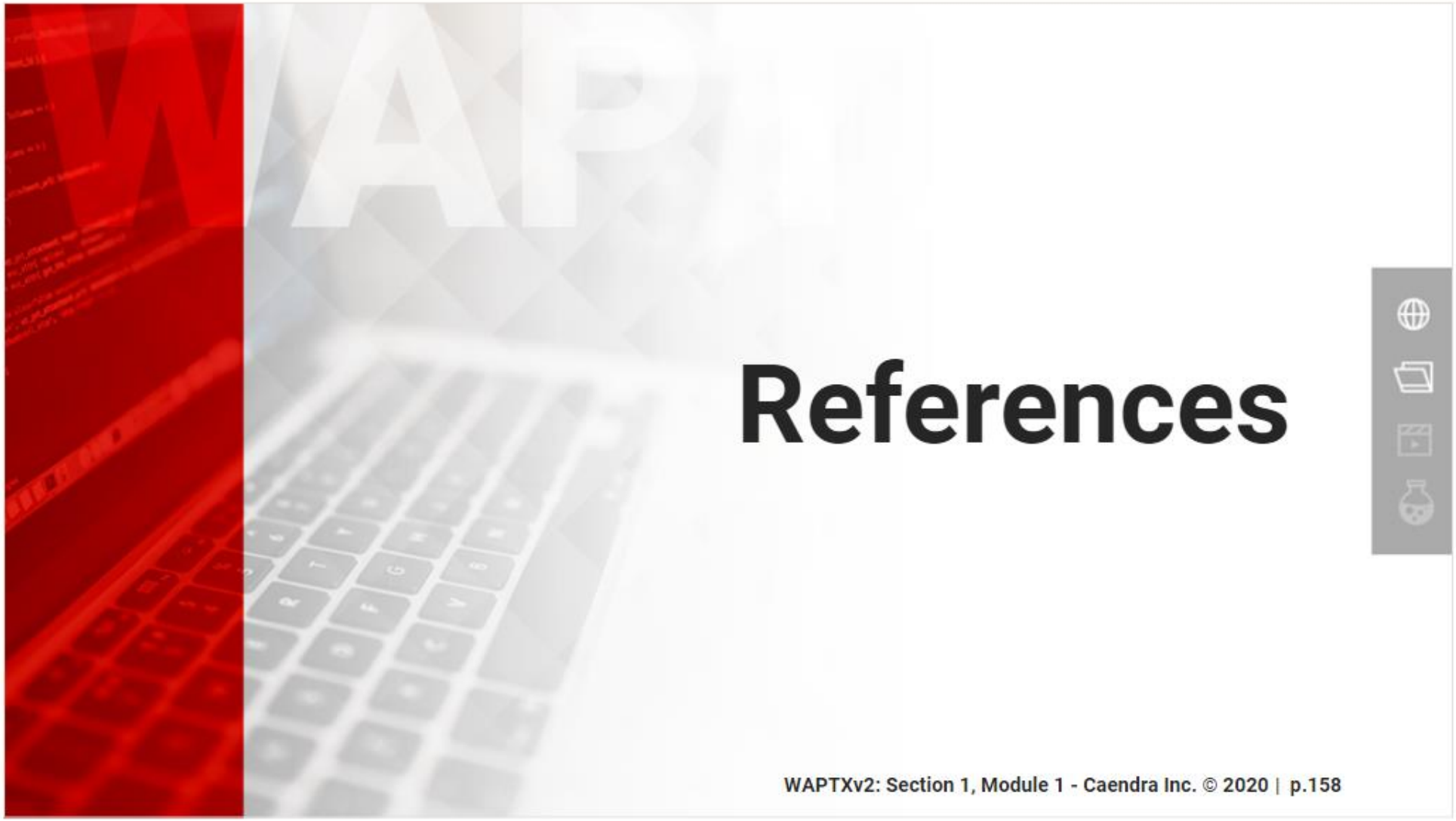
OUTLINE

1.2.3.2 Native Browser Filters

XSSAuditor (WebKit/Blink)

Over the years, even with XSS Auditor, security researchers found multiple bypasses...Oh yes, they did!

A simple search on google about `XSS Auditor` returns more information on bypasses than on the filter itself.



References

OUTLINE

References

RFC 3986

http://tools.ietf.org/html/rfc3986#section-2.1

(Please) Stop Using Unsafe Characters in URLs: Character Encoding Chart

http://perishablepress.com/stop-using-unsafe-characters-in-urls/

RFC 2616

https://tools.ietf.org/html/rfc2616

ISO/IEC 8859-1

http://en.wikipedia.org/wiki/ISO/IEC_8859-1

References

PHP header()

http://www.php.net/header

HttpResponse Class

http://msdn.microsoft.com/en-us/library/system.web.httpresponse

HTML Document Representation: 5.3 Character References

http://www.w3.org/TR/1998/REC-html40-19980424/charset.html#h-5.3

HTML Standard: 12.1.4 Character References

http://www.w3.org/TR/html5/single-page.html#character-references

References

Character entity references in HTML

http://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references#Character_entity_references_in_HTML

