Exploiting Reversing (ER) series: Article 03 | Chrome (part 01)

(a step-by-step security research series on Win, macOS, hypervisors, browsers, and others)

by Alexandre Borges
release date: JAN/22/2025 | rev: A.1

00. Quote

"Our lives are the sum of our choices…and we cannot escape the past."
(Eugene Kittridge played by Henry Czerny| "Mission: Impossible – Dead Reckoning – Part One" movie – 2023)

01. Introduction

Welcome to the third article of the **Exploiting Reversing (ER) series**, where we will review concepts, techniques and practical steps related to vulnerability research, internals, reverse engineering and, eventually, exploitation. My last articles are listed below:

- **ERS_02:** https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/
- **ERS_01:** https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/
- **MAS_10:** https://exploitreversing.com/2025/01/15/malware-analysis-series-mas-article-10/
- **MAS_09:** https://exploitreversing.com/2025/01/08/malware-analysis-series-mas-article-09/
- **MAS_08:** https://exploitreversing.com/2024/08/07/malware-analysis-series-mas-article-08/
- **MAS_07:** https://exploitreversing.com/2023/01/05/malware-analysis-series-mas-article-7/
- **MAS_06:** https://exploitreversing.com/2022/11/24/malware-analysis-series-mas-article-6/
- **MAS_05:** https://exploitreversing.com/2022/09/14/malware-analysis-series-mas-article-5/
- **MAS_04:** https://exploitreversing.com/2022/05/12/malware-analysis-series-mas-article-4/
- **MAS_03:** https://exploitreversing.com/2022/05/05/malware-analysis-series-mas-article-3/
- **MAS_02:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/
- **MAS_01**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/

This text is an introductory, step-by-step article on Chrome, and V8 in special. Actually, it is the first part of articles to be published related to security research on Chrome, which presents fundamentals concepts, architecture's details and initial analysis to prepare readers to next articles about the topic.

Although browsers like Chrome offer most of its code as open-source code, it is not a trivial task to understand its internal mechanisms nor how they interact with each other. Furthermore, it is substantially hard to exploit vulnerabilities found on Chrome due to its increasing number of security protections against exploitation over years. The learning path to exploit Chrome is similar to other targets on information security: we need to learn concepts, architecture, how mechanisms work, how they interact

with each other, how they interact with user data (in this case, web pages) and how we can manipulate them.

02. Acknowledgments

The year is 2025. Four years ago, I started drafting articles with the sole purpose of helping the hacking and information security community, and as I could already imagine at that time, it would be challenging to find time to continue producing content, and indeed this side effect has been confirmed over time. As I have been using IDA Pro for a long time, I needed a license of my own, and that was when **Ilfak Guilfanov (@ilfak)** and **Hex-Rays SA (@HexRaysSA)** decided to help me, and since then they have provided continuous and decisive support to write this **Malware Analysis Series (MAS)**, which is focused on malware analysis, and the **Exploiting Reversing series (ERs)**, which is my **current and long-term series** on internals, vulnerability research and, eventually, exploitation in critical topics such as Windows, kernel drivers, macOS, browser and hypervisors.

Time flies, and companies around the world have become more demanding in terms of technical skills, but I still believe that one of the most effective ways to help these professionals is to write articles because such content can offer a solid method to learn details that would be a bit more difficult in live conferences or even other media. I still face serious time constraints to write, but I keep trying to do it because, in some way, I know that these series have been important for people's careers.

Life may be short, but every moment is worth it. Enjoy the journey and keep exploiting it!

03. Environment infrastructure

This article is quite different from others, and it does not have a fixed requirement, and it can be followed on Windows, Linux or macOS. Anyway, one of many possibilities is using a physical host or virtual machines, depending on reader's available resources, which will be used in this and next articles:

- **One physical or virtual machine with Windows 11.** You have two options:
  - Download a **virtual machine for VMware, Hyper-V, VirtualBox or Parallels from Microsoft** on: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/.
  - If you already have a valid license for Windows 11, so you can download the **ISO file** from: https://www.microsoft.com/software-download/windows11 .

- **Visual Studio Code:** https://code.visualstudio.com

- **Visual Studio:** https://visualstudio.microsoft.com/downloads/. and make sure to include the following components:
  - Desktop development with C++
  - C++/CLI support for v143 tools (x64/x86)
  - C++ CMake tools for Windows
  - C++ Clang tools for Windows

  - C++ MFC for latest v143 build tools (x86 & x64)
  - C++ ATL for latest v143 build tools (x86 & x64)
  - MSVC v143 – VS2022 C++ x64/x86 build tools (latest)
- **Windows SDK:** https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/
- **Physical or virtual machine running Ubuntu 24.04.x LTS:** https://ubuntu.com/download/desktop

04. Environment configuration

We will be starting our journey by setting up the necessary environment to interact with Chrome, and I am going to show simplified steps for Windows and Linux, which are not difficult to do, but demands attention. Basically, we will be generating the V8, which is JavaScript and WebAssembly engine. Details and explanations came later and, for now, it is time to set up a test environment.

4.1 **Compiling V8 on Windows**

The following steps can be adapted according to your system and your own choices, and I only tried to make them as simple as possible. Obviously, they are based on the official documentation offered by Chrome, eventually stripped off a few details that are not necessary this time, and with small additions to make the process clearer. Make sure you execute each step from a Windows terminal (not PowerShell):

- **md c:\articles**
- **cd articles**
- **md chromium**
- **cd chromium**
- Download and extract **depot_tools** from https://storage.googleapis.com/chrome-infra/depot_tools.zip
- Extract the package into articles\chromium\depot_tools folder.
- Modify the **PATH** system environment variable to include *C:\articles\chromium\depot_tools* folder at the front of the PATH variable.
- Add **DEPOT_TOOLS_WIN_TOOLCHAIN** user environment variable containing **0** as value.
- Add **vs2022_install** user variable and set it with the following value: *C:\Program Files\Microsoft Visual Studio\2022\Community*
- **cd articles\chromium\depot_tools**
- **gclient**
- **cp python3.bat python.bat**

  Make sure that the **python.bat** will be executed, and no other possible installed Python 3 version. Readers can check it by executing **where python** command.

- **gclient config** https://chromium.googlesource.com/v8/v8
- **gclient sync**
- **cd c:\articles\chromium**
- **fetch v8**

- **cd v8**
- **git fetch**
- **gclient sync -D**

It is timely to highlight that we usually can update the current v8 branch with the **git pull** command, even though if we are not in a branch then it will not work, and we will need to use **git fetch** command instead.

Before proceeding, readers should know that options used in the next two commands are appropriate for certain analysis' contexts while they are not for other ones, at the same way as many available options. If you want a normal and pure installation to learn about the environment, execute the same commands without including such options. Therefore, either you run:

- **gn gen out\x64.debug --args="v8_no_inline=true v8_optimized_debug=false is_component_build=false v8_enable_memory_corruption_api=true"**
- **gn gen out\x64.release --args="v8_no_inline=true is_debug=false v8_enable_memory_corruption_api=true"**
- **gn args out\x64.debug --list | more** *(it shows explanation about each used option)*
- **gn args out\x64.release --list | more** *(it shows explanation about each used option)*

Or execute only the following two commands, which are simpler:

- **gn gen out\x64.debug**
- **gn gen out\x64.release**

Anyway, the choice is yours. We should continue and run the next commands, which will take some time to finish:

- **python tools\dev\gm.py x64.debug** (from c:\articles\chromium\v8 folder)
- **python tools\dev\gm.py x64.release** (from c:\articles\chromium\v8 folder)

If everything goes well, you will see the following output:

```
C:\articles\chromium\v8>python tools\dev\gm.py x64.debug
# autoninja -C out\x64.debug d8
ninja: Entering directory `out\x64.debug'
[2408/2408] LINK d8.exe d8.exe.pdb
Done! - V8 compilation finished successfully.
C:\articles\chromium\v8>
C:\articles\chromium\v8>python tools\dev\gm.py x64.release
# autoninja -C out\x64.release d8
ninja: Entering directory `out\x64.release'
[2407/2407] LINK d8.exe d8.exe.pdb
Done! - V8 compilation finished successfully.
C:\articles\chromium\v8>
```

[Figure 01: v8 building process results on Windows]

If you want to update your source and installation, you should run the following commands:

- **git checkout main**
- **git pull origin**
- **gclient sync -D**

- **gn gen out\x64.debug --args="v8_no_inline=true v8_optimized_debug=false is_component_build=false v8_enable_memory_corruption_api=true"**
- **gn gen out\x64.release --args="v8_no_inline=true is_debug=false v8_enable_memory_corruption_api=true"**
- **python tools\dev\gm.py x64.debug** (from c:\articles\chromium\v8 folder)
- **python tools\dev\gm.py x64.release** (from c:\articles\chromium\v8 folder)

The entire process produces two versions (debug and release versions) of V8 (JavaScript and WebAssembly) used by Chrome browser, which can be found in out\x64.debug and out\x64.release folder, respectively. However, there is a command line version named **d8,** and we can use it to do JavaScript experiments like we were inside the Chrome browser, but using the command line, which we can evaluate by using a simple script named **script_test.js** that I have written for such a test:

```
c:\articles\chromium\v8>more script_test.js
function fact(a){
    if (a <= 1) return 1;
    c =  a * fact(a-1);
    o = new Object;
    o.x = c;
    %DebugPrint(o);
    return o.x;
}

c:\articles\chromium\v8>out\x64.debug\d8.exe --allow-natives-syntax
V8 version 13.0.0 (candidate)
d8> load("script_test.js")
undefined
d8> fact(5)
DebugPrint: 000002AA00049F29: [JS_OBJECT_TYPE]
 - map: 0x02aa00299525 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x02aa00282791 <Object map = 000002AA00281D9D>
 - elements: 0x02aa00000775 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x02aa00000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    000002AA00002D31: [String] in ReadOnlySpace: #x: 2 (const data field 0, attrs:
[WEC]) @ Any, location: in-object
 }
000002AA00299525: [Map] in OldSpace
 - map: 0x02aa0028183d <MetaMap (0x02aa0028188d <NativeContext[300]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 3
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x02aa002825bd <Map[28](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x02aa00299571 <Cell value= 0>
 - instance descriptors (own) #1: 0x02aa00049f45 <DescriptorArray[1]>
 - prototype: 0x02aa00282791 <Object map = 000002AA00281D9D>
 - constructor: 0x02aa002822b1 <JSFunction Object (sfi = 000002AA00253541)>
 - dependent code: 0x02aa00000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0

DebugPrint: 000002AA00049F61: [JS_OBJECT_TYPE]
 - map: 0x02aa00299525 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x02aa00282791 <Object map = 000002AA00281D9D>
 - elements: 0x02aa00000775 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x02aa00000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    000002AA00002D31: [String] in ReadOnlySpace: #x: 6 (const data field 0, attrs:
[WEC]) @ Any, location: in-object
 }
```

```
000002AA00299525: [Map] in OldSpace
 - map: 0x02aa0028183d <MetaMap (0x02aa0028188d <NativeContext[300]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 3
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x02aa002825bd <Map[28](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x02aa00299571 <Cell value= 0>
 - instance descriptors (own) #1: 0x02aa00049f45 <DescriptorArray[1]>
 - prototype: 0x02aa00282791 <Object map = 000002AA00281D9D>
 - constructor: 0x02aa002822b1 <JSFunction Object (sfi = 000002AA00253541)>
 - dependent code: 0x02aa00000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0

DebugPrint: 000002AA00049F7D: [JS_OBJECT_TYPE]
 - map: 0x02aa00299525 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x02aa00282791 <Object map = 000002AA00281D9D>
 - elements: 0x02aa00000775 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x02aa00000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    000002AA00002D31: [String] in ReadOnlySpace: #x: 24 (const data field 0, attrs:
 [WEC]) @ Any, location: in-object
 }
000002AA00299525: [Map] in OldSpace
 - map: 0x02aa0028183d <MetaMap (0x02aa0028188d <NativeContext[300]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 3
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x02aa002825bd <Map[28](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x02aa00299571 <Cell value= 0>
 - instance descriptors (own) #1: 0x02aa00049f45 <DescriptorArray[1]>
 - prototype: 0x02aa00282791 <Object map = 000002AA00281D9D>
 - constructor: 0x02aa002822b1 <JSFunction Object (sfi = 000002AA00253541)>
 - dependent code: 0x02aa00000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0

DebugPrint: 000002AA00049F99: [JS_OBJECT_TYPE]
 - map: 0x02aa00299525 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x02aa00282791 <Object map = 000002AA00281D9D>
 - elements: 0x02aa00000775 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x02aa00000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    000002AA00002D31: [String] in ReadOnlySpace: #x: 120 (const data field 0, attrs
: [WEC]) @ Any, location: in-object

 }
000002AA00299525: [Map] in OldSpace
 - map: 0x02aa0028183d <MetaMap (0x02aa0028188d <NativeContext[300]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 3
```

```
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x02aa002825bd <Map[28](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x02aa00299571 <Cell value= 0>
 - instance descriptors (own) #1: 0x02aa00049f45 <DescriptorArray[1]>
 - prototype: 0x02aa00282791 <Object map = 000002AA00281D9D>
 - constructor: 0x02aa002822b1 <JSFunction Object (sfi = 000002AA00253541)>
 - dependent code: 0x02aa00000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0

120
```

[Figure 02: d8 debug output]

The output seems to be complicated to understand, and we will be talking about it in later sections. The option **--allow-natives-strings** permits us to use debug functions during the JavaScript program. There are dozens of other options, but it is suitable to explain them when they are used in the right context.

4.1 **Compiling V8 on Linux**

As readers will notice, steps to generate V8 (and d8, of course) are remarkably similar to Windows, including slight changes. All these steps have been taken on Ubuntu 24.04, and they should work well in other distributions. Therefore, run the following commands:

- **mkdir -p articles/chromium**
- **cd articles/chromium**
- **git clone** https://chromium.googlesource.com/chromium/tools/depot_tools.git
- **export PATH=/home/aborges/articles/chromium/depot_tools:$PATH**
- **cd depot_tools/**
- **cd depot_tools**
- **git branch**
- **git checkout main** *(only to confirm that you are in the branch main)*
- **git pull origin** *(only to confirm that is everything up to date)*
- **gclient config https://chromium.googlesource.com/v8/v8**
- **gclient sync**
- **cd ..**
- **fetch v8**
- **cd v8**
- **git branch**
- **git checkout main**
- **git branch**
- **git pull origin**
- **gclient sync**
- **./build/install-build-deps.sh**
- **gn gen out/x64.debug --args='v8_no_inline=true v8_optimized_debug=false is_component_build=false v8_enable_memory_corruption_api=true'**

- **gn gen out/x64.release --args="v8_no_inline=true is_debug=false v8_enable_memory_corruption_api=true"**
- **gn args out/x64.debug --list | more**
- **tools/dev/gm.py x64.release**
- **tools/dev/gm.py x64.debug**

As expected, the compilation has finished without presenting any issue.

```
aborges@ubuntu01:~/articles/chromium/v8$ tools/dev/gm.py x64.release
# autoninja -C out/x64.release d8
ninja: Entering directory `out/x64.release'
[2420/2420] LINK ./d8
aborges@ubuntu01:~/articles/chromium/v8$ tools/dev/gm.py x64.debug
# autoninja -C out/x64.debug d8
ninja: Entering directory `out/x64.debug'
[2421/2421] LINK ./d8
```

[Figure 03: v8 building process results on Linux]

If you want to update your source and installation, you should run the following commands from /articles/chromium/v8 folder:

- **git map** (only to follow what's happening)
- **git fetch --tags**
- **git branch -a** (it helps you to learn what the last versions are)
- **git branch** (check the current branch)
- **git checkout main** (or any other branch)
- **git pull origin** (update)
- **gclient sync** (update)
- **gn gen out/x64.debug --args='v8_no_inline=true v8_optimized_debug=false is_component_build=false v8_enable_memory_corruption_api=true'**
- **gn gen out/x64.release --args="v8_no_inline=true is_debug=false v8_enable_memory_corruption_api=true"**
- **tools/dev/gm.py x64.release**
- **tools/dev/gm.py x64.debug**

Install the GEF (GDB Enhanced Features):

- **bash -c "$(curl -fsSL https://gef.blah.cat/sh)"**
- **echo -e '\nsource /home/<user>/articles/chromium/v8/tools/gdbinit' >> ~/.gdbinit**

We will be a slightly different script:

