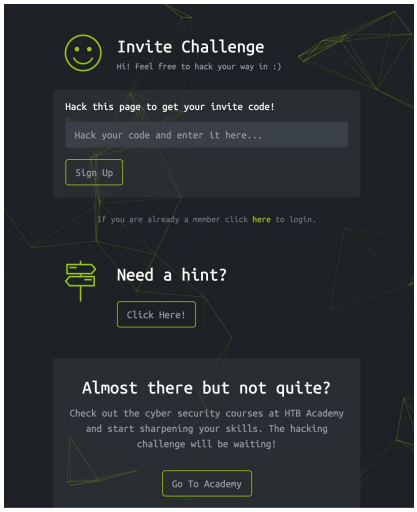
Nicolas Barraclough
12/3/20
CS373 - Defense Against the Dark Arts

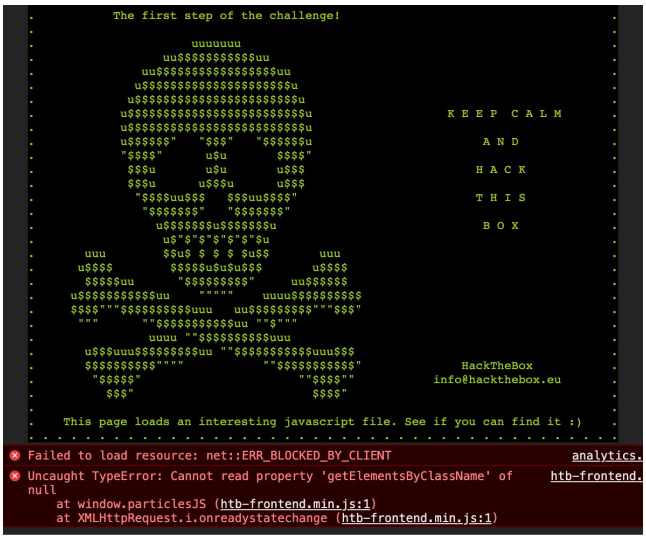
Lab 5 - Hack the Box



I see why getting the invite code is worth 5 points!

The hint was to check the console.

A cool ascii art skull & crossbones with a message saying to look for a javascript file showed which could refer to one of three found within the page sources:
calm.js (The ascii image for the console)
htb-frontend.min.js (which returned an Uncaught TypeError)
inviteapi.min.js (an oddly commented, obfuscated .js file)