Reddit

http://www.reddit.com/

TinyURL

http://tinyurl.com/

base_convert

http://www.php.net/manual/en/function.base-convert.php

References

Base64: Implementations and History

http://en.wikipedia.org/wiki/Base64#Implementations_and_history

base64_encode

https://www.php.net/base64_encode

base64_decode

https://www.php.net/base64_decode

WindowOrWorkerGlobalScope.btoa()

https://developer.mozilla.org/en-US/docs/Web/API/Window.btoa

References

The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

http://www.joelonsoftware.com/articles/Unicode.html

Homoglyph

http://en.wikipedia.org/wiki/Homoglyph

Unicode® Technical Standard #39 - UNICODE SECURITY MECHANISMS

http://www.unicode.org/reports/tr39/

Unicode Utilities: Confusables

http://unicode.org/cldr/utility/confusables.jsp

References

Homoglyph Attack Generator

http://www.irongeek.com/homoglyph-attack-generator.php

Out of Character: Use of Punycode and Homoglyph Attacks to Obfuscate URLs for Phishing

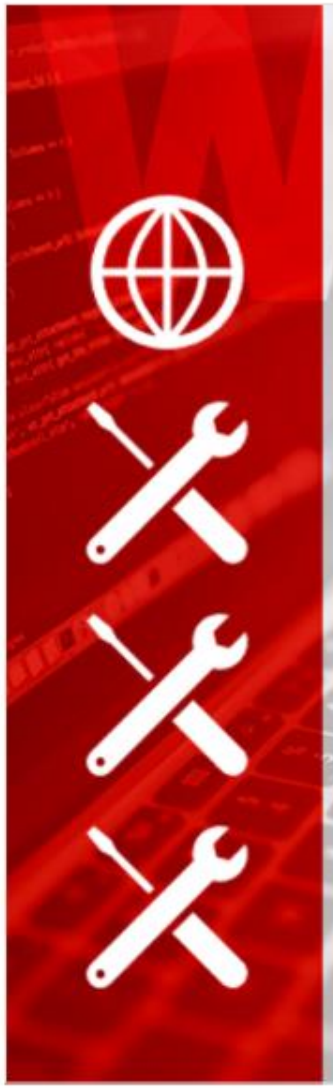
http://www.irongeek.com/i.php?page=security/out-of-character-use-of-punycode-and-homoglyph-attacks-to-obfuscate-urls-for-phishing

Creative usernames and Spotify account hijacking

http://labs.spotify.com/2013/06/18/creative-usernames/

Unicode Utilities: Description and Index

http://unicode.org/cldr/utility/



References

Codepoints

http://codepoints.net/

Transformation tools for Unicode text

http://txtn.us/

Unicode Text Converter

http://www.panix.com/~eli/unicode/convert.cgi

OWASP Enterprise Security API

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

References

ModSecurity

http://www.modsecurity.org/

Deterministic finite automaton

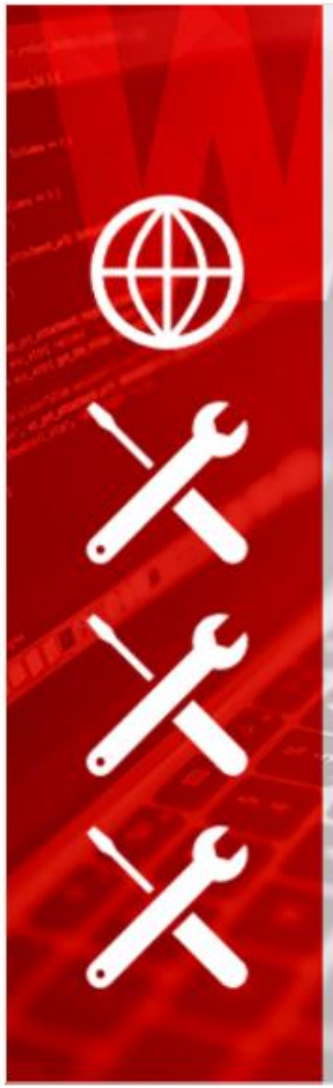
http://en.wikipedia.org/wiki/Deterministic_finite_automaton

Nondeterministic finite automaton

http://en.wikipedia.org/wiki/Nondeterministic_finite_automaton

Control character

http://en.wikipedia.org/wiki/Control_character



References

GitHub: SpiderLabs / ModSecurity Documentation

https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual#drop

GitHub: EnableSecurity / wafw00f

https://code.google.com/p/waffit/

File http-waf-fingerprint

http://nmap.org/nsedoc/scripts/http-waf-fingerprint.html

imperva-detect

https://code.google.com/p/imperva-detect/

References

NoScript Security Suite

https://addons.mozilla.org/en-US/firefox/addon/noscript/

NoScript: Anti-XSS protection

http://noscript.net/features#xss

IE8 Security Part IV: The XSS Filter

http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx

IE 8 XSS Filter Architecture / Implementation

http://blogs.technet.com/b/srd/archive/2008/08/18/ie-8-xss-filter-architecture-implementation.aspx

References

Event 1046 - Cross-Site Scripting Filter: Remediation

http://msdn.microsoft.com/en-us/library/dd565647(v=vs.85).aspx#remediation

Regular Expressions Considered Harmful in Client-Side XSS Filters

http://www.adambarth.com/papers/2010/bates-barth-jackson.pdf

The META element

https://www.w3.org/TR/html401/struct/global.html#h-7.4.4.2

Unicode Security Guide

http://websec.github.io/unicode-security-guide/

References

Comparison of regular expression engines

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

Regular Expressions Reference Table of Contents

https://www.regular-expressions.info/refflavors.html

Base 36 as senary compression

http://tinyurl.com/jfvqr