- changing the script and adding a few lines line after the function declaration defining a new variable with the returned value, printing debugging information of associated object, printing the output to the screen, defining an array with the returned value and printing debugging information again.
- opening the d8 program on the gdb debugger: **gdb out/x64.debug/d8**
- running the modified script using the following syntax: **run --allow-natives-syntax --print-bytecode ./script_test.js.** The **--print-bytecode** option does exactly what is expected.
- checking element's content.

```
aborges@ubuntu01:~/articles/chromium/v8$ cat script_test.js


function fact(a){
  if (a <= 1) return 1;
  c = a * fact(a-1);
  o = new Object;
  o.x = c;
  return o.x;
}

let output = fact(4);
%DebugPrint(o);
console.log(output);
array1 = [output];
%DebugPrint(array1);
%SystemBreak();
aborges@ubuntu01:~/articles/chromium/v8$
aborges@ubuntu01:~/articles/chromium/v8$ gdb out/x64.debug/d8


GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
93 commands loaded and 5 functions added for GDB 15.0.50.20240403-git in 0.00ms
using Python engine 3.12
Reading symbols from out/x64.debug/d8...
gef➤  run --allow-natives-syntax ./script_test.js
Starting program: /home/aborges/articles/chromium/v8/out/x64.debug/d8 --allow-na
tives-syntax ./script_test.js

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff70006c0 (LWP 1506057)]
[New Thread 0x7ffff66006c0 (LWP 1506058)]
[New Thread 0x7ffff5c006c0 (LWP 1506059)]
DebugPrint: 0x8fa00048955: [JS_OBJECT_TYPE]
 - map: 0x08fa002990f5 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x08fa00282989 <Object map = 0x8fa00281f95>
 - elements: 0x08fa00000775 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x08fa00000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    0x8fa00002d31: [String] in ReadOnlySpace: #x: 24 (const data field 0, attrs:
[WEC]) @ Any, location: in-object
 }
```

```
0x8fa002990f5: [Map] in OldSpace
 - map: 0x08fa00281a35 <MetaMap (0x08fa00281a85 <NativeContext[300]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 3
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x08fa002827b5 <Map[28](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x08fa00299141 <Cell value= 0>
 - instance descriptors (own) #1: 0x08fa0004891d <DescriptorArray[1]>
 - prototype: 0x08fa00282989 <Object map = 0x8fa00281f95>
 - constructor: 0x08fa002824a9 <JSFunction Object (sfi = 0x8fa00253541)>
 - dependent code: 0x08fa00000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0

24
DebugPrint: 0x8fa0004897d: [JSArray]
 - map: 0x08fa0028c7f1 <Map[16](PACKED_SMI_ELEMENTS)> [FastProperties]
 - prototype: 0x08fa0028ca65 <JSArray[0]>
 - elements: 0x08fa00048971 <FixedArray[1]> [PACKED_SMI_ELEMENTS]
 - length: 1
 - properties: 0x08fa00000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    0x8fa00000dc1: [String] in ReadOnlySpace: #length: 0x08fa0024eab1 <AccessorI
nfo name= 0x08fa00000dc1 <String[6]: #length>, data= 0x08fa00000069 <undefined>>
 (const accessor descriptor, attrs: [W__]), location: descriptor
 }
 - elements: 0x08fa00048971 <FixedArray[1]> {
           0: 24
 }

0x8fa0028c7f1: [Map] in OldSpace
 - map: 0x08fa00281a35 <MetaMap (0x08fa00281a85 <NativeContext[300]>)>
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - unused property fields: 0
 - elements kind: PACKED_SMI_ELEMENTS
 - enum length: invalid
 - back pointer: 0x08fa00000069 <undefined>
 - prototype_validity cell: 0x08fa00000ab1 <Cell value= 1>
 - instance descriptors #1: 0x08fa0028d07d <DescriptorArray[1]>
 - transitions #1: 0x08fa0028d099 <TransitionArray[5]>
   Transition array #1:
     0x08fa00000e85 <Symbol: (elements_transition_symbol)>: (transition to HOLEY
_SMI_ELEMENTS) -> 0x08fa0028d0b5 <Map[16](HOLEY_SMI_ELEMENTS)>
 - prototype: 0x08fa0028ca65 <JSArray[0]>
 - constructor: 0x08fa0028c741 <JSFunction Array (sfi = 0x8fa00253f91)>
 - dependent code: 0x08fa00000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0


Thread 1 "d8" received signal SIGTRAP, Trace/breakpoint trap.
v8::base::OS::DebugBreak () at ../../src/base/platform/platform-posix.cc:772
772     }

[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────
──────── registers ────
$rax   : 0x00007fffffffc4f0  →  0x00005555610a5000  →  0x000008fa00000000  →  0x
0000000000040940
$rbx   : 0x000055555d55fc90  →  <v8::internal::Runtime_SystemBreak(int, unsigned
 long*, v8::internal::Isolate*)+0000> push rbp
```

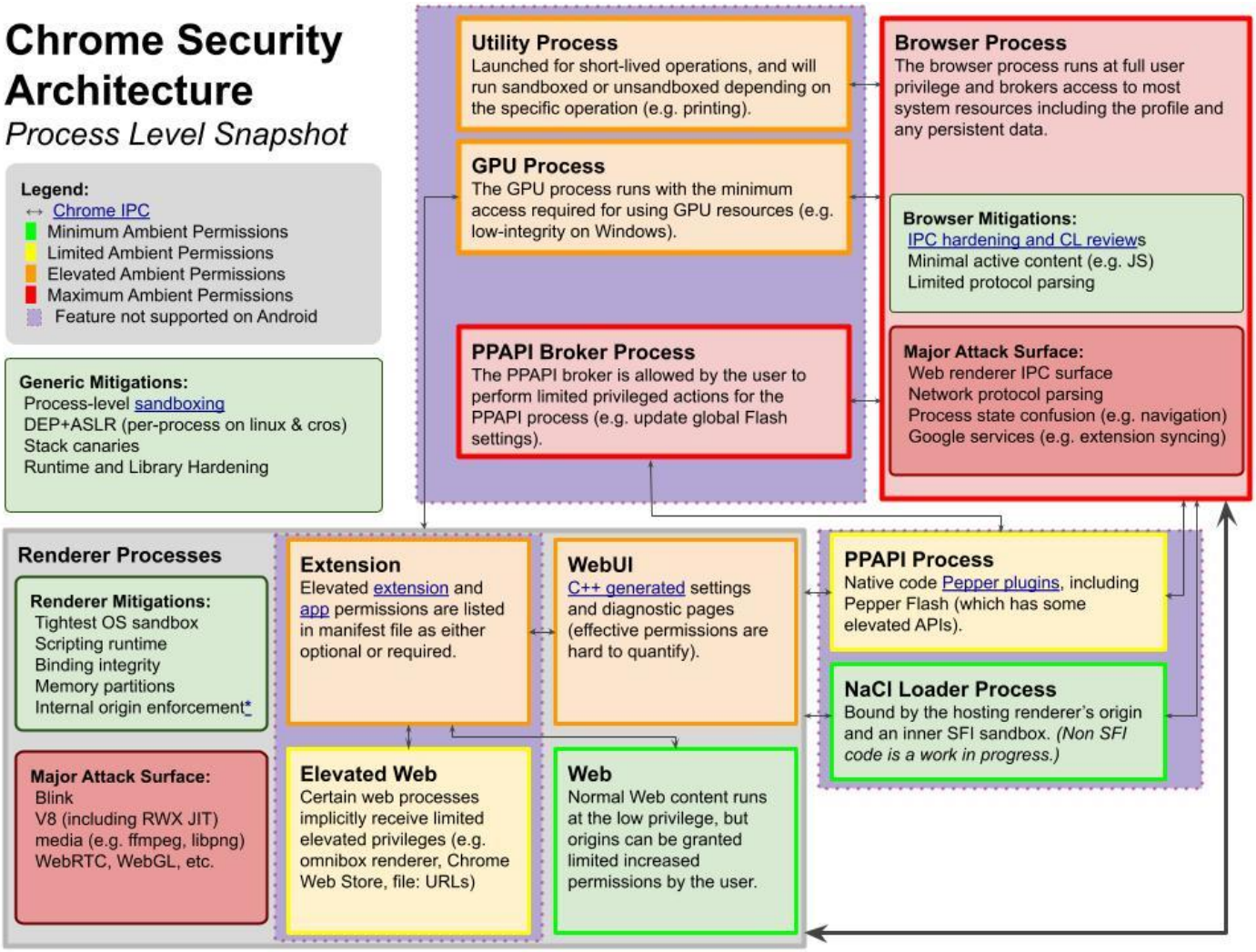
[Figure 04]: debugging the test_script.js using gdb (truncated)

The output is longer than presented, and readers can follow the execution with respective source code, which helps a lot. Additionally, now it is possible to check values in memory through the debugger. We will use this environment in later sections, and you will be able to understand the highlights left on these figures. :)

05. Introduction to Chrome general architecture

Learning about browsers and, in this case, Chrome exploitation represents several challenges that seems hard even though Google offers excellent documentation. We should try to learn concepts in detail, but initially it is wise to read only the necessary topics and practical aspects and, afterwards when we touch security aspects then we can take a step forward in such details.

The **Chromium project** (https://www.chromium.org/Home/) offers the best image to initiate our discussion:



[Figure 05]: Chrome Security Architecture

Readers are able to identify the main critical components and permissions under the security aspect, and each one exerts a key role and contributes to the whole browser security.

There are multiple components and processes that make part of Chrome architecture:

- **Browser process:** controls the chrome browser itself and can be initially interpreted as a general manager of other parts. For this reason, it is the most privileged process in the browser application.
- **Renderer process:** it controls the content of a tab, which is the place where the website is displayed. There are many renderer processes, which each one controls a tab and offers isolation between tabs. The current renderer engine is Blink.
- **Plugin and extension processes:** as expected, they control existing plugins and extensions, and also control their interactions with other components.
- **GPU process:** it manages GPU tasks initiated by applications inside the browser.
- **Utility process:** it is started during short-lived operations and acts as a support-process.

All these processes are under a sandbox and the least privileged security protection, and they communicate to each other through **Interprocess Communication (IPC)** mechanism. Although there are multiple security protections blocking non-authorized operations, all vulnerabilities reported over years, and still in the current days, have shown that is still possible.

One of the most complex components of Chrome is its **JIT (Just-In-Time) engine** (written mostly in C++), which is named V8, and it is responsible for compiling and executing the JavaScript and WebAssembly code that runs in the browser's tabs. Additionally, JIT also takes care about allocation of objects in memory and their respective garbage collection of such objects when they are no longer necessary.

The parsing process of web content by the browser can be summarized in various stages, which were valid until 2021:

- The **Chrome** parser scans the web content, convert it to **DOM (Document Object Model)** representation and the JavaScript code is tokenized.
- Tokens are analyzed, checked against syntax and grammar rules and **the Abstract Syntax Tree (AST)** is generated.
- The **AST tree** is passed on to the **bytecode generator** and interpreter, which is known as **Ignition**.
- In the Ignition stage, which is a register-based interpreter (not real registers, but virtual ones), the AST is transformed into bytecode, received by the interpreter that starts the bytecode execution.
- If there is a piece of the bytecode that is executed frequently, the respective section containing this hot bytecode is passed on to Turbofan.
- **Turbofan** is **another JIT engine** that transforms such bytecode into an optimized machine code, which should run faster.
- If such transformation does not produce an effectively faster code, then such optimized machine code is discarded, and execution returns to the interpreter (Ignition).

Here is a note necessary because render parse the code to transform it to AST, but this is not a direct operation. Actually, this translation is composed of a three-stages operation:

- **Tokenization (lexical analysis):** it parses the JavaScript code and converts it to tokens.
- **Syntax Analysis:** the engine checks the generated tokens to make sure that the syntax is correct.
- **AST:** the parse tree, which also contains information about the grammar, is transformed into AST.

Although I will not refer to these three-steps process from this point onward, readers need to remember that the AST generation is performed as shown here.

As we have seen so far, the V8 processing is based on Ignition (interpreter) and Turbofan, which is an optimizing compiler. Therefore, the general execution flow is:

- Web content → DOM
- JavaScript → Tokens
- Token → AST
- AST → Ignition (interpreter)
- Ignition → bytecode → Turbofan (compiler)
- Turbofan → optimized code (it can be utilized or not, so returning to Ignition).

Since 2021 changes in V8-engine have happened, and the **Sparkplug compiler** has been introduced, and place between Ignition (interpreter) and Turbofan (compiler). The **Sparkplug is a non-optimizing JavaScript compiler and, once again, it is placed between the Ignition interpreter and Turbofan compiler**. In fact, it works as a transpiler, by converting bytecode from Ignition to CPU bytecode (native code):

- Web content → DOM
- JavaScript → Tokens
- Token → AST
- AST → Ignition (interpreter)
- Ignition → bytecode → Sparkplug (non-optimizing compiler | transpiler)
- Sparkplug → Turbofan (compiler)
- Turbofan → optimized code (it can be utilized or not, so returning to Ignition).

The Sparkplug's role is acting as a very-fast and aggressive compiler, which works and optimizes the code based on the existing bytecode instead of starting its processing based on JavaScript code. Additionally, it does not produce intermediate representation like AST. Sparkplug generates little own code, and most of its resulting codes are built-in, which are small pieces of code that are responsible for performing the hard-work through a series of calls and control flow decisions, and that are embedded in a binary

In 2023, Chrome introduced a new optimizing compiler named **Maglev**, which is placed between Sparkplug and Turbofan.

- Web content → DOM
- JavaScript → Tokens
- Token → AST
- AST → Ignition (interpreter)
- Ignition → bytecode → Sparkplug (non-optimizing compiler | transpiler)
- Sparkplug → Maglev (optimizer JIT compiler)
- Maglev → Turbofan (compiler)
- Turbofan → optimized code (it can be utilized or not, so returning to Ignition).

Using simple words, the performance increase offered by the **Sparkplug** is limited, and as the previous starting point (**Ignition + Turbofan**) was low, so the final performance using **Ignition + Sparkplug + Turbofan** is higher but also limited. Using Maglev, **Turbofan** starts at a much higher performance point provided by **Ignition + Sparkplug + Maglev**, reaching a much better performance.

Maglev works examining and collecting information of existing bytecode to find branch targets, loops, and local variable assignments within these loops. Actually, it seems like a meta-information gathering that will

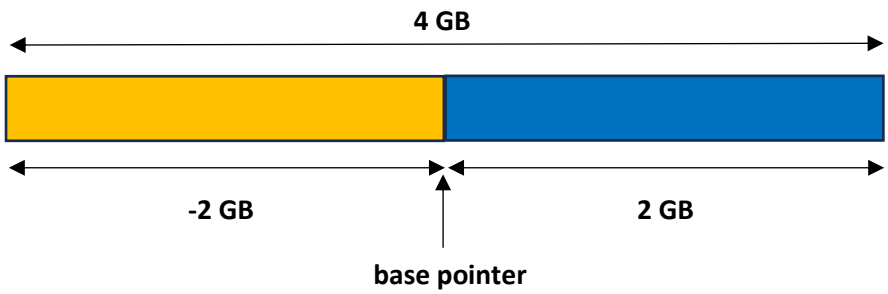
be used for tracking. The saved information will be used for building a graph composed by **SSA nodes (Static Single Assignment Form)**, which is a type of **Intermediary Representation (IR)**, and that each node contains a single instance of a variable assignment that, in this case, is the result of each expression and path evaluation. Every task executed by **Maglev** must be as fast as possible, and efficiently processed to do everything just once. However, to attend this goal, all optimized information received by Maglev needs to be deoptimized and re-evaluated to make possible a better information collection and, as consequence, more effective SSA nodes. Although I have not mentioned, paths and respective variables are also merged and transformed into Phi nodes, where the correct value can be selected depending on the context (taken path).

As Maglev is pursuing the maximum efficiency and performance, representation stored into its SSA nodes is also optimized, and the classical case is when the integer part of numbers is small, V8 engine will encode such numbers as **Smi** (Small Integer | 31-bit tagged integer) to save memory too. We should remember that different JavaScript values are represented as objects and stored into V8 heap, so there is a pointer to such objects, which will be used in memory operations later, and respective results or data will be also saved with V8 pointers on heap through a technique named pointer tagging (a memory pointer with additional data/information associated with it). I will return to this topic later.

The V8 pointer representation differentiates Smi from object pointers using the least significant bit, where the LSB==1 means that the value is a pointer and, obviously, LSB==0 means that the value is a Smi, and such Smi values have 31-bits of payload.

In the case of pointers, there are only 30 bits available for holding the address because its second-least significant bit is used to distinguish strong pointer (indicates that the referenced object is and must remain in memory) from weak ones (indicates that the referenced object might have been deleted). In 64-bit architecture, the context is better because for Smi value its upper 32 bits are used to hold the value, and in case of pointer there are 62 bits for holding the address, which already excludes the two-least significant bits.

The V8 heap layout uses an additional approach that formats and represents both types (Smi and pointer) tagged values using 32 bits even in 64-bit architecture through a technique named **Pointer Compression**, which comes from the fact the all V8 objects are allocated in a 4-GB memory range and such pointers are offset within this 4 GB range. In a few words, the base of such pointers is in the middle of range and, as such pointers are offsets, negative offsets reach up to 2 GB on the left and on the right (readers should imagine an x-axis). Thus, from the farther point on the left to the farther point on the right, we have a total 4 GB heap memory, and this pointer is treated as a signed 32-bit offset from the base:



[Figure 06]: Pointer compression in the V8 heap layout

Once again, I will return to this topic later. Retaking the Maglev discussion, it needs to make a series of decisions in picking the optimal representations for value nodes, where it can unbox values that it knows the respective type and do the entire processing on these unpacked values instead of the original JavaScript numbers. At the end of this cycle, Maglev builds a graph with SSA nodes and does a selection of the appropriate values for processing. To generate the final code, Maglev will make decisions related to register allocation and expressions that will be used to generate the code, which will be really optimized.

At this point in the article, we have a general overview of Chrome architecture and, in particular, one or another concept related to its components. It is the moment to move forward and learn a bit more about internals concepts.

06. Review of internals concepts

V8 engine needs to manage a constant characteristic of the **JavaScript language** that is the fact it is a **dynamic typed language**, which comes from the fact that such type of information comes from the runtime, and it can cause different side effects that depending on the context also generate security issues whether the interpretation is not correct or appropriate. For example, **type confusion bugs** can arise if the Chrome components interpret an array object as a dictionary object, and vice versa. To manage such types, V8 stores this information within a **Map** that is also known as **Hidden Class**, and whose data structures are **Map** (hidden class, which is the first pointer in an object), **DescriptionArray** (it contains information, and a list of properties associated with the class) and **TransitionArray** (it contains edges – properties names -- from the Map to sibling Maps).

Once again, readers can understand Maps as a kind C++ class, where there is a hierarch of classes, with the upper-parent class as the root Map and a series of children classes (Maps) with a parent-child relationship. New **Hidden Maps** arise when new properties that do not exist in their parent classes are added and, as consequence, lower Maps in the hierarch will have more properties associated .

Therefore, an **object** contains a **Map**, a list of **properties** and **elements**, and such a **Map** is regarded as a **hidden class** when arguments of this type are passed to a function. The **Map** itself contains dynamic type of the object, size of the object, properties, a pointer to a prototype, and additional pointers (constructor, descriptor array, and other ones). A relevant detail is that each property associated with **Map** holds the location where such property has been defined (that is the role of the **prototype**), which makes it possible to track a **property** through the entire **chain of Maps** until reaching the root prototype.

Additionally, if the property is marked as **const**, it means that its values have not been changed since its definition (constructor). If you review Figures 2 and 3 previously in this article, you will see different **Maps** and **properties**, where most of them are labeled with the keyword const.

So far, we have reviewed multiple concepts that are valuable inside the V8 context, but we should check effects of such definitions in practical terms and, for V8 in special, it is important to establish a debugging session using **d8 program** that we generated previously.