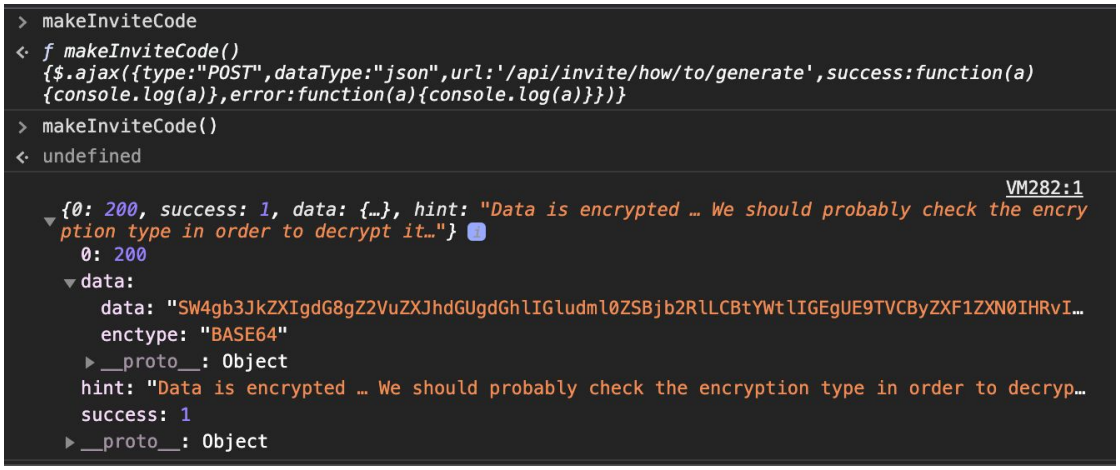


The inviteapi javascript file appears to be making a bunch of different functions by splitting a string (shown at the end of the file below) and appending curly braces to each one. Here is the inviteapi.min.js file contents:

```
//This javascript code looks strange...is it obfuscated???

eval(function(p,a,c,k,e,r){e=function(c){return
c.toString(a)};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}('0 3()
{$.4({5:"6",7:"8",9:\'/b/c/d/e/f\',g:0(a){1.2(a)},h:0(a)
{1.2(a)}})}',18,18,'function|console|log|makeInviteCode|ajax|t
ype|POST|dataType|json|url||api|invite|how|to|generate|success
|error'.split('|'),0,{}))
```

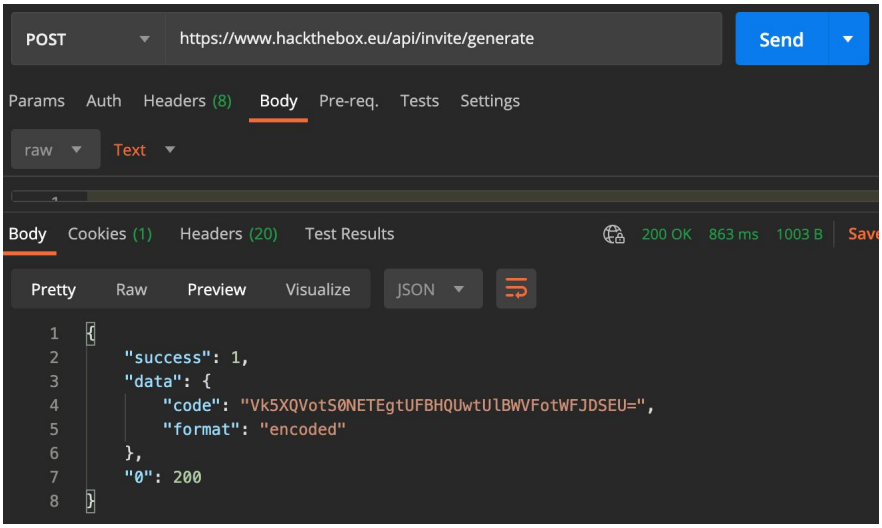
There were a few function names that caught my attention, but one that seemed to be more of what I was looking for, was the name “makeInviteCode”. I’m not sure what to do with this function, so I will try to just use it within the invite page’s console.



Aha! The json formatted data property returned by the function is a long, encrypted string. The encryption type is given (BASE64) with a hint that advises decrypting the ‘data’ string. I don’t feel like doing the decryption by hand, so I found a decryption website online called, base64decode.org. The decoded string reads:

In order to generate the invite code, make a POST request to /api/invite/generate.

In my Cloud App Development class, we used Postman to generate HTTP requests for testing purposes. I’ll try using Postman to send the request. I’m assuming the url preceding the file path will just be the same as domain as the invite page (https://www.hackthebox.eu/).
I’m not sure what needs to be in the body of the request right now, so I’ll first try an empty POST request to see what response I get back.



200 status code! Looks like an empty request worked. I’ve received a “code”, but I’m skeptical that this will be the final invite code that I enter into the Invite Challenge. The “format” property suggests that this code is still encoded. I’ll try that Base64 tool that I used previously.

After decoding the ‘code’ string with Base64, I am left with what appears to be a properly formatted invite code:

VNWAZ-KCDLH-PPGAL-RPVTZ-XRCHE

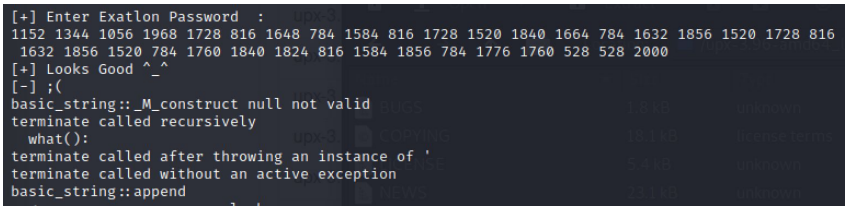
I’ll try entering that into the Invite Challenge page!