Before starting, the constant and usual recommendation is to verify whether your environment is updated with new additions from Chrome, which are often, and re-compile v8 when needed. I am using Windows and readers can use another operating system if you want.

```
function Hack(researcher, topic, conference) {
  this.researcher = researcher;
  this.topic = topic;
  this.conference = conference;
  var firstobj = { profile_prop1: topic, profile_prop2: conference };
  var lastobj = { };
  lastobj.topic = this.topic;
  lastobj.conf = this.conference;
  lastobj.order = 1;
}

obj1 = new Hack("Hacker01", "Windows", "Conf_Germany");
obj2 = new Hack("Hacker02", "macOS", "Conf_USA");
obj3 = new Hack("Hacker03", "Hypervisor", "Conf_Canada");
obj4 = new Hack("Hacker04", "Browsers", "Conf_Japan");

const final = { obj1, obj2, obj3, obj4 }
```

[Figure 07]: script_test_02.js

Loading the script we have the following results:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe --allow-natives-syntax
V8 version 13.2.0 (candidate)
d8> load("script_test_02.js")
undefined
d8> %DebugPrint(final)
DebugPrint: 0000038E002891E5: [JS_OBJECT_TYPE]
 - map: 0x038e00099241 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x038e00082795 <Object map = 0000038E00081DA1>
 - elements: 0x038e00000775 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x038e00000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    0000038E00098B41: [String] in OldSpace: #obj1: 0x038e00288ab1 <Hack map = 0000038E00099029> (const
data field 0, attrs: [WEC]) @ Class(0000038E00099029), location: in-object
    0000038E00098B91: [String] in OldSpace: #obj2: 0x038e00288c91 <Hack map = 0000038E00099029> (const
data field 1, attrs: [WEC]) @ Class(0000038E00099029), location: in-object
    0000038E00098BDD: [String] in OldSpace: #obj3: 0x038e00288cf9 <Hack map = 0000038E00099029> (const
data field 2, attrs: [WEC]) @ Class(0000038E00099029), location: in-object
    0000038E00098C31: [String] in OldSpace: #obj4: 0x038e0028917d <Hack map = 0000038E00099029> (const
data field 3, attrs: [WEC]) @ Class(0000038E00099029), location: in-object
 }
0000038E00099241: [Map] in OldSpace
 - map: 0x038e0008183d <MetaMap (0x038e0008188d <NativeContext[301]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 0
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x038e00099219 <Map[28](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x038e00000ab1 <Cell value= 1>
 - instance descriptors (own) #4: 0x038e00289279 <DescriptorArray[4]>
 - prototype: 0x038e00082795 <Object map = 0000038E00081DA1>
 - constructor: 0x038e000822b5 <JSFunction Object (sfi = 0000038E00254A3D)>
 - dependent code: 0x038e00000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0

{obj1: {researcher: "Hacker01", topic: "Windows", conference: "Conf_Germany"}, obj2: {researcher: "Hack
er02", topic: "macOS", conference: "Conf_USA"}, obj3: {researcher: "Hacker03", topic: "Hypervisor", con
ference: "Conf_Canada"}, obj4: {researcher: "Hacker04", topic: "Browsers", conference: "Conf_Japan"}}
d8>
```

[Figure 08]: d8 session (script_test_02.js)

A quick review of the script shows the following points:

- Four objects (**obj1, obj2, obj3** and **obj4**) have been created. Each of these objects is an instance of Hack (clause **new Hack(…)**), and with assigned values to **researcher, topic,** and **conference** properties for each instance. Later such objects will become part of the **final** object.
- **Hack( )** is the constructor responsible for creating objects with three properties: **researcher, topic** and **conference**.
- There is a local variable named **firstobj** that stores an object that contains two properties (**profile_prop1** and **profile_prop2**), which are assigned **topic** and **conference** arguments (values), respectively.
- There is another local variable named **lastobj** the stores an object, which contains **topic, conference** (both holds values passed as arguments) and **1** (assigned to **order** property), respectively.
- The big picture of this simple script is the creation of **Hack objects**, which are combined into the **final** object.

The output offers an apparently complex output, which has multiple keywords and information, as well as a series of addresses. Therefore, my objective here is to provide readers with a summarized explanation of the most important aspects right now, and later we can clear additional details on specific properties in a different context:

- **JS_OBJECT_TYPE:** it indicates that the script is handling with a JavaScript object.
- **map:** as we learned previously, **Map** works like a class (in this case, a hidden class), which describes the structure of the object. Additionally, these maps are used to optimize the access to object properties, and such objects contain the mentioned properties and prototype.
- **HOLEY_ELEMENT:** it indicates an internal representation of an array that contains holes, which are elements that were removed or even existed at that location. In other words, it is like a kind of "sparse array". As expected, the V8 engine needs to manage this array in a distinguished way, and due to that it will be using a differentiated approach that would be used whether the array was **PACKED_ELEMENTS** (no existing holes). Therefore, V8 needs to check for potential holes, and comprehensively this additional task can cause side effects in the performance of the script execution.
- **FastProperties**: it indicates optimization due to frequent access to these properties then they are stored in an efficient and compact format.
- **prototype:** it indicates the prototype of the object, and readers can notice that it points to an object map. A prototype provides a methodology to share properties and even methods among objects. Therefore, each JavaScript object owns a prototype, which declares objects and inherits properties and methods, propagating this inheritance to down and, at the end, establishes a chain of prototypes that can be used to return to the root prototype. This chain is used by a search mechanism, and when a property or method is not found on an object then the engine searches up the prototype chain trying to find it.
- **elements**: it represents the elements storage (**SMI_ELEMENTS, DOUBLE_ELEMENTS, ELEMENTS**) for a given object, which is a **FixedArray**, which is basically a class (actually a **Map** -- that represents a fixed-sized array) containing **HOLEY_ELEMENTS**. More information about it later.
- **own properties:** it makes direct reference to properties of final object, and all of them offer a **Hack instance map**.

- **Oldspace**: it represents the heap memory space where long-live objects (in this case, **final** objects) reside, and it is preserved after repeated garbage collector actions, which are governed by the Mark-Sweep-Compact algorithm.
- **WEC:** they are attributes that mean Writable (W), Enumerable (E) and Configurable(C).
- **NativeContext:** there is a reference to NativeContext, which is a class that represents internal implementation and is used to manage built-in objects and functions.
- **inobject properties:** it represents the number of properties declared within the object.
- **stable_map:** it is used to ensure that a Map (actually, Hidden Map) of an object remains stable after including new properties to such an object, so avoiding creating new hidden classes to represent the new-added property, preventing additional transitions to other and previous maps, and improving the performance.
- **back pointer:** as expected, it points to the previous map and helps to build the map transition chain.

There are other attributes and fields, but I think that the main ones have been quickly explained. Anyway, my first and most valuable recommendation is that readers always must read the source code because most of the time it is well-commented, and it is vital to bring a much more detailed insight into concepts. For example, examining **v8\src\objects\map.h** file, you find the Map definition, as shown below:

```
using MapHandles = std::vector<Handle<Map>>;
using MapHandlesSpan = v8::MemorySpan<Handle<Map>>;

#include "torque-generated/src/objects/map-tq.inc"

// All heap objects have a Map that describes their structure.
//  A Map contains information about:
//  - Size information about the object
//  - How to iterate over an object (for garbage collection)
//
// Map layout:
// +---------------+-----------------------------------------------+
// |   _ Type _    | _ Description _                               |
// +---------------+-----------------------------------------------+
// | TaggedPointer | map - Always a pointer to the MetaMap root    |
// +---------------+-----------------------------------------------+
// | Int           | The first int field                           |
// `---+----------+-----------------------------------------------+
//     | Byte     | [instance_size]                               |
//     +----------+-----------------------------------------------+
//     | Byte     | If Map for a primitive type:                  |
//     |          |   native context index for constructor fn     |
//     |          | If Map for an Object type:                    |
//     |          |   inobject properties start offset in words   |
//     +----------+-----------------------------------------------+
//     | Byte     | [used_or_unused_instance_size_in_words]       |
//     |          | For JSObject in fast mode this byte encodes   |
//     |          | the size of the object that includes only     |
//     |          | the used property fields or the slack size    |
//     |          | in properties backing store.                  |
//     +----------+-----------------------------------------------+
//     | Byte     | [visitor_id]                                  |
// +---+----------+-----------------------------------------------+
// | Int          | The second int field                          |
// `---+----------+-----------------------------------------------+
//     | Short    | [instance_type]                               |
//     +----------+-----------------------------------------------+
//     | Byte     | [bit_field]                                   |
//     |          |   - has_non_instance_prototype (bit 0)        |
//     |          |   - is_callable (bit 1)                       |
//     |          |   - has_named_interceptor (bit 2)             |
//     |          |   - has_indexed_interceptor (bit 3)           |
//     |          |   - is_undetectable (bit 4)                   |
//     |          |   - is_access_check_needed (bit 5)            |
//     |          |   - is_constructor (bit 6)                    |
//     |          |   - has_prototype_slot (bit 7)                |
//     +----------+-----------------------------------------------+
//     | Byte     | [bit_field2]                                  |
//     |          |   - new_target_is_base (bit 0)                |
//     |          |   - is_immutable_proto (bit 1)                |
//     |          |   - elements_kind (bits 2..7)                 |
// +---+----------+-----------------------------------------------+
// | Int          | [bit_field3]                                  |
//     |          |   - enum_length (bit 0..9)                    |
//     |          |   - number_of_own_descriptors (bit 10..19)    |
//     |          |   - is_prototype_map (bit 20)                 |
//     |          |   - is_dictionary_map (bit 21)                |
//     |          |   - owns_descriptors (bit 22)                 |
//     |          |   - is_in_retained_map_list (bit 23)          |
//     |          |   - is_deprecated (bit 24)                    |
//     |          |   - is_unstable (bit 25)                      |
//     |          |   - is_migration_target (bit 26)              |
//     |          |   - is_extensible (bit 28)                    |
//     |          |   - may_have_interesting_properties (bit 28)  |
//     |          |   - construction_counter (bit 29..31)         |
//     |          |                                               |
// +***********************************************************+
// | Int          | On systems with 64bit pointer types, there    |
//     |          | is an unused 32bits after bit_field3          |
// +***********************************************************+
// | TaggedPointer | [prototype]                                  |
// +---------------+----------------------------------------------+
// | TaggedPointer | [constructor_or_back_pointer_or_native_context] |
// +---------------+----------------------------------------------+
// | TaggedPointer | [instance_descriptors]                       |
// +***********************************************************+
// | TaggedPointer | [dependent_code]                             |
// +---------------+----------------------------------------------+
// | TaggedPointer | [prototype_validity_cell]                    |
// +---------------+----------------------------------------------+
// | TaggedPointer | If Map is a prototype map:                   |
//     |          |   [prototype_info]                            |
//     |          | Else:                                         |
//     |          |   [raw_transitions]                           |
// +---------------+----------------------------------------------+
```

[Figure 09]: source code: v8\src\objects\map.h

Naturally, I have omitted and even approximated a series of concepts in my previous explanation because as I used to say, too many details can cause comprehension issues for readers at this point. Readers will learn that the source code is a gold mine because there are tons of concepts, structures and, of course,

code explained. You can continue examining the output related to the final object or even study details of another object such as **obj1, obj2, obj3** and **obj4**. For example, you could want to follow the **backpointe**r and examine the upper **Map**, as shown below:

```
d8> %DebugPrint(0x038e00099219)
DebugPrint: 0000038E0031A97D: [HeapNumber] in OldSpace
 - map: 0x038e0000056d <Map[12](HEAP_NUMBER_TYPE)>
 - value: 3908420866585.0
0000038E0000056D: [Map] in ReadOnlySpace
 - map: 0x038e000004cd <MetaMap (0x038e00000085 <null>)>
 - type: HEAP_NUMBER_TYPE
 - instance size: 12
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - non-extensible
 - back pointer: 0x038e00000069 <undefined>
 - prototype_validity cell: 0
 - instance descriptors (own) #0: 0x038e000007a9 <DescriptorArray[0]>
 - prototype: 0x038e00000085 <null>
 - constructor: 0x038e00000085 <null>
 - dependent code: 0x038e00000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0
```

[Figure 10]: Checking the backpointer

At the same way I show details related to the **obj1 object** below:

```
d8> %DebugPrint(obj1)
DebugPrint: 0000038E00302AF5: [JS_OBJECT_TYPE] in OldSpace
 - map: 0x038e003199f5 <Map[56](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x038e00302ad9 <Object map = 0000038E00319975>
 - elements: 0x038e00000775 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x038e00000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    0000038E0031956D: [String] in OldSpace: #researcher: 0x038e003195c1 <String[8]: #Hacker01> (const d
ata field 0, attrs: [WEC]) @ Any, location: in-object
    0000038E00319585: [String] in OldSpace: #topic: 0x038e003195d5 <String[7]: #Windows> (const data fi
eld 1, attrs: [WEC]) @ Any, location: in-object
    0000038E00319599: [String] in OldSpace: #conference: 0x038e003195e9 <String[12]: #Conf_Germany> (co
nst data field 2, attrs: [WEC]) @ Any, location: in-object
 }
0000038E003199F5: [Map] in OldSpace
 - map: 0x038e00304ab9 <MetaMap (0x038e00304b09 <NativeContext[301]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 56
 - inobject properties: 11
 - unused property fields: 8
 - elements kind: HOLEY_ELEMENTS
 - enum length: 3
 - stable_map
 - back pointer: 0x038e003199cd <Map[56](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x038e003199c5 <Cell value= 0>
 - instance descriptors (own) #3: 0x038e00302b49 <DescriptorArray[3]>
 - prototype: 0x038e00302ad9 <Object map = 0000038E00319975>
 - constructor: 0x038e00319835 <JSFunction Hack (sfi = 0000038E0031977D)>
 - dependent code: 0x038e00000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 6

{researcher: "Hacker01", topic: "Windows", conference: "Conf_Germany"}
```

[Figure 11]: Checking the object obj1

In the first output (**Figure 10**), we see Map in **ReadOnlySpace**, and it does mean in this memory heap there are objects that are not planned to change. The **MetaMap** keyword shows that this is a map of maps,

indicating that it is a high-level structure. Finally, the **HEAP_NUMBER_TYPE** indicates it is associated with **HeapNumber objects**. As I provided you with a quick description of **Fast Property**, which is accessed by querying the **Descriptor Array** (type of storage that stores information on properties of the objects and can be shared between maps to avoid duplication) that is inside the **Hidden Class**, you also need to know that there is the **Slow Property**, which is a type of storage that also allows you to add and remove properties, but it is slower than **Fast Property** (as expected).

In the second output (**Figure 11**) you can clearly see indications of researcher, topic and conference properties, and respective values assigned to them. Furthermore, the **constructor** of the object is well-defined (**Hack**), there is information that is used for other instances (**sfi** means **Shared Function Info**), and it is possible to see how many times it has been called.

Moving a bit away from specific objects, we have discussed and even involved different arrays, and V8 really keeps track of each type of element that such arrays contain because this tracking is necessary to improve the performance and applied approaches to manage each one of these elements. However, V8 engine classify such elements as **SMI_ELEMENTS** (small integers), **DOUBLE_ELEMENTS** (floating-point numbers) and **ELEMENTS** (string literals or even values that cannot be represented by **SMI_ELEMENTS** or **DOUBLE_ELEMENTS**.

We have used only one **d8** option so far that is **--allow-natives-syntax**. There are hundreds of options that can be taken to investigate a wide range of details in V8:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe --help | more
Synopsis:
  shell [options] [--shell] [<file>...]
  d8 [options] [-e <string>] [--shell] [--module|] <file>...]

  -e        execute a string in V8
  --shell   run an interactive JavaScript shell
  --module  execute a file as a JavaScript module
SSE3=1 SSSE3=1 SSE4_1=1 SSE4_2=1 SAHF=1 AVX=1 AVX2=1 AVX_VNNI=0 AVX_VNNI_INT8=0 FMA3=1 F16C=1 BMI1=1 BMI2=1 LZCN
T=1 POPCNT=1 ATOM=0
The following syntax for options is accepted (both '-' and '--' are ok):
  --flag        (bool flags only)
  --no-flag     (bool flags only)
  --flag=value  (non-bool flags only, no spaces around '=')
  --flag value  (non-bool flags only)
  --            (captures all remaining args in JavaScript)

Options:
  --experimental (Indicates that V8 is running with experimental features enabled. This flag is typically not se
t explicitly but instead enabled as an implication of other flags which enable experimental features.)
        type: bool  default: --no-experimental
  --abort-on-contradictory-flags (Disallow flags or implications overriding each other.)
        type: bool  default: --abort-on-contradictory-flags
  --exit-on-contradictory-flags (Exit with return code 0 on contradictory flags.)
        type: bool  default: --no-exit-on-contradictory-flags
  --allow-overwriting-for-next-flag (temporary disable flag contradiction to allow overwriting just the next fla
g)
        type: bool  default: --no-allow-overwriting-for-next-flag
  --use-strict (enforce strict mode)
        type: bool  default: --no-use-strict
  --trace-temporal (trace temporal code)
        type: bool  default: --no-trace-temporal
  --harmony (enable all completed harmony features)
        type: bool  default: --no-harmony
```

[Figure 12]: First options presented by d8.exe (truncated list)

Later I will comment on **sandbox protections**, and as expected, there are options too:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe --help | findstr "\-\-sandbox"
  --sandbox-testing (Enable sandbox testing mode. Useful for demonstrating and validating sandbox byp
asses. This exposes the memory corruption API and enables the sandbox crash filter to filter out cras
hes that do not represent a sandbox bypass)
  --sandbox-fuzzing (Enable sandbox fuzzing mode. This exposes the memory corruption API and enables
the sandbox crash filter, causing all crashes that are not sandbox violations to be ignored.)
```

[Figure 13]: Options starting with --sandbox in d8.exe

At the same form, there are options to control **Turbofan, Sparkplug, Maglev,** and other components. Obviously, it is quite difficult to learn and use most of them in daily work, but soon you will learn the most important ones. For example, there is an interesting **--print-ast** option that, as expected, allows you to print an **Abstract Syntax Tree** of the source code, which is used to analyze and improve the performance of the runtime code:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe --print-ast
V8 version 13.2.0 (candidate)
d8> load("script_test_02.js")
[generating bytecode for function: ]
--- AST ---
FUNC at 0
. KIND 0
. LITERAL ID 0
. SUSPEND COUNT 0
. NAME ""
. INFERRED NAME ""
. EXPRESSION STATEMENT at 0
. . kAssign at -1
. . . VAR PROXY local[0] (0000017409AF6DD0) (mode = TEMPORARY, assigned = true) ".result"
. . . CALL
. . . . VAR PROXY unallocated (0000017409AF6E90) (mode = DYNAMIC_GLOBAL, assigned = false) "load"
. . . . LITERAL "script_test_02.js"
. RETURN at -1
. . VAR PROXY local[0] (0000017409AF6DD0) (mode = TEMPORARY, assigned = true) ".result"

[generating bytecode for function: ]
--- AST ---
FUNC at 0
. KIND 0
. LITERAL ID 0
. SUSPEND COUNT 0
. NAME ""
. INFERRED NAME ""
. DECLS
. . FUNCTION "Hack" = function Hack
. . VARIABLE (0000017409AF7240) (mode = CONST, assigned = false) "final"
. EXPRESSION STATEMENT at 323
. . kAssign at 328
. . . VAR PROXY unallocated (0000017409AF77A8) (mode = DYNAMIC_GLOBAL, assigned = true) "obj1"
. . . CALL NEW at 330
. . . . VAR PROXY unallocated (0000017409AF6EE0) (mode = VAR, assigned = true) "Hack"
. . . . LITERAL "Hacker01"
. . . . LITERAL "Windows"
. . . . LITERAL "Conf_Germany"
. EXPRESSION STATEMENT at 380
. . kAssign at 385
. . . VAR PROXY unallocated (0000017409AF77D8) (mode = DYNAMIC_GLOBAL, assigned = true) "obj2"
. . . CALL NEW at 387
. . . . VAR PROXY unallocated (0000017409AF6EE0) (mode = VAR, assigned = true) "Hack"
. . . . LITERAL "Hacker02"
. . . . LITERAL "macOS"
```

[Figure 14]: AST output (truncated)

The AST output is really long but remember that its content is used by Ignition to generate the bytecode, which is a kind of low-level representation of instructions that can be executed directly. The recommended reading to learn about AST expressions and other definitions is the **v8\src\ast\ast.h file**, which is initially shown below (comments in the source code are always helpful, as usual):

```
1:  // Copyright 2012 the V8 project authors. All rights reserved.
2:  // Use of this source code is governed by a BSD-style license that can be
3:  // found in the LICENSE file.
4:
5:  #ifndef V8_AST_AST_H_
6:  #define V8_AST_AST_H_
7:
8:  #include <memory>
9:
10: #include "src/ast/ast-value-factory.h"
11: #include "src/ast/modules.h"
12: #include "src/ast/variables.h"
13: #include "src/base/pointer-with-payload.h"
14: #include "src/base/threaded-list.h"
15: #include "src/codegen/bailout-reason.h"
16: #include "src/codegen/handler-table.h"
17: #include "src/codegen/label.h"
18: #include "src/common/globals.h"
19: #include "src/heap/factory.h"
20: #include "src/objects/elements-kind.h"
21: #include "src/objects/function-syntax-kind.h"
22: #include "src/objects/literal-objects.h"
23: #include "src/objects/shared-function-info.h"
24: #include "src/objects/smi.h"
25: #include "src/parsing/token.h"
26: #include "src/runtime/runtime.h"
27: #include "src/zone/zone-list.h"
28:
29: namespace v8 {
30: namespace internal {
31:
32: // The abstract syntax tree is an intermediate, light-weight
33: // representation of the parsed JavaScript code suitable for
34: // compilation to native code.
35:
36: // Nodes are allocated in a separate zone, which allows faster
37: // allocation and constant-time deallocation of the entire syntax
38: // tree.
39:
40:
41: // ----------------------------------------------------------------------------
42: // Nodes of the abstract syntax tree. Only concrete classes are
43: // enumerated here.
44:
45: #define DECLARATION_NODE_LIST(V) \
46:   V(VariableDeclaration)         \
47:   V(FunctionDeclaration)
48:
49: #define ITERATION_NODE_LIST(V) \
50:   V(DoWhileStatement)          \
```

[Figure 15]: ast.h file

Before we proceed, there is a key fact that I did not have the opportunity to explain yet, but it can be useful to understand a missing piece of the puzzle. In V8 engine, we have **transitions**, which are used when a map changes due to the addition or removal of a property. In other words, the structure of an object transits from the previous version to the new one, which demands a new **Map (Hidden Map)** because it was not possible to keep the old map. This new map is used while optimizing access to the object and its structures. Transitions appeared at the beginning of this article (check Figure 04). Now readers have all needed concepts to analyze and also understand the first pages when I showed an example while I described the set up and configuration of the environment.

Once again, returning to the main topic, we have seen how to use the **gdb** on Linux to debug a **d8 session** previously, and now I am going to use **WinDbg** and attach it to the **d8.exe process**, as shown below:

```
0:046> k
 # Child-SP          RetAddr               Call Site
00 00000003`569ff8b8 00007fff`c7fa735e     ntdll!DbgBreakPoint
01 00000003`569ff8c0 00007fff`c6be259d     ntdll!DbgUiRemoteBreakin+0x4e
02 00000003`569ff8f0 00007fff`c7f2af38     KERNEL32!BaseThreadInitThunk+0x1d
03 00000003`569ff920 00000000`00000000     ntdll!RtlUserThreadStart+0x28
0:046> dq 0000038E00303041
0000038e`00303041  75000007`7500319b 7d00302a`f5000007
0000038e`00303051  0900302b`b500302b 04000006`d5003030
0000038e`00303061  bd000000`11000400 84003195`b10031a3
0000038e`00303071  01003199`f7000009 f7001005`84003196
0000038e`00303081  84003196`4d003199 a1003199`f700200d
0000038e`00303091  f7003001`84003196 32000001`0d003199
0000038e`003030a1  0a00000a`dc00bab9 6f697463`6e756628
0000038e`003030b1  75220a7b`2029286e 63697274`73206573
0:046> dq 0000038E00303040
0000038e`00303040  00000775`00319b25 00302af5`00000775
0000038e`00303050  00302bb5`00302b7d 000006d5`00303009
0000038e`00303060  00000011`00040004 003195b1`0031a3bd
0000038e`00303070  003199f7`00000984 00100584`00319601
0000038e`00303080  0031964d`003199f7 003199f7`00200d84
0000038e`00303090  00300184`003196a1 0000010d`003199f7
0000038e`003030a0  00000adc`00bab932 6974636e`7566280a
0000038e`003030b0  220a7b20`29286e6f 69727473`20657375

d8> %DebugPrint(final)
DebugPrint: 0000038E00303041: [JS_OBJECT_TYPE] in OldSpace
 - map: 0x038e00319b25 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x038e003059ad <Object map = 0000038E00304FCD>
 - elements: 0x038e00000775 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x038e00000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    0000038E003195B1: [String] in OldSpace: #obj1: 0x038e00302af5 <Hack map = 0000038E003
199F5> (const data field 0, attrs: [WEC]) @ Class(0000038E003199F5), location: in-object
    0000038E00319601: [String] in OldSpace: #obj2: 0x038e00302b7d <Hack map = 0000038E003
199F5> (const data field 1, attrs: [WEC]) @ Class(0000038E003199F5), location: in-object
    0000038E0031964D: [String] in OldSpace: #obj3: 0x038e00302bb5 <Hack map = 0000038E003
199F5> (const data field 2, attrs: [WEC]) @ Class(0000038E003199F5), location: in-object
    0000038E003196A1: [String] in OldSpace: #obj4: 0x038e00303009 <Hack map = 0000038E003
199F5> (const data field 3, attrs: [WEC]) @ Class(0000038E003199F5), location: in-object
 }
0000038E00319B25: [Map] in OldSpace
 - map: 0x038e00304ab9 <MetaMap (0x038e00304b09 <NativeContext[301]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 0
 - elements kind: HOLEY_ELEMENTS
 - enum length: 4
 - stable_map
 - back pointer: 0x038e00319afd <Map[28](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x038e00000ab1 <Cell value= 1>
 - instance descriptors (own) #4: 0x038e0030305d <DescriptorArray[4]>
 - prototype: 0x038e003059ad <Object map = 0000038E00304FCD>
 - constructor: 0x038e003054e1 <JSFunction Object (sfi = 0000038E00254A3D)>
 - dependent code: 0x038e00000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0
```

[Figure 16]: d8 and respective WinDbg debugging session

The following observations can be made by taking both outputs from the last page:

- I have run the **%DebugPrint(final)** command and attached the WinDbg to the **d8.exe** process.
- V8 uses **Pointer Tagging**, which used the two LSB (Least Significant Bit) to indicate whether the value is **data/SMI (0)** or a **heap pointer (1)**.
- A **pointer** can be a **strong (end with 00)**, which points to an object that must remain in memory, or a **weak (ends with 11),** which points to an object that might not be in memory anymore.
- To prevent being similar to a pointer, **SMI are stored doubled**.
- The first **dq** command (**dq 0000038E00303041**) did not work. Ending 1 indicates that it is the given address is a pointer, so we need to subtract the address by 1, as we have done in the next command (**dq 0000038E00303040**)
- Only part of the pointer's values is shown in WinDbg due to **Pointer Compression** feature implemented by V8 engine. Basically, the first part of each pointer is common to all objects because they are close to one another other on memory.

The **V8 Pointer Compression** is implemented in **v8\common\ptr-comp-inl.h** file and referenced in v8-**internal.h** file:

```
#ifndef V8_COMMON_PTR_COMPR_INL_H_
#define V8_COMMON_PTR_COMPR_INL_H_

#include "include/v8-internal.h"
#include "src/execution/isolate.h"
#include "src/execution/local-isolate-inl.h"

namespace v8 {
namespace internal {

#ifdef V8_COMPRESS_POINTERS

PtrComprCageBase::PtrComprCageBase(const Isolate* isolate)
    : address_(isolate->cage_base()) {}
PtrComprCageBase::PtrComprCageBase(const LocalIsolate* isolate)
    : address_(isolate->cage_base()) {}

//
// V8HeapCompressionSchemeImpl
//

constexpr Address kPtrComprCageBaseMask = ~(kPtrComprCageBaseAlignment - 1);

// static
template <typename Cage>
constexpr Address V8HeapCompressionSchemeImpl<Cage>::GetPtrComprCageBaseAddress(
    Address on_heap_addr) {
  return RoundDown<kPtrComprCageBaseAlignment>(on_heap_addr);
}

// static
template <typename Cage>
Address V8HeapCompressionSchemeImpl<Cage>::GetPtrComprCageBaseAddress(
    PtrComprCageBase cage_base) {
  Address base = cage_base.address();
  V8_ASSUME((base & kPtrComprCageBaseMask) == base);
  base = reinterpret_cast<Address>(V8_ASSUME_ALIGNED(
      reinterpret_cast<void*>(base), kPtrComprCageBaseAlignment));
  return base;
}
```

[Figure 17]: Pointer Compression implementation in v8\common\ptr-compr-inl.h file.

Additionally, the V8 source code (**v8\src\object\tagged.h**) also provides us with a complete explanation about **Tagged Pointers**, including multiple details:

```
// Copyright 2023 the V8 project authors. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.

#ifndef V8_OBJECTS_TAGGED_H_
#define V8_OBJECTS_TAGGED_H_

#include <type_traits>

#include "src/common/globals.h"
#include "src/objects/tagged-impl.h"
#include "src/objects/union.h"

namespace v8 {
namespace internal {

class BigInt;
class FieldType;
class HeapObject;
class HeapNumber;
class HeapObjectLayout;
class TrustedObject;
class TrustedObjectLayout;
class Object;
class TaggedIndex;
class Smi;

// Tagged<T> represents an uncompressed V8 tagged pointer.
//
// The tagged pointer is a pointer-sized value with a tag in the LSB. The value
// is either:
//
//   * A small integer (Smi), shifted right, with the tag set to 0
//   * A strong pointer to an object on the V8 heap, with the tag set to 01
//   * A weak pointer to an object on the V8 heap, with the tag set to 11
//   * A cleared weak pointer, with the value 11
//
// The exact encoding differs depending on 32- vs 64-bit architectures, and in
// the latter case, whether or not pointer compression is enabled.
//
// On 32-bit architectures, this is:
//              |----- 32 bits -----|
// Pointer:     |______address____w1|
//    Smi:      |____int31_value___0|
//
// On 64-bit architectures with pointer compression:
//              |----- 32 bits -----|----- 32 bits -----|
// Pointer:     |________base_______|______offset_____w1|
//    Smi:      |......garbage......|____int31_value___0|
//
// On 64-bit architectures without pointer compression:
//              |----- 32 bits -----|----- 32 bits -----|
// Pointer:     |_________________address_______________w1|
//    Smi:      |____int32_value____|00..............00|
//
// where `w` is the "weak" bit.
//
// We specialise Tagged separately for Object, Smi and HeapObject, and then all
// other types T, so that:
//
//                     Tagged<Object> -> StrongTaggedBase
//                       Tagged<Smi> -> StrongTaggedBase
//   Tagged<T> -> Tagged<HeapObject> -> StrongTaggedBase
```

[Figure 18]: Tagged Pointers (v8\src\object\tagged.h)

From the documentation above, **pointer compression** takes the **first 32 bits** as base and the **next 30 bits** (excluding the lowest two bits) as **offset**. Additionally, the **strong pointer** ends with **01** while **weak pointers** end with **11**.

Now we have some basic elements, and we can move forward.

Remember that a basic representation of data flow in Chrome is:

- Web content → DOM
- JavaScript → Tokens
- Token → AST
- AST → Ignition (interpreter)
- Ignition → bytecode → Sparkplug (non-optimizing compiler | transpiler)
- Sparkplug → Maglev (optimizer JIT compiler)
- Maglev → Turbofan (compiler)
- Turbofan → optimized code (it can be utilized or not, so returning to Ignition).

When a user visits a website, which is composed by static and dynamic content, the dynamic part (JavaScript) is extracted from the html, converted to **DOM (Document Object Model)** representation and the JavaScript content is tokenized, syntactically and grammarly checked and finally converted to AST. Events related to this process can be followed by using **--log-function-events** (parse, compile, execute):

```
C:\articles\chromium\v8>out\x64.debug\d8.exe script_test_02.js --log-function-events

C:\articles\chromium\v8>type v8.log
v8-version,13,2,0,0,1
v8-platform,windows,unknown
new,CodeRange,0x7ff6e0000000,0
new,MemoryChunk,0x31e00000000,262144
new,MemoryChunk,0x31e00180000,262144
new,MemoryChunk,0x31e001c0000,262144
new,MemoryChunk,0x31e00200000,262144
new,MemoryChunk,0x31e00240000,262144
new,MemoryChunk,0x31e00380000,262144
new,MemoryChunk,0x31e00280000,262144
new,MemoryChunk,0x31e002c0000,262144
new,MemoryChunk,0x31e00300000,262144
new,MemoryChunk,0x31e00340000,262144
heap-capacity,1048512
heap-available,4295753600
new,MemoryChunk,0x31e00040000,262144
script,deserialize,1,25090
script-details,1,,0,0,
new,MemoryChunk,0x21700000000,262144
script,deserialize,2,26646
script-details,2,,0,0,
new,MemoryChunk,0x31e00080000,262144
script,reserve-id,3,81125
script,create,3,81200
script-details,3,script_test_02.js,0,0,
function,preparse-no-resolution,3,13,319,0.164,81465,Hack
function,parse-script,3,0,596,0.515,81781,
new,MemoryChunk,0x21700040000,262144
function,interpreter,3,0,596,0.641,82666,
compilation-cache,put,script,3,0,596,82707
function,first-execution,3,0,596,0,82979,
function,full-parse,3,13,319,0.108,83354,Hack
function,parse-function,3,13,319,0.201,83431,Hack
function,interpreter,3,13,319,0.128,83780,Hack
function,first-execution,3,13,319,0,84339,Hack
delete,MemoryChunk,0x26931bf5e80
delete,MemoryChunk,0x26931bf8010
delete,MemoryChunk,0x26931bfa1a0
delete,MemoryChunk,0x26931bfc330
delete,MemoryChunk,0x26931c15220
delete,MemoryChunk,0x26931c20e20
delete,MemoryChunk,0x26931c173b0
delete,MemoryChunk,0x26931c0d5e0
```

[Figure 19]: Logging parse, compile and execute events

I have commented about options that can be used with **d8.exe debugger** in previous pages, and there are two options classes that usually can be useful, that is **--log** and **--print**, and I highlighted some of them:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe --help | findstr "\-\-log"
  --log-or-trace-osr (internal helper flag, please use --trace-osr instead.)
  --log-deopt (log deoptimization)
  --log-colour (When logging, try to use coloured output.)
  --log-ic (Log inline cache state transitions for tools/ic-processor)
  --log-maps (Log map creation)
  --log-maps-details (Also log map details)
        type: bool  default: --log-maps-details
  --logfile (Specify the name of the log file, use '-' for console, '+' for a temporary file.)
        type: string  default: --logfile="v8.log"
  --logfile-per-isolate (Separate log files for each isolate.)
        type: bool  default: --logfile-per-isolate
  --log (Minimal logging (no API, code, GC, suspect, or handles samples).)
  --log-all (Log all events to the log file.)
  --log-internal-timer-events (See --log-timer-events)
  --log-timer-events (Log timer events (incl. console.time* and Date.now).)
  --log-source-code (Log source code.)
  --log-source-position (Log detailed source information.)
  --log-code (Log code events to the log file without profiling.)
  --log-feedback-vector (Log FeedbackVectors on first creation)
  --log-code-disassemble (Log all disassembled code to the log file.)
  --log-function-events (Log function events (parse, compile, execute) separately.)
  --prof (Log statistical profiling information (implies --log-code).)