I’m in!!

I tried to start off with an easier challenge and found one called “Exatlon” by OctopusTR that was worth only 20 points. I extracted the zip file that came with the challenge and copied it into my Kali VM to execute it there. Upon execution, the following message is shown in the terminal:

After entering an invalid password, a ‘;(‘ is printed and the same title and prompt is given again. I decided to run strings on the file to see if any coherent strings would be output as possible passwords. No luck. Not a single string out of the hundreds (or thousands) of strings looked like it could be the possible password. I did notice however, that the very first and last two strings that were output was “UPX!”
Turns out UPX stands for the Ultimate Packer for eXecutables.
After decompressing the file with upx, and retrying the strings command on it, I was presented with a much more coherent list of strings.

I found the starting prompt with a bunch of 3-4 digit numbers and then the frowning face and what appears to be the success message!



My guess is that the password has something to do with the list of numbers (since the list of numbers was not the password).

Using GDB, I found the address of the main() function, I’ll set a break at that address.

I was able to find the function that encrypts my input and upon entering a single input, I can see what the function returns. I’ve decided to do a brute force method of mapping out each single character input to try to get it to match the string of numbers I found in the above screenshot.

Here's my mapping:

A -> 1040	N -> *13	[-> *26	a -> 1552	n -> 1760	{ -> 1968 (last)
B -> + 16*1	O ->	\ -> *27	b -> 1568	o -> 1776	
C ->	P ->	] -> *28	c -> 1584	p -> 1792	SPACE -> 528
D ->	Q ->	^ -> *29	d -> 1600	q -> 1808	
E ->	R ->	_ -> *30	e -> 1616	r -> 1824	
F	S ->	` -> 1536	f -> 1632	s -> 1840	
G	T ->		g -> 1648	t -> 1856	
H	U ->		h -> 1664	u -> 1872	
I	V ->		i -> 1680	v -> 1888	
J	W ->		j -> 1696	w -> 1904	
K	X ->		k -> 1712	x -> 1920	
L	Y -> *24		l -> 1728	y -> 1936	
M	Z -> *25		m -> 1744	z -> 1952	

Okay I'm spending too much time mapping, I think I've found the encryption pattern. The encrypted string is made up of characters where the character's decimal ascii number is multiplied by 16. So, for each number in the encrypted string, I'll divide it by 16 and convert to the corresponding ascii character.

encrypted string:

```
1152 1344 1056 1968 1728 816 1648 784 1584 816 1728 1520 1840 1664 784 1632 1856 1520 1728 816
 1632 1856 1520 784 1760 1840 1824 816 1584 1856 784 1776 1760 528 528 2000
```

1152 / 16 = H	1728 / 16 = l	816 / 16 = 3	1856 / 16 = t
1344 / 16 = T	1520 / 16 = _	1632 / 16 = f	784 / 16 = 1
1056 / 16 = B	1840 / 16 = s	1856 / 16 = t	1776 / 16 = o
1968 / 16 = {	1664 / 16 = h	1520 / 16 = _	1760 / 16 = n
1728 / 16 = l	784 / 16 = 1	784 / 16 = 1	528 / 16 = !
816 / 16 = 3	1632 / 16 = f	1760 / 16 = n	528 / 16 = !
1648 / 16 = g	1856 / 16 = t	1840 / 16 = s	2000 / 16 = }
784 / 16 = 1	1520 / 16 = _	1824 / 16 = r	
1584 / 16 = c	1728 / 16 = l	816 / 16 = 3	
816 / 16 = 3		1584 / 16 = c	

Password = HTB{l3g1c3l_sh1ft_l3ft_1nsr3ct1on!!}