C:\articles\chromium\v8>out\x64.debug\d8.exe --help | findstr "\-\-print"
  --print-maglev-code (print maglev code)
  --print-maglev-deopt-verbose (print verbose deopt info)
  --print-maglev-graph (print the final maglev graph)
  --print-maglev-graphs (print maglev graph across all phases)
  --print-bytecode (print bytecode generated by ignition interpreter)
  --print-bytecode-filter (filter for selecting which functions to print bytecode)
        type: string  default: --print-bytecode-filter="*"
  --print-deopt-stress (print number of possible deopt points)
  --print-wasm-code (print WebAssembly code)
  --print-wasm-code-function-index (print WebAssembly code for function at index)
        type: int  default: --print-wasm-code-function-index=-1
  --print-wasm-stub-code (print WebAssembly stub code)
  --print-all-exceptions (print exception object and stack trace on each thrown exception)
  --print-flag-values (Print all flag values of V8)
  --print-ast (print source AST)
  --print-scopes (print scopes)
  --print-handles (report handles after GC)
  --print-global-handles (report global handles after GC)
  --print-break-location (print source location on debug break)
  --print-opt-source (print source code of optimized and inlined functions)
  --print-code (print generated code)
  --print-opt-code (print optimized code)
  --print-opt-code-filter (filter for printing optimized code)
        type: string  default: --print-opt-code-filter="*"
  --print-code-verbose (print more information for code)
  --print-builtin-code (print generated code for builtins)
  --print-builtin-code-filter (filter for printing builtin code)
        type: string  default: --print-builtin-code-filter="*"
  --print-regexp-code (print generated regexp code)
  --print-regexp-bytecode (print generated regexp bytecode)
  --print-builtin-size (print code size for builtins)
  --print-all-code (enable all flags related to printing code)
```

[Figure 19]: Logging parse, compile and execute events

Using two of the exposed options, we can check the generated bytecode of the **Hack function**, as shown below:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe script_test_02.js --print-bytecode --print-bytecode-filter=Hack
[generated bytecode for function: Hack (0x016700098ba5 <SharedFunctionInfo Hack>)]
Bytecode length: 64
Parameter count 4
Register count 3
Frame size 24
         0000026F00040184 @    0 : 0b 03             Ldar a0
         0000026F00040186 @    2 : 39 02 00 00       SetNamedProperty <this>, [0], [0]
         0000026F0004018A @    6 : 0b 04             Ldar a1
         0000026F0004018C @    8 : 39 02 01 02       SetNamedProperty <this>, [1], [2]
         0000026F00040190 @   12 : 0b 05             Ldar a2
         0000026F00040192 @   14 : 39 02 02 04       SetNamedProperty <this>, [2], [4]
         0000026F00040196 @   18 : 85 03 06 29       CreateObjectLiteral [3], [6], #41
         0000026F0004019A @   22 : cc                Star2
         0000026F0004019B @   23 : 0b 04             Ldar a1
         0000026F0004019D @   25 : 3a f7 04 07       DefineNamedOwnProperty r2, [4], [7]
         0000026F000401A1 @   29 : 0b 05             Ldar a2
         0000026F000401A3 @   31 : 3a f7 05 09       DefineNamedOwnProperty r2, [5], [9]
         0000026F000401A7 @   35 : 1b f7 f9          Mov r2, r0
         0000026F000401AA @   38 : 86                CreateEmptyObjectLiteral
         0000026F000401AB @   39 : cd                Star1
         0000026F000401AC @   40 : 33 02 01 0b       GetNamedProperty <this>, [1], [11]
         0000026F000401B0 @   44 : 39 f8 01 0d       SetNamedProperty r1, [1], [13]
         0000026F000401B4 @   48 : 33 02 02 0f       GetNamedProperty <this>, [2], [15]
         0000026F000401B8 @   52 : 39 f8 06 11       SetNamedProperty r1, [6], [17]
         0000026F000401BC @   56 : 0d 01             LdaSmi [1]
         0000026F000401BE @   58 : 39 f8 07 13       SetNamedProperty r1, [7], [19]
         0000026F000401C2 @   62 : 0e                LdaUndefined
         0000026F000401C3 @   63 : b3                Return
Constant pool (size = 8)
0000026F00040135: [TrustedFixedArray]
 - map: 0x0167000005e5 <Map(TRUSTED_FIXED_ARRAY_TYPE)>
 - length: 8
           0: 0x016700098960 <String[10]: #researcher>
           1: 0x016700098985 <String[5]: #topic>
           2: 0x016700098999 <String[10]: #conference>
           3: 0x016700098d7d <ObjectBoilerplateDescription[4]>
           4: 0x016700098cf9 <String[13]: #profile_prop1>
           5: 0x016700098d15 <String[13]: #profile_prop2>
           6: 0x016700098d31 <String[4]: #conf>
           7: 0x016700098d41 <String[5]: #order>
Handler Table (size = 0)
Source Position Table (size = 0)
```

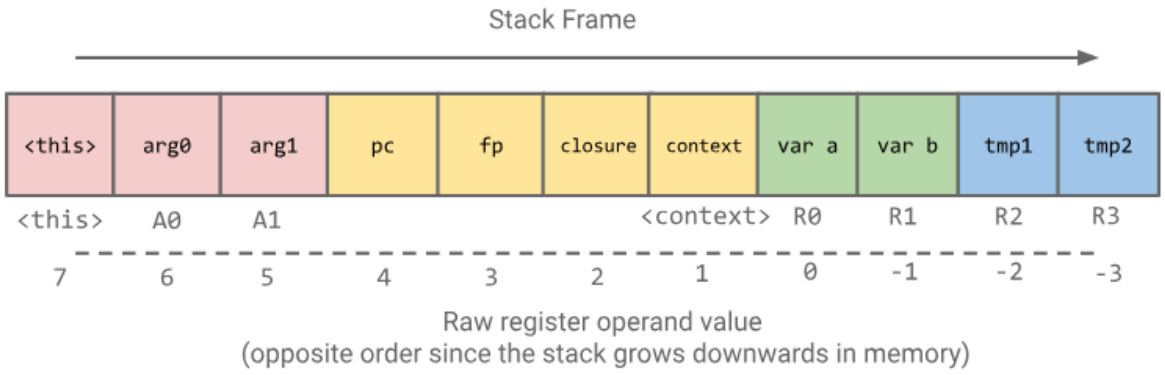
[Figure 20]: Generated bytecodes for Hack function

The code above is not complicated to understand, even though there are many details that I will leave out of the current discussion. Anyway, before explaining it, a few definitions and concepts are needed.

There is an accumulator register, which does not make part of the register file, but readers will notice that it will be used in multiple situations, mitigating the number of memory operations (loading and storing register content). The group of instructions involved with bytecode are:

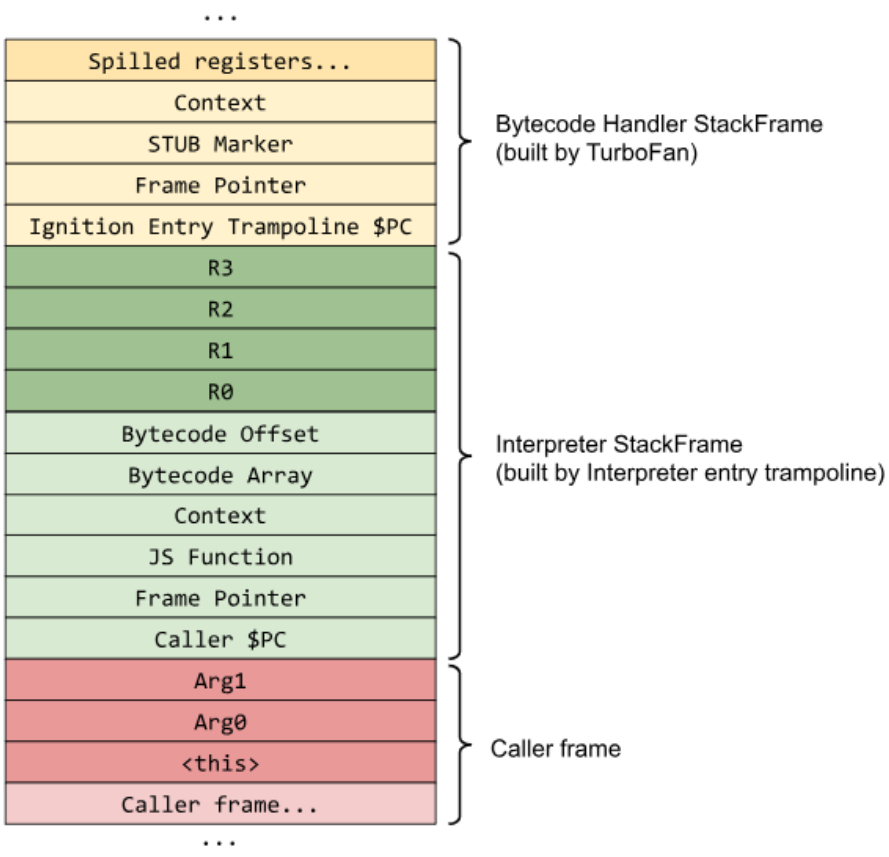
- **Accumulator Load:** LdaSmi, LdaConstant, LdaTrue, LdaFalse and so on.
- **Load/Store Globals:** LdaGlobal
- **Context Operations:** PushContext, PopContext, LdaContextSlot, StaContextSlot
- **Register Transfers:** Ldar, Mov, Star
- **ControlFlow**: Jump, JumpConstant

There are many other instructions from other groups, but either they are easy to understand, or they are extremely specific, so they are not necessary here. In our sample script, we have a function named Hack( ), and the Ignition allocates registers to the **register file of this function** for its local variables, and also includes **context object pointers** and **needed temporary values**. Therefore, registers are not real registers, but special slots in the register file, which is allocated during the prologue to the function's stack frame. As the Ignition generates bytecodes from the AST (actually, for each AST node), it also offers a set of **bytecode handlers** that operate on these registers, which are operated using indexes that are mapped to the stack frame of the function. A simplified picture representing the function's stack frame follows below:



[Figure 21]: Stack frame 01 | Ignition Design Doc (credit: Google)

Another and more complete visualization of the stack frame associated with the Ignition interpreter is shown below:



[Figure 22]: Stack frame 02 | Ignition Design Doc (credit: Google)

As the most generated code involves constant values and strings, they are kept in **a constant pool**, which is stored in slots in the heap, and references as necessary by the bytecode, but other types such as **Smi integers** are not stored in this constant. Additionally, in different moments, readers have seen a reference to "**context**", and basically, they are used to keep state across function closures (pointer to the provided function, which is a **JSFunction object**, and that putting everything necessary together in the same context) and, as expected, they are considered during optimizations. The interpreter is also able to track the sequence of contexts (context chain) that are usually involved when there are nested functions. Additionally, the need for temporary values that have been mentioned, and they are required only for expression evaluation and also valid while the expression exists.

A succinct explanation of instructions from **Figure 20** follows:

- **Ldar a0**: it loads the value given by a0 argument into the accumulator.
- **SetNamedProperty <this>, [0], [0]:** it sets the named property [0] given **by this pointer** to the value stored in the accumulator. The first [0] is an **index to the name of the property in the constant pool**, which as explained, keeps a list of constants (numbers and strings). Thus, the [**0] clearly refers to the position/slot of the referred constant**. The **second [0] is used for optimization**.
- **Ldar a1:** it loads the value given by the argument **a1** into the accumulator.
- **SetNamedProperty <this>, [1], [2]:** it sets the named property [1] given by **this pointer** to the value stored in the accumulator.
- **Ldar a2**: it loads the value given by the argument a2 into the accumulator.
- **SetNamedProperty <this>, [2], [4]:** it sets the named property [2] of the object referred by this pointer to the value provided by the accumulator.
- **CreateObjectLiteral [3], [6], #41**: it creates an object literal set to value 41, with the specified properties, and store It in the accumulator.
- **Star2:** it stores the value in the accumulator into the second temporary register, which is given by r2.
- **Ldar a1:** it loads the value in argument a1 into the accumulator.
- **DefineNamedOwnProperty r2, [4], [7]:** it defines a named property [4] on the object given by r2 with the value in the accumulator and property attributes provided by [7].
- **Ldar a2:** it loads the value in argument a2 into the accumulator.
- **DefineNamedOwnProperty r2, [5], [9]:** it defines a named own property [5] on the object given by r2 with the value provided by the accumulator and property attribute [9].
- **Mov r2, r0:** it moves value from register r0 to register r2.
- **CreateEmptyObjectLiteral:** it creates an empty object literal and stores it in the accumulator.
- **Star1:** it stores the value in the accumulator into the first temporary register, which is r1.
- **GetNamedProperty <this>, [1], [11]:** it gets the named property [1] from the object referenced by this pointer and store it into the accumulator.
- **SetNamedProperty r1, [1], [13]:** it sets the named property [1] on the object given by r1 to the value provided by the accumulator.
- **GetNamedProperty <this>, [2], [15]:** it gets the named property [2] from the object referenced by this pointer and stores it into the accumulator.
- **SetNamedProperty r1, [6], [17]:** it sets the named property [6] on the object given by r1 to the value provided by accumulator.
- **LdaSmi [1]:** it loads the small integer [1] into the accumulator.

- **SetNamedProperty r1, [7], [19]:** it sets the named property [7] on the object given by r1 to the value provided by the accumulator.
- **LdaUndefined:** it loads an undefined value into the accumulator.
- **Return:** it returns the value stored in the accumulator.

As readers have noticed so far, most of Chrome V8 (in general) are based on objects, where such objects can be different things such as arrays, strings, integers and other "types", and there is a whole **Object class** that starts an extensive hierarchy, but without having C++ vtables involved, as shown in **src\objects\objects.h**:

```
121: // Object is the abstract superclass for all classes in the
122: // object hierarchy.
123: // Object does not use any virtual functions to avoid the
124: // allocation of the C++ vtable.
125: // There must only be a single data member in Object: the Address ptr,
126: // containing the tagged heap pointer that this Object instance refers to.
127: // For a design overview, see https://goo.gl/Ph4CGz.
128: class Object : public AllStatic {
129:  public:
130:   enum class Conversion {
131:     kToNumber,  // Number = Smi or HeapNumber
132:     kToNumeric  // Numeric = Smi or HeapNumber or BigInt
133:   };
134:
135:   // ES6, #sec-isarray.  NOT to be confused with %_IsArray.
136:   V8_INLINE
137:   V8_WARN_UNUSED_RESULT static Maybe<bool> IsArray(Handle<Object> object);
138:
139:   // Extract the double value of a Number (Smi or HeapNumber).
140:   static inline double NumberValue(Tagged<Number> obj);
141:   static inline double NumberValue(Tagged<Object> obj);
142:   static inline double NumberValue(Tagged<HeapNumber> obj);
143:   static inline double NumberValue(Tagged<Smi> obj);
144:   V8_EXPORT_PRIVATE static bool ToInt32(Tagged<Object> obj, int32_t* value);
145:   static inline bool ToUint32(Tagged<Object> obj, uint32_t* value);
146:
147:   static inline Representation OptimalRepresentation(
148:       Tagged<Object> obj, PtrComprCageBase cage_base);
149:
150:   static inline ElementsKind OptimalElementsKind(Tagged<Object> obj,
151:                                                  PtrComprCageBase cage_base);
152:
153:   // If {allow_coercion} is true, then a Smi will be considered to fit
154:   // a Double representation, since it can be converted to a HeapNumber
155:   // and stored.
156:   static inline bool FitsRepresentation(Tagged<Object> obj,
157:                                         Representation representation,
158:                                         bool allow_coercion = true);
```

[Figure 23]: src/objects/objecs.h file | Object Class definition

Actually, the engine reacts to the declaration of content, and depending on such content and its boundaries, the chosen object is different as, for example, the choice between small integers (smi), doubles and pointers, which adds 1 to differentiate from data (memory tagging) and, mainly, saves memory because it makes possible to store a 64 bits value (pointer) into a 32 bit (memory heap). As you have probably understood from previous JavaScript executions, and similarly we have in other addresses spaces from programs, there are stack and heap, and this one is subdivided into different segments (as known as sections) such as code, new space, old pointers space, old data space and large object space heap memory.

Furthermore, each object can be composed of one or more types of elements, as shown in the next figure:

```
105: enum ElementsKind : uint8_t {
106:   // The "fast" kind for elements that only contain SMI values. Must be first
107:   // to make it possible to efficiently check maps for this kind.
108:   PACKED_SMI_ELEMENTS,
109:   HOLEY_SMI_ELEMENTS,
110:
111:   // The "fast" kind for tagged values. Must be second to make it possible to
112:   // efficiently check maps for this and the PACKED_SMI_ELEMENTS kind
113:   // together at once.
114:   PACKED_ELEMENTS,
115:   HOLEY_ELEMENTS,
116:
117:   // The "fast" kind for unwrapped, non-tagged double values.
118:   PACKED_DOUBLE_ELEMENTS,
119:   HOLEY_DOUBLE_ELEMENTS,
120:
121:   // The nonextensible kind for elements.
122:   PACKED_NONEXTENSIBLE_ELEMENTS,
123:   HOLEY_NONEXTENSIBLE_ELEMENTS,
124:
125:   // The sealed kind for elements.
126:   PACKED_SEALED_ELEMENTS,
127:   HOLEY_SEALED_ELEMENTS,
128:
129:   // The frozen kind for elements.
130:   PACKED_FROZEN_ELEMENTS,
131:   HOLEY_FROZEN_ELEMENTS,
132:
133:   // SharedArray elements kind. A FAST_SEALED_ELEMENTS variation useful to
134:   // code specific paths for SharedArrays.
135:   SHARED_ARRAY_ELEMENTS,
136:
137:   // The "slow" kind.
138:   DICTIONARY_ELEMENTS,
139:
140:   // Elements kind of the "arguments" object (only in sloppy mode).
141:   FAST_SLOPPY_ARGUMENTS_ELEMENTS,
142:   SLOW_SLOPPY_ARGUMENTS_ELEMENTS,
143:
144:   // For string wrapper objects ("new String('...')"), the string's characters
145:   // are overlaid onto a regular elements backing store.
146:   FAST_STRING_WRAPPER_ELEMENTS,
147:   SLOW_STRING_WRAPPER_ELEMENTS,
148:
149: // Fixed typed arrays.
150: #define TYPED_ARRAY_ELEMENTS_KIND(Type, type, TYPE, ctype) TYPE##_ELEMENTS,
151:   TYPED_ARRAYS(TYPED_ARRAY_ELEMENTS_KIND)
152:       RAB_GSAB_TYPED_ARRAYS(TYPED_ARRAY_ELEMENTS_KIND)
153: #undef TYPED_ARRAY_ELEMENTS_KIND
```

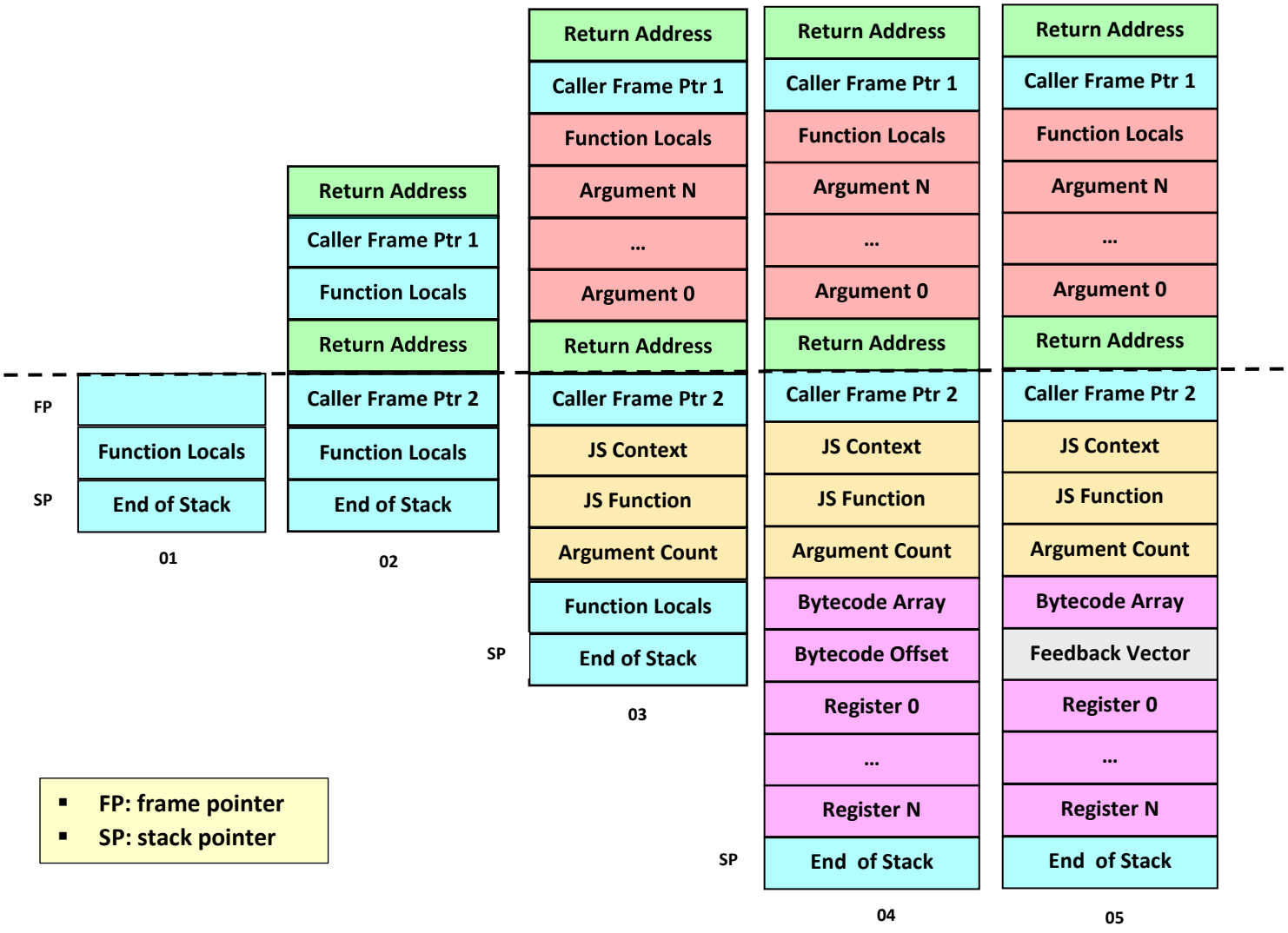
[Figure 24]: src/objects/elements-kind.h file

Curiously we have already seen part of such elements previously, which can be optimized or not by the V8 engine, and without forgetting that an object can have **elements** (usually -- not always -- referred by indexes as in an array -- and easily optimizable and with fast access) and **properties** (usually accessed by names like in a dictionary with key and associated value -- and not easily optimizable and slow access).

Regardless of used objects, and similarly to other programming languages, JavaScript also is concerned with memory management and provides us with a garbage collection mechanism, which acts on new and old space sections, and preserves sections such as code and large objects. The general criteria for whether to keep an object or not is based on the length of time that is in memory, where recent objects (young generation) are preferred to be kept at the expense of the older ones and thus such objects are grouped (classified) in new and old ones, distinguishing data from pointers using memory tagging as explained previously.

Actually, young generation memory is separated into two parts: **active/new semi-space** and **non-active semi-space**. Once the active semi-space becomes full, the **garbage collection mechanism** moves all active objects to the non-active space, and active objects that were already there (long-living objects) are moved to the old generation space. Once the size of the object in the old generation space becomes larger than a determined size, it is marked as disposable and will be removed later. It is important to highlight that any V8 optimization continues while the garbage collector is marking objects to later removing.

The next stage of the execution flow is the **Sparkplug** that acts as a non-optimizing fast JavaScript compiler, which had been introduced as a stage between Ignition and Turbofan, and nowadays works in a prior stage of **MagLev** (**Ignition → Sparkplug → MagLev → Turbofan**). **Sparkplug** is really fast because it compiles functions that have already been compiled (transformed) to bytecode, Thus, in other words, it starts its job from a bytecode instead of starting from a source code, and not generate any intermediate code. As **Sparkplug** handles with compiling details from an existing bytecode, reviewing the stack and V8 stack might be useful:



[Figure 25]: Regular stack frame (1,2) and V8 stack interpreter (3,4,5) | credits: based on Google specs.

There are five JavaScript stack frame layouts in **Figure 25**, which details about them follow below:

- In **layout 01** we have a basic stack frame with the **Frame Pointer (FP)** and **Stack Pointer (SP)**.
- In **layout 02**, there is an almost identical situation, but the caller function and callee function, in their respective frames, are shown in the same stack.
- In **layout 03**, we have a standard V8 JavaScript stack frame which is similar to layout 02, but adds JavaScript Function, JavaScript Context and Argument Count.
- In **layout 04**, we realize it is associated with the Ignition interpreter, which is register based, and there are virtual registers that store the current state of the interpreter and the respective bytecode array pointer.
- In **layout 05**, we have the Sparkplug stack that does not include the bytecode offset, but the feedback vector, which stores the object shape data, for the currently executing function. In this case, everything aims to achieve better performance with a minimal footprint (code) even because most of its code is composed of control flow and built-in calls.

In conclusion, Sparkplug works like a "bridge" from Ignition (emulator) to CPU bytecode (native execution). Details about **registers' operands, feedback vector, helpers calling built-in functions** and even preparation for protections such as **CFI (Control Flow Integrity)** can be checked in **src\baseline\baseline-compiler.h**:

```
172:   // Mark location as a jump target reachable via indirect branches, required
173:   // for CFI.
174:   enum class MarkAsIndirectJumpTarget { kNo, kYes };
175:
176:   Label* EnsureLabel(int offset, MarkAsIndirectJumpTarget mark =
177:                                  MarkAsIndirectJumpTarget::kNo) {
178:     Label* label = &labels_[offset];
179:     if (!label_tags_.Contains(offset * 2)) {
180:       label_tags_.Add(offset * 2);
181:       new (label) Label();
182:     }
183:     if (mark == MarkAsIndirectJumpTarget::kYes) {
184:       MarkIndirectJumpTarget(offset);
185:     }
186:     return label;
187:   }
188:   bool IsJumpTarget(int offset) const {
189:     return label_tags_.Contains(offset * 2);
190:   }
191:   bool IsIndirectJumpTarget(int offset) const {
192:     return label_tags_.Contains(offset * 2 + 1);
193:   }
194:   void MarkIndirectJumpTarget(int offset) { label_tags_.Add(offset * 2 + 1); }
195:
196:   Label* labels_;
197:   BitVector label_tags_;
198:
199: #ifdef DEBUG
200:   friend class SaveAccumulatorScope;
201:
202:   struct EffectState {
203:     bool may_have_deopted = false;
204:     bool accumulator_on_stack = false;
205:     bool safe_to_skip = false;
```

[Figure 26]: src\baseline\baseline-compiler.h

The **src\baseline\baseline-compiler.cc** and **src\baseline\x64\baseline-compiler-x64.inl.h** offers multiple details about helper functions, stack frame operation and interaction with Ignition:

```
BaselineCompiler::BaselineCompiler(
    LocalIsolate* local_isolate,
    Handle<SharedFunctionInfo> shared_function_info,
    Handle<BytecodeArray> bytecode)
    : local_isolate_(local_isolate),
      stats_(local_isolate->runtime_call_stats()),
      shared_function_info_(shared_function_info),
      bytecode_(bytecode),
      zone_(local_isolate->allocator(), ZONE_NAME),
      masm_(
          local_isolate->GetMainThreadIsolateUnsafe(), &zone_,
          BaselineAssemblerOptions(local_isolate->GetMainThreadIsolateUnsafe()),
          CodeObjectRequired::kNo, AllocateBuffer(bytecode)),
      basm_(&masm_),
      iterator_(bytecode_),
      labels_(zone_.AllocateArray<Label>(bytecode_->length())),
      label_tags_(2 * bytecode_->length(), &zone_) {
  // Empirically determined expected size of the offset table at the 95th %ile,
  // based on the size of the bytecode, to be:
  //
  //   16 + (bytecode size) / 4
  bytecode_offset_table_builder_.Reserve(
      base::bits::RoundUpToPowerOfTwo(16 + bytecode_->Size() / 4));
} « end BaselineCompiler »

void BaselineCompiler::GenerateCode() {
  {
    RCS_BASELINE_SCOPE(PreVisit);
    // Mark exception handlers as valid indirect jump targets. This is required
    // when CFI is enabled, to allow indirect jumps into baseline code.
    HandlerTable table(*bytecode_);
    for (int i = 0; i < table.NumberOfRangeEntries(); ++i) {
      MarkIndirectJumpTarget(table.GetRangeHandler(i));
    }
    for (; !iterator_.done(); iterator_.Advance()) {
      PreVisitSingleBytecode();
    }
    iterator_.Reset();
  }

  // No code generated yet.
```

[Figure 27]: src\baseline\baseline-compiler.cc

```
// A builtin call/jump mode that is used then short builtin calls feature is
// not enabled.
constexpr BuiltinCallJumpMode kFallbackBuiltinCallJumpModeForBaseline =
    BuiltinCallJumpMode::kIndirect;

void BaselineCompiler::Prologue() {
  ASM_CODE_COMMENT(&masm_);
  DCHECK_EQ(kJSFunctionRegister, kJavaScriptCallTargetRegister);
  int max_frame_size = bytecode_->max_frame_size();
  CallBuiltin<Builtin::kBaselineOutOfLinePrologue>(
      kContextRegister, kJSFunctionRegister, kJavaScriptCallArgCountRegister,
      max_frame_size, kJavaScriptCallNewTargetRegister, bytecode_);

  PrologueFillFrame();
}

void BaselineCompiler::PrologueFillFrame() {
  ASM_CODE_COMMENT(&masm_);
  // Inlined register frame fill
  interpreter::Register new_target_or_generator_register =
      bytecode_->incoming_new_target_or_generator_register();
  if (v8_flags.debug_code) {
    __ masm()->Cmp(kInterpreterAccumulatorRegister,
                   handle(ReadOnlyRoots(local_isolate_).undefined_value(),
                          local_isolate_));
    __ masm()->Assert(equal, AbortReason::kUnexpectedValue);
  }
  int register_count = bytecode_->register_count();
  // Magic value
```

[Figure 28]: src\baseline\x64\baseline-compiler-x64-inl.h

Our next step is to bring details on **Turbofan**. Actually, **Maglev** acts prior to Turbofan, but we will approach Maglev after Turbofan because it has been introduced in the last few years. Once again, we have:

- **Ignition → bytecode → Sparkplug** (non-optimizing compiler | transpiler)
- **Sparkplug → Maglev** (optimizer JIT compiler)
- **Maglev → Turbofan** (compiler)
- **Turbofan** → optimized code (it can be utilized or not, so returning to Ignition).

Disregarding temporarily the Maglev, the objective of **Turbofan** (the V8's JIT -- Just in Time compiler) is to transform bytecode (of course, bytecodes themselves are slow -- they are emulated) into optimized code using supporting techniques and mechanisms, when an **intermediate code (IR)** is produced, and it will help to accomplish this objective. For example, the **feedback vector** (from Ignition) is one of the mechanisms used by Ignition to store context and runtime information (by function) that will be used by Turbofan for optimizing the code, and such technique is also known as **Speculative Optimization**. Besides this approach, there are other phases such as **type analysis** (mitigates type confusion vulnerabilities), **range analysis** (mitigates integer and buffer overflow vulnerabilities), **type feedback propagation**, different reductions such as **branch and constant folding, escape analysis** (replacement of aggregations), **dead code** (also node) and **redundancy elimination, peeled loops, redundant store elimination** and **control flow optimization**. In a general way, Turbofan acts when a piece of code is hot (executed multiple times) then it can be optimized for saving time for next executions. However, as expected, the initial observation and searching for hot codes, through inline caches that hold information about functions, can cause a delay before Turbofan really starts its operation. The optimization cycle has multiple phases and is tracked data and code flow graphs. A quick visualization might come from the following script:

```
function Hack(researcher, topic, conference) {
  this.researcher = researcher;
  this.topic = topic;
  this.conference = conference;
  var firstobj = { profile_prop1: topic, profile_prop2: conference };
  var lastobj = { };
  lastobj.topic = this.topic;
  lastobj.conf = this.conference;
  lastobj.order = 1;
}

function HotHack(a, b, c, d){
  const final = {a, b, c, d};
  // console.log(final);
}

obj1 = new Hack("Hacker01", "Windows", "Conf_Germany");
obj2 = new Hack("Hacker02", "macOS", "Conf_USA");
obj3 = new Hack("Hacker03", "Hypervisor", "Conf_Canada");
obj4 = new Hack("Hacker04", "Browsers", "Conf_Japan");

 for (let k=0; k < 500000; k++){
    HotHack(obj1, obj2, obj3, obj4);
}
```

[Figure 29]: script_test_03.js

The **script_test_03.js** is almost identical to its predecessor, but there are some changes:

- The **HotHack function** has been created and contains the assignment to final variable.
- There is a **stressing loop** calling HotHack function 500000 times.

As I have educationally skipped **Maglev**, I need to disable it during its execution to pretend that it does not exist. Therefore, two additional options will be used:

- **--no-maglev:** disable Maglev optimizer.
- **--trace-turbo:** provide means to trace Turbofan execution.

You should know that other possibilities exist, but it is enough for now. Executing the given script:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe --allow-natives-syntax --no-maglev --trace-turbo
Concurrent recompilation has been disabled for tracing.
V8 version 13.2.0 (candidate)
d8> load("script_test_03.js")
-----------------------------------------------------
Begin compiling method HotHack using TurboFan
-----------------------------------------------------
Finished compiling method HotHack using TurboFan
-----------------------------------------------------
Begin compiling method  using TurboFan
-----------------------------------------------------
Finished compiling method  using TurboFan
undefined
d8> %DebugPrint(HotHack)
DebugPrint: 000002D100098EC5: [Function] in OldSpace
 - map: 0x02d10008201d <Map[32](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x02d100081f45 <JSFunction (sfi = 000002D1000474DD)>
 - elements: 0x02d100000775 <FixedArray[0]> [HOLEY_ELEMENTS]
 - function prototype:
 - initial_map:
 - shared_info: 0x02d100098de1 <SharedFunctionInfo HotHack>
 - name: 0x02d100098bcd <String[7]: #HotHack>
 - formal_parameter_count: 5
 - kind: NormalFunction
 - context: 0x02d10008188d <NativeContext[301]>
 - code: 0x02b300040619 <Code TURBOFAN_JS>
 - dispatch_handle: 0x264200
 - canonical feedback cell dispatch_handle: 0x264000
 - source code: (a, b, c, d){
  const final = {a, b, c, d};
  // console.log(final);
}

 - properties: 0x02d100000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    000002D100000DC1: [String] in ReadOnlySpace: #length: 0x02d10024ff5d <AccessorInfo name= 0x02d
100000dc1 <String[6]: #length>, data= 0x02d100000069 <undefined>> (const accessor descriptor, attr
s: [__C]), location: descriptor
    000002D100000DED: [String] in ReadOnlySpace: #name: 0x02d10024ff45 <AccessorInfo name= 0x02d10
0000ded <String[4]: #name>, data= 0x02d100000069 <undefined>> (const accessor descriptor, attrs: [
__C]), location: descriptor
    000002D100004309: [String] in ReadOnlySpace: #arguments: 0x02d10024ff15 <AccessorInfo name= 0x
02d100004309 <String[9]: #arguments>, data= 0x02d100000069 <undefined>> (const accessor descriptor
, attrs: [___]), location: descriptor
    000002D100004589: [String] in ReadOnlySpace: #caller: 0x02d10024ff2d <AccessorInfo name= 0x02d
100004589 <String[6]: #caller>, data= 0x02d100000069 <undefined>> (const accessor descriptor, attr
s: [___]), location: descriptor
    000002D100000DD5: [String] in ReadOnlySpace: #prototype: 0x02d10024ff75 <AccessorInfo name= 0x
02d100000dd5 <String[9]: #prototype>, data= 0x02d100000069 <undefined>> (const accessor descriptor
, attrs: [W__]), location: descriptor
 }
```

```
 - feedback vector: 000002D1000993A5: [FeedbackVector] in OldSpace
 - map: 0x02d100000831 <Map(FEEDBACK_VECTOR_TYPE)>
 - length: 9
 - shared function info: 0x02d100098de1 <SharedFunctionInfo HotHack>
 - tiering state: TieringState::kNone
 - invocation count: 3002
 - closure feedback cell array: 000002D10000219D: [ClosureFeedbackCellArray] in ReadOnlySpace
 - map: 0x02d100000809 <Map(CLOSURE_FEEDBACK_CELL_ARRAY_TYPE)>
 - length: 0
 - elements:

 - slot #0 Literal  {
     [0]: 0x02d100099489 <AllocationSite>
  }
 - slot #1 DefineNamedOwn MONOMORPHIC
   0x02d10009937d <Map[28](HOLEY_ELEMENTS)>: StoreHandler(Smi)(kind = kField, descriptor = 0, is i
n object = 1, representation = h, field index = 3)
 {
     [1]: [weak] 0x02d10009937d <Map[28](HOLEY_ELEMENTS)>
     [2]: 3604480
  }
 - slot #3 DefineNamedOwn MONOMORPHIC
   0x02d10009937d <Map[28](HOLEY_ELEMENTS)>: StoreHandler(Smi)(kind = kField, descriptor = 1, is i
n object = 1, representation = h, field index = 4)
 {
     [3]: [weak] 0x02d10009937d <Map[28](HOLEY_ELEMENTS)>
     [4]: 4653120
  }
 - slot #5 DefineNamedOwn MONOMORPHIC
   0x02d10009937d <Map[28](HOLEY_ELEMENTS)>: StoreHandler(Smi)(kind = kField, descriptor = 2, is i
n object = 1, representation = h, field index = 5)
 {
     [5]: [weak] 0x02d10009937d <Map[28](HOLEY_ELEMENTS)>
     [6]: 5701760
  }

 - slot #7 DefineNamedOwn MONOMORPHIC
   0x02d10009937d <Map[28](HOLEY_ELEMENTS)>: StoreHandler(Smi)(kind = kField, descriptor = 3, is i
n object = 1, representation = h, field index = 6)
 {
     [7]: [weak] 0x02d10009937d <Map[28](HOLEY_ELEMENTS)>
     [8]: 6750400
  }
000002D10008201D: [Map] in OldSpace
 - map: 0x02d10008183d <MetaMap (0x02d10008188d <NativeContext[301]>)>
 - type: JS_FUNCTION_TYPE
 - instance size: 32
 - inobject properties: 0
 - unused property fields: 0
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - callable
 - constructor
 - has_prototype_slot
 - back pointer: 0x02d100000069 <undefined>
 - prototype_validity cell: 0x02d100000ab1 <Cell value= 1>
 - instance descriptors (own) #5: 0x02d100082045 <DescriptorArray[5]>
 - prototype: 0x02d100081f45 <JSFunction (sfi = 000002D1000474DD)>
 - constructor: 0x02d100081fe9 <JSFunction Function (sfi = 000002D10025536D)>
 - dependent code: 0x02d100000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0

function HotHack(a, b, c, d){
  const final = {a, b, c, d};
  // console.log(final);
}
d8>
```

[Figure 30]: executing and debugging script_test_03.js

The following comments can bring context to the output above:

- We have noticed that the **HotHack function** is being compiled by **Turbofan**.
- **prototype: 0x02d100081f45 <JSFunction (sfi = 000002D1000474DD)>:**
  - there is a **prototype object** associated with the function in 0x02d100081f45.
  - the **sfi (small function ID)** is an identifier for the function.
- **SharedFunctionInfo:** it stores information that is shared with other instances function. We know that **HotHack function** is called thousands of times and there are the same number of instances, but the function itself is given by one object.
- The **HotHack function** has 5 parameters (**this, a, b, c** and **d**).
- The code is given as **TURBOFAN_JS** (**code: 0x02b300040619 <Code TURBOFAN_JS>**) and there are other possible types such as **MAGLEV, BASELINE,** and **INTERPRETED_FUNCTION** (**src\objects\code-kind.cc**):

```
#include "src/objects/code-kind.h"

namespace v8 {
namespace internal {

const char* CodeKindToString(CodeKind kind) {
  switch (kind) {
#define CASE(name)     \
  case CodeKind::name: \
    return #name;
    CODE_KIND_LIST(CASE)
#undef CASE
  }
  UNREACHABLE();
}

const char* CodeKindToMarker(CodeKind kind) {
  switch (kind) {
    case CodeKind::INTERPRETED_FUNCTION:
      return "~";
    case CodeKind::BASELINE:
      return "^";
    case CodeKind::MAGLEV:
      return "+";
    case CodeKind::TURBOFAN_JS:
      return "*";
    default:
      return "";
  }
}
```

[Figure 31]: src\objects\code-kind.cc

- Readers can see a term named **feedback cell**, and it makes part of the **feedback vector mechanisms** mentioned previously, which is used to store runtime information (value type and function call) that will be used by the Turbofan for optimization.
- **ReadOnlySpace** is a part of the memory heap where values that do not change during the execution of the script are stored, which helps during the optimization.
- **Feedback vector** is the mechanism used to gather script's runtime information that will be used for optimization and, according to the highlighted line (**feedback vector: 000002D1000993A5: [FeedbackVector] in OldSpace**), such information is saved into **OldSpace heap memory**, which is reserved for long-lived objects (objects that have been kept in the memory after multiple garbage collection operations).

- As the code is being optimized, there is a **Map** (metadata structure that describes the shape and objects, in a comparable way to C++ programming) whose type is **FEEDBACK_VECTOR_TYPE**.
- As I already have explained previously, **ClosureFeedbackCellArray** stores feedback information on **closures**, which are JavaScript functions that have access to the parent scope (regardless the parent function has been closed or not). As consequence, we see a reference to a **Map type CLOSURE_FEEDBACK_CELL_ARRAY_TYPE.**
- The reference to **MONOMORPHIC** for several elements (different slots) means that every time the HotHack function has been executed, the same type has been found, which obviously helps a lot during the optimization phase. Certainly, it would not be the case if the type were **POLYMORPHIC, MEGAMORPHIC** or other type, as defined in **src\ic\ic.cc f**ile:

```
50: namespace v8 {
51: namespace internal {
52:
53: // Aliases to avoid having to repeat the class.
54: // With C++20 we can use "using" to introduce scoped enums.
55: constexpr InlineCacheState NO_FEEDBACK = InlineCacheState::NO_FEEDBACK;
56: constexpr InlineCacheState UNINITIALIZED = InlineCacheState::UNINITIALIZED;
57: constexpr InlineCacheState MONOMORPHIC = InlineCacheState::MONOMORPHIC;
58: constexpr InlineCacheState RECOMPUTE_HANDLER =
59:     InlineCacheState::RECOMPUTE_HANDLER;
60: constexpr InlineCacheState POLYMORPHIC = InlineCacheState::POLYMORPHIC;
61: constexpr InlineCacheState MEGAMORPHIC = InlineCacheState::MEGAMORPHIC;
62: constexpr InlineCacheState MEGADOM = InlineCacheState::MEGADOM;
63: constexpr InlineCacheState GENERIC = InlineCacheState::GENERIC;
64:
65: char IC::TransitionMarkFromState(IC::State state) {
66:   switch (state) {
67:     case NO_FEEDBACK:
68:       return 'X';
69:     case UNINITIALIZED:
70:       return '0';
71:     case MONOMORPHIC:
72:       return '1';
73:     case RECOMPUTE_HANDLER:
74:       return '^';
75:     case POLYMORPHIC:
76:       return 'P';
77:     case MEGAMORPHIC:
78:       return 'N';
79:     case MEGADOM:
80:       return 'D';
81:     case GENERIC:
82:       return 'G';
83:   }
84:   UNREACHABLE();
85: } « end TransitionMarkFromState »
```

[Figure 32]: src\ic\ic.cc

- The **NativeContext type** indicates a data structure, which can include functions and some built-in objects, beside other information context on a JavaScript code.
- The last two highlighted lines in the output show information about a **prototype object** and a **constructor function**, respectively.

The output itself is not complicated to understand, but I could mention further concepts related to the subject and, eventually, make a few correlations.

Additionally, when I started the **d8.exe** debugging session, I used **--trace-turbo** option, which produced two JSON files containing the Turbofan execution tracing, in the same directory:

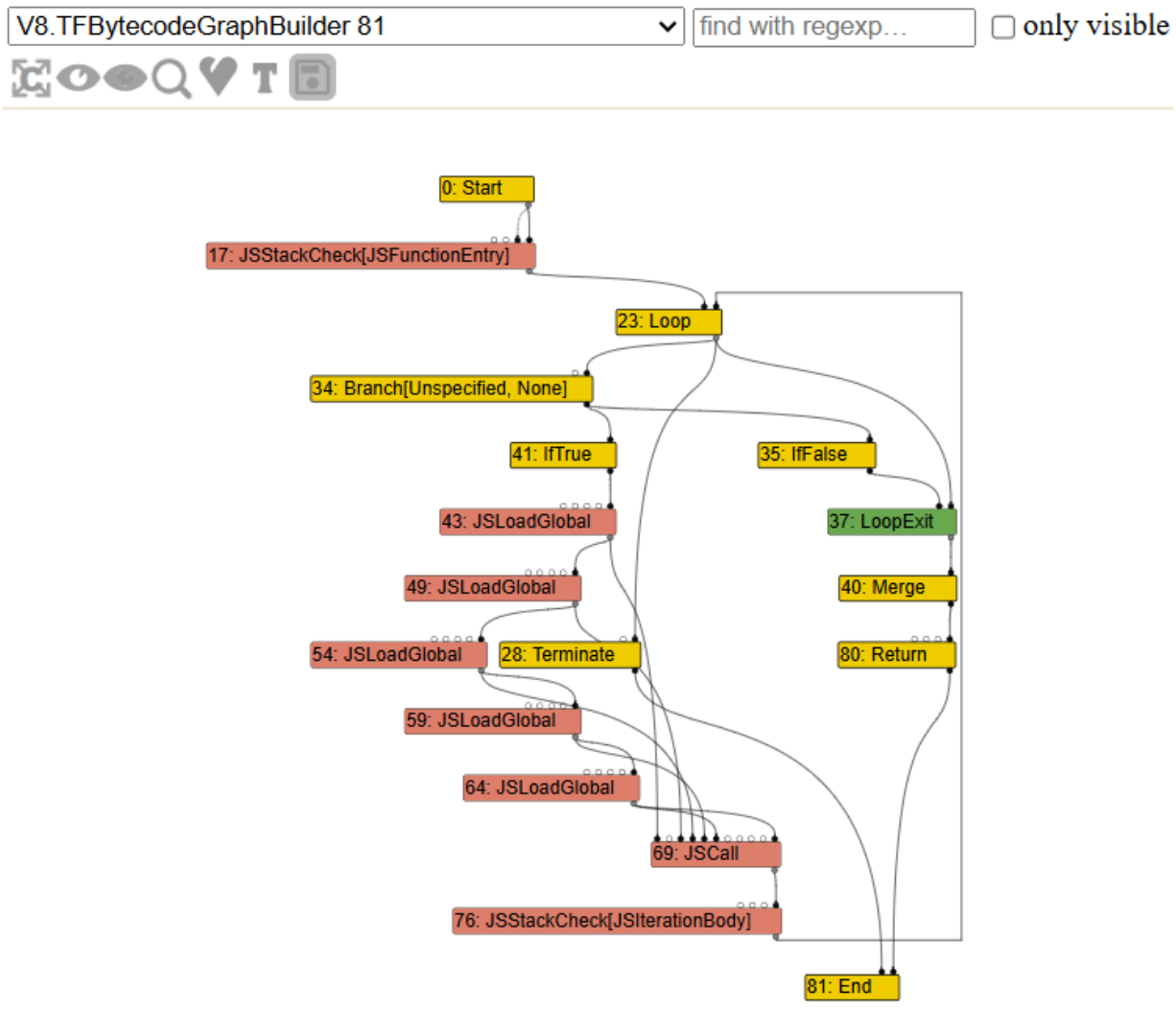
```
C:\articles\chromium\v8>dir *.json
 Volume in drive C has no label.
 Volume Serial Number is 6008-4344

 Directory of C:\articles\chromium\v8

11/01/2024  08:03 PM                68 pyrightconfig.json
12/21/2024  01:20 AM         1,481,743 turbo-000002D100098D48-1.json
12/21/2024  01:20 AM           754,591 turbo-HotHack-0.json
```

[Figure 33]: produced JSON files (last two files)

To visualize the content of this tracing we can use the **turbolizer tool**, which is stored in **v8\tools\turbolizer** folder. To use it you have to execute the following steps:

- **cd v8\tools\turbolizer**
- **npm -i**
- **npm run-script build**
- **python3 -m http.server 8000**

Open the browser up, press **CTRL+L**, point to the **first JSON file** and you will see the following image:



[Figure 34]: Turbolizer | Bytecode Graph Builder View

This tool offers a series of advantages. The graph provides necessary information to understand the optimization process, we can interact with nodes and change the view according to our goals. Furthermore, the entire disassembly is also shown:

```
kind = TURBOFAN_JS
stack_slots = 17
compiler = turbofan
address = 000002B300040941
Instructions (size = 556)
00007FF765C801C0     0  REX.W leaq rbx,[rip+0xfffffff9]
00007FF765C801C7     7  REX.W cmpq rbx,rcx
00007FF765C801CA     a  jz 00007FF765C801D9  <+0x19>
00007FF765C801CC     c  movl rdx,0000000000000084
00007FF765C801D1    11  call [r13+0x5700]
00007FF765C801D8    18  int3l
00007FF765C801D9    19  REX.W leaq r10,[r13+0x817df80]
00007FF765C801E0    20  REX.W movq rbx,r15
00007FF765C801E3    23  shrl rbx, 9
00007FF765C801E6    26  shll rbx, 4
00007FF765C801E9    29  movzxwl rbx,[r10+rbx*1+0x8]
00007FF765C801EF    2f  cmpl rbx,0x1
00007FF765C801F2    32  jz 00007FF765C80201  <+0x41>
00007FF765C801F4    34  movl rdx,0000000000000088
00007FF765C801F9    39  call [r13+0x5700]
00007FF765C80200    40  int3l
00007FF765C80201    41  movl rbx,[rcx-0xc]
00007FF765C80204    44  REX.W orq rbx,[r13+0x1e0]
00007FF765C8020B    4b  testb [rbx+0x1a],0x20
```

[Figure 35]: Turbolizer | Disassembly

Another interesting point is that the **inlined functions** are also presented:

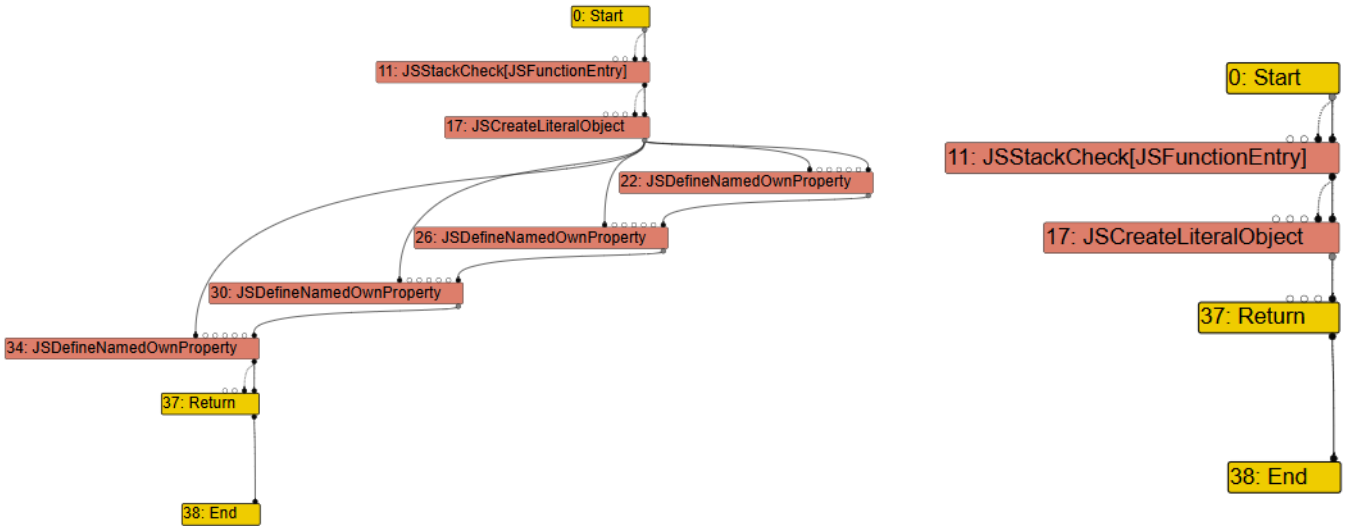
script_test_03.js:HotHack — inlined function

```
1 (a, b, c, d){
2   const final = {a, b, c, d};
3   // console.log(final);
4 }
```

[Figure 36]: Turbolizer | Inlined Function

Inlining function (this activity replaces a function call by the code itself) is one of the main steps adopted by Turbofan to optimize the target code, and Turbofan does a good job in selectively choosing the appropriate function to be inlined, which in this case is an obvious choice due to the fact that the function is executed so many times. Consequently, the overhead is also reduced.

If readers check the second Json file (named **turbo-Hack-0.json**), which shows a representation focused only on the **inlined function**, and it is feasible to make comparisons using multiple optimization views, at different moments and phases, which makes evident differences:



[Figure 37]: Turbolizer | Optimized Function -- Bytecode GraphBuilder and Inlining View

All graphs presented so far have colored nodes:

- **Yellow:** control nodes.
- **Light Blue:** input and return values.
- **Blue:** simplified nodes, which are operations related to IR (intermediate representation).
- **Red:** JavaScript node.
- **Green:** machine node (low-level instructions).

As suggestions, check the following optimizing views:

- **Typer:** it is responsible for determining the type of each node, which helps the optimizer to choice the best technique.
- **Typed Lowering:** this optimizing technique translates high-level operation into basic and low-level operations, which are simpler to optimize.
- **Loop Peeling:** it eliminates the first operations, so decreasing the number of loops to be accomplished.
- **Load Elimination:** it is responsible for eliminating unnecessary memory load operations.
- **Simplified Lowering:** this technique simplifies complex nodes, which helps optimizing phase.
- **Untyper:** it works as a double-checking type, when the optimized undo the typing phase to confirm that the original type has been propagated correctly to the optimizing stage.
- **Generic Optimization:** it is responsible for regular optimization techniques such as constant unfolding, dead code elimination and redundancy elimination.
- **Schedule:** this technique is responsible for reordering the code to a more efficient, lowers latency and faster execution. This technique involves Control Flow Graph (CFG), instruction selection and even block placement.
- **Escape Analysis:** it is an optimization technique used to determine whether an object should be allocated in the stack or heap, and such decision impact the garbage collection phase and increase the performance.

A recommended way to learn further details about these optimizations is checking the **v8\src\compiler folder** and checking for a series of files (with .cc and .h extensions) associated with optimization techniques, which a few of them are:

- branch-elimination.cc
- bytecode-analysis.cc
- constant-folding-reducer.cc
- dead-code-elimination.cc
- escape-analysis.cc
- escape-analysis-reducer.cc
- int64-lowering.cc
- js-call-reducer.cc
- js-type-lowering.cc
- linear-scheduler.cc
- load-elimination.cc
- loop-analysis.cc
- loop-peeling.cc
- loop-unrolling.cc
- memory-lowering.cc
- memory-optimizer.cc
- schedule.cc
- simplified-lowering.cc
- turbofann-typer.cc

There are many other files that should be examined. Moreover, there are a series of options that can be passed to **d8.exe debugger** for tracing specific optimizations, as shown below:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe --help | findstr /i trace-turbo
  --trace-turbo (trace generated TurboFan IR)
        type: bool  default: --no-trace-turbo
  --trace-turbo-path (directory to dump generated TurboFan IR to)
        type: string  default: --trace-turbo-path=""
  --trace-turbo-filter (filter for tracing turbofan compilation)
        type: string  default: --trace-turbo-filter="*"
  --trace-turbo-graph (trace generated TurboFan graphs)
        type: bool  default: --no-trace-turbo-graph
  --trace-turbo-scheduled (trace TurboFan IR with schedule)
        type: bool  default: --no-trace-turbo-scheduled
  --trace-turbo-file-prefix (trace turbo graph to a file with given prefix)
        type: string  default: --trace-turbo-file-prefix="turbo"
  --trace-turbo-cfg-file (trace turbo cfg graph (for C1 visualizer) to a given file name)
        type: string  default: --trace-turbo-cfg-file=""
  --trace-turbo-types (trace TurboFan's types)
        type: bool  default: --trace-turbo-types
  --trace-turbo-scheduler (trace TurboFan's scheduler)
        type: bool  default: --no-trace-turbo-scheduler
  --trace-turbo-reduction (trace TurboFan's various reducers)
        type: bool  default: --no-trace-turbo-reduction
  --trace-turbo-trimming (trace TurboFan's graph trimmer)
        type: bool  default: --no-trace-turbo-trimming
  --trace-turbo-jt (trace TurboFan's jump threading)
        type: bool  default: --no-trace-turbo-jt
```

[Figure 38]: Turbolizer | d8 available options associated with turbo-tracing (truncated)

Next step in our article is to present details of **Maglev compiler**.

Maglev is the optimizing compiler that has been introduced between Sparkplug and Turbofan compilers and, it uses **SSA (Static Single Assignment)** approach and **CFG (control flow graph)**. In general, and rough words, we can say that Maglev aims to provide Turbofan with better prepared code to be optimized.

Maglev does its work through the following considerations:

- Building an abstract interpretation of the frame state (runtime information of the stack during JavaScript code execution).
- Creating **SSA (Static Single Assignment) nodes** that represent results of the present expression evaluation (each SSA node describes an operation).
- SSA is a kind of property of an intermediate representation (IR), where variables are defined and initialized before their usage, which is used by the Turbofan compiler. Therefore, each note offers a relationship between a given definition and its direct use, and this representation makes simpler the analysis and optimization later.
- Eventually, there are multiple control paths with multiple **SSA nodes**, which are merged into special nodes known as **Phi Nodes**, whose nodes can have multiple values and one of them is chosen according to the runtime execution path. Thus, first **Maglev generates SSA nodes** then it produces **Phi nodes**, if it is necessary.
- Building a graph that retracts and contains all necessary information for later optimization.
- Deoptimizing nodes when it is necessary.
- Maglev generates assembly code through its nodes.
- At the end, **Maglev** does a more complex task than **Sparkplug**, but it is simpler and easier than **Turbofan**.

In the source code, Maglev files are located in **v8\src\maglev** folder:

```
 Directory of C:\articles\chromium\v8\src\maglev

maglev-assembler-inl.h                      maglev-graph-processor.h
maglev-assembler.cc                         maglev-graph-verifier.h
maglev-assembler.h                          maglev-graph.h
maglev-basic-block.h                        maglev-interpreter-frame-state.cc
maglev-code-gen-state.h                     maglev-interpreter-frame-state.h
maglev-code-generator.cc                    maglev-ir-inl.h
maglev-code-generator.h                     maglev-ir.cc
maglev-compilation-info.cc                  maglev-ir.h
maglev-compilation-info.h                   maglev-phi-representation-selector.cc
maglev-compilation-unit.cc                  maglev-phi-representation-selector.h
maglev-compilation-unit.h                   maglev-pipeline-statistics.cc
maglev-compiler.cc                          maglev-pipeline-statistics.h
maglev-compiler.h                           maglev-post-hoc-optimizations-processors.h
maglev-concurrent-dispatcher.cc             maglev-pre-regalloc-codegen-processors.h
maglev-concurrent-dispatcher.h              maglev-regalloc-data.h
maglev-graph-builder.cc                     maglev-regalloc.cc
maglev-graph-builder.h                      maglev-regalloc.h
maglev-graph-labeller.h                     maglev-register-frame-array.h
maglev-graph-printer.cc
maglev-graph-printer.h
```

[Figure 39]: Maglev source files

It is unnecessary to describe each of these files here, and some of them are related to Maglev graph and graph builder, frame state, Intermediate Representation, Phi representation, Maglev pipeline, registers and, as expected, compiler and associated tasks. If readers have spare time, try to inspect the **src\maglev\maglev-compiler.cc**, which a part of its code is shown below:

```
58: bool MaglevCompiler::Compile(LocalIsolate* local_isolate,
59:                              MaglevCompilationInfo* compilation_info) {
60:   compiler::CurrentHeapBrokerScope current_broker(compilation_info->broker());
61:   Graph* graph =
62:       Graph::New(compilation_info->zone(),
63:                  compilation_info->toplevel_compilation_unit()->is_osr());
64:
65:   // Build graph.
66:   if (v8_flags.print_maglev_code || v8_flags.code_comments ||
67:       v8_flags.print_maglev_graph || v8_flags.print_maglev_graphs ||
68:       v8_flags.trace_maglev_graph_building ||
69:       v8_flags.trace_maglev_escape_analysis ||
70:       v8_flags.trace_maglev_phi_untagging || v8_flags.trace_maglev_regalloc ||
71:       v8_flags.trace_maglev_object_tracking) {
72:     compilation_info->set_graph_labeller(new MaglevGraphLabeller());
73:   }
74:
75:   {
76:     UnparkedScopeIfOnBackground unparked_scope(local_isolate->heap());
77:
78:     if (v8_flags.print_maglev_code || v8_flags.print_maglev_graph ||
79:         v8_flags.print_maglev_graphs || v8_flags.trace_maglev_graph_building ||
80:         v8_flags.trace_maglev_phi_untagging || v8_flags.trace_maglev_regalloc) {
81:       MaglevCompilationUnit* top_level_unit =
82:           compilation_info->toplevel_compilation_unit();
83:       std::cout << "Compiling " << Brief(*compilation_info->toplevel_function())
84:                 << " with Maglev\n";
85:       BytecodeArray::Disassemble(top_level_unit->bytecode().object(),
86:                                  std::cout);
87:       if (v8_flags.maglev_print_feedback) {
88:         Print(*top_level_unit->feedback().object(), std::cout);
89:       }
90:     }
91:
92:     MaglevGraphBuilder graph_builder(
93:         local_isolate, compilation_info->toplevel_compilation_unit(), graph);
94:
95:     {
96:       TRACE_EVENT0(TRACE_DISABLED_BY_DEFAULT("v8.compile"),
97:                    "V8.Maglev.GraphBuilding");
98:       graph_builder.Build();
99:
100:      if (v8_flags.print_maglev_graphs) {
101:        std::cout << "\nAfter graph building" << std::endl;
102:        PrintGraph(std::cout, compilation_info, graph);
103:      }
104:    }
105:
106:    if (v8_flags.maglev_licm) {
107:      TRACE_EVENT0(TRACE_DISABLED_BY_DEFAULT("v8.compile"),
108:                   "V8.Maglev.LoopOptimizations");
109:
```

[Figure 40]: src\maglev\maglev-compiler.cc

Maglev makes part of Chrome since Chrome 117 beta (AUG/16/2023) and, of course, we should consider it in any analysis. Therefore, it is time to repeat our short experiment with **script_test_03.js**, but this time executing the d8.exe debugger without any option to disable Maglev (**out\x64.debug\d8.exe --allow-natives-syntax**), as shown below:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe --allow-natives-syntax
V8 version 13.2.0 (candidate)
d8> load("script_test_03.js")
undefined
d8> %DebugPrint(HotHack)
DebugPrint: 000002D300098E39: [Function] in OldSpace
 - map: 0x02d30008201d <Map[32](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x02d300081f45 <JSFunction (sfi = 000002D3000474DD)>
 - elements: 0x02d300000775 <FixedArray[0]> [HOLEY_ELEMENTS]
 - function prototype:
 - initial_map:
 - shared_info: 0x02d300098d55 <SharedFunctionInfo HotHack>
 - name: 0x02d300098b41 <String[7]: #HotHack>
 - formal_parameter_count: 5
 - kind: NormalFunction
 - context: 0x02d30008188d <NativeContext[301]>
 - code: 0x03f70008016d <Code MAGLEV>
 - dispatch_handle: 0x264200
 - canonical feedback cell dispatch_handle: 0x264000
 - source code: (a, b, c, d){
  const final = {a, b, c, d};
  // console.log(final);
}
```

[Figure 41]: executing and debugging script_test_03.js (truncated) |Maglev enabled

The shortened output above clearly shows that Maglev is responsible for the optimization this time instead of Turbofan. Maglev graph can be printed by using **--print-maglev-graph** option while calling **d8.exe** debugger:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe --allow-natives-syntax --print-maglev-graph
Concurrent maglev has been disabled for tracing.
V8 version 13.2.0 (candidate)
d8> load("script_test_03.js")
Compiling 0x035800098e39 <JSFunction HotHack (sfi = 0000035800098D55)> with Maglev
Parameter count 5
Register count 2
Frame size 16
         000000B80004026C @    0 : 85 00 00 29       CreateObjectLiteral [0], [0], #41
         000000B800040270 @    4 : cd                Star1
         000000B800040271 @    5 : 0b 03             Ldar a0
         000000B800040273 @    7 : 3a f8 01 01       DefineNamedOwnProperty r1, [1], [1]
         000000B800040277 @   11 : 0b 04             Ldar a1
         000000B800040279 @   13 : 3a f8 02 03       DefineNamedOwnProperty r1, [2], [3]
         000000B80004027D @   17 : 0b 05             Ldar a2
         000000B80004027F @   19 : 3a f8 03 05       DefineNamedOwnProperty r1, [3], [5]
         000000B800040283 @   23 : 0b 06             Ldar a3
         000000B800040285 @   25 : 3a f8 04 07       DefineNamedOwnProperty r1, [4], [7]
         000000B800040289 @   29 : 1b f8 f9          Mov r1, r0
         000000B80004028C @   32 : 0e                LdaUndefined
         000000B80004028D @   33 : b3                Return
Constant pool (size = 5)
```

[Figure 42]: Maglev: bytecode and SSA nodes (first part)

The first part above is only part of the low-level instructions (V8 bytecodes) involved in the script execution. The second part shows a part of the graph, and it has colors and arrows due to SSA nodes, but as the used terminal does not support such visual indications, so readers will see “strange strings”:

```
After register allocation
Graph

    1/7: Constant(0x03580008188d <NativeContext[301]>) ΓåÆ v-1, live range: [1-28]
    2/6: Constant(0x035800098e39 <JSFunction HotHack (sfi = 0000035800098D55)>) ΓåÆ v-1, live range
: [2-28]
   3/13: RootConstant(uninitialized_value) ΓåÆ v-1, live range: [3-21]
    4/8: RootConstant(undefined_value) ΓåÆ v-1, live range: [4-31]
   5/11: RootConstant(empty_fixed_array) ΓåÆ v-1, live range: [5-17]
 Block b1
0x035800098d55 <SharedFunctionInfo HotHack> (0x035800288a4d <String[17]: "script_test_03.js">)
   0 : CreateObjectLiteral [0], [0], #41
    6/1: InitialValue(<this>) ΓåÆ [stack:-6|t], live range: [6-28]
    7/2: InitialValue(a0) ΓåÆ [stack:-7|t], live range: [7-28]
    8/3: InitialValue(a1) ΓåÆ [stack:-8|t], live range: [8-28]
    9/4: InitialValue(a2) ΓåÆ [stack:-9|t], live range: [9-28]
   10/5: InitialValue(a3) ΓåÆ [stack:-10|t], live range: [10-29]
   11/9: FunctionEntryStackCheck
         Γå│ lazy @-1 (6 live vars)
  12/10: Jump b2
      Γåô
 Block b2
  13/14: AllocationBlock(Young) ΓåÆ [rdi|R|t], live range: [13-14]
  14/15: InlinedAllocation(0x0358000992f1 <Map[28](HOLEY_ELEMENTS)>) [v13/n14:[rdi|R|t]] ΓåÆ [rdi|R
|t], live range: [14-29]
  15/16: StoreMap(0x0358000992f1 <Map[28](HOLEY_ELEMENTS)>, InitializingYoung) [v14/n15:[rdi|R|t]]
     33: ConstantGapMove(v5/n11 ΓåÆ [rax|R|t])
  16/17: StoreTaggedFieldNoWriteBarrier(0x4) [v14/n15:[rdi|R|t], v5/n11:[rax|R|t]]
  17/18: StoreTaggedFieldNoWriteBarrier(0x8) [v14/n15:[rdi|R|t], v5/n11:[rax|R|t]]
     34: ConstantGapMove(v3/n13 ΓåÆ [rax|R|t])
  18/19: StoreTaggedFieldNoWriteBarrier(0xc) [v14/n15:[rdi|R|t], v3/n13:[rax|R|t]]
  19/20: StoreTaggedFieldNoWriteBarrier(0x10) [v14/n15:[rdi|R|t], v3/n13:[rax|R|t]]
  20/21: StoreTaggedFieldNoWriteBarrier(0x14) [v14/n15:[rdi|R|t], v3/n13:[rax|R|t]]
  21/22: StoreTaggedFieldNoWriteBarrier(0x18) [v14/n15:[rdi|R|t], v3/n13:[rax|R|t]]
```

[Figure 43]: Maglev: bytecode and SSA nodes (second part)

Initially, readers might think that such details are not useful, but as Maglev uses **speculative optimization** to gain performance, multiple issues might arise from this process due to non-initialize variable or even incorrectly typed variable that may not be longer valid (actually, the entire input might not be longer valid), which potentially can cause security issues such as logical vulnerabilities, type confusion, out-of-boundary and memory corruption. Even finding vulnerabilities related to these well-known bug classes, usually it is necessary to find a second one to circumvent the Chrome sandbox to really compromise the system.

It is interesting to notice that:

- **Maglev** transforms bytecodes into **SSA (Static Single Assignment) nodes**.
- **Ignition** transforms bytecodes into **Sea of Nodes**, which are those graphs that I have shown previously (without mentioning this name).

If readers think about it for a minute you will notice that there are multiple optimizations only in these two components:

- **Maglev** does two optimizations that are the transformation of bytecode into **SSA nodes** and the usage of **Phi nodes**.
- **Turbofan** does multiple and different optimizations such as **typed lowering, loop peeling, simplified lowering, schedule, escape analysis, load elimination** and many other ones.

All performance mechanisms involved are excellent, but unavoidably problems with object building will come up too and, as readers might imagine, a potential increasing in security issues is expected too. Readers can check on the **Chromium tracker** the last issues related to Maglev component (https://issues.chromium.org/issues?q=status:open%20componentid:1456177&s=created_time:desc&pli=1):

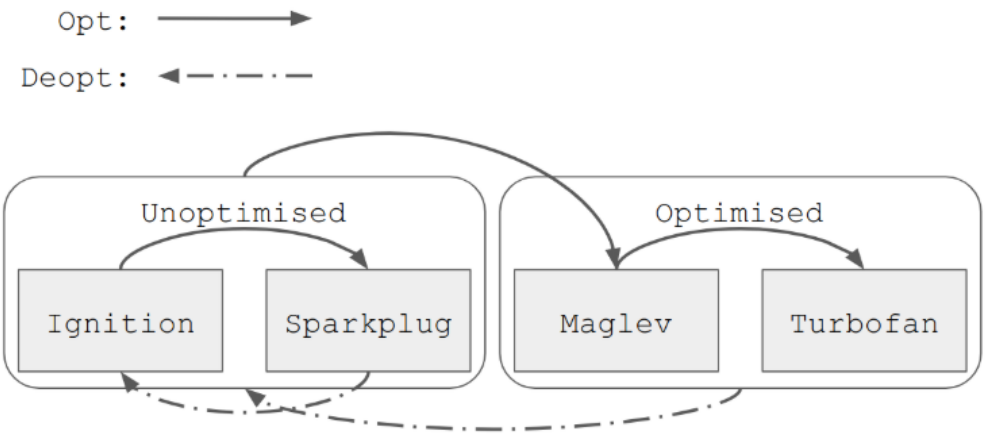
status:open componentid:1456177

1 - 26 of 26

P	TYPE	TITLE	ASSIGNEE	STATUS	7D VIEWS	ID	LAST MODIFIED
P1	Bug	mjsunit/string-wrapper-add-1 sta...		Assigned	0	376696919	Nov 4, 2024 04:28AM
P2	Bug	[riscv] AssertZeroExtended failed...	--	New	0	370177584	Sep 29, 2024 02:30AM
P2	Bug	[riscv64] many testcases failed a...	--	New	0	367771118	Sep 18, 2024 05:05AM
P2	Task	Tracking bug: Loop SPeeling		Assigned	0	350764035	Oct 9, 2024 10:09AM
P2	Bug	[riscv] Enabling Maglev on RISC-...		Assigned	0	342428026	Sep 26, 2024 06:38AM
P1	Feature Request	Reorder code based on branch po...		Assigned	0	340100647	Nov 7, 2024 06:42AM
P2	Bug	Segmentation fault in Node.js x6...	--	New	0	338535750	May 3, 2024 08:59AM
P2	Bug	Strange crashes if v8 run on a thr...	--	New	0	42204637	Apr 4, 2024 12:18PM
P2	Bug	Chrome crashes randomly with S...	--	New	0	328850735	Jun 25, 2024 07:08AM
P2	Bug	Maglev seems to be impacting M...		Assigned	0	41495190	Feb 1, 2024 09:15AM
P2	Bug	Array access becomes much slo...		In Progress	0	42204541	Jun 21, 2024 06:18AM

[Figure 44]: Chromium tracker: Maglev issues

Security vulnerabilities happen in any Google Chrome component, and multiple times they can originate from a huge effort to get better performance. Actually, the official **Google Maglev document** (https://docs.google.com/document/d/13CwgSL4yawxuYg3iNlM-4ZPCB8RgJya6b8H_E2F-Aek/edit?tab=t.0#heading=h.djws22xta9wz) illustrates the cycle of optimization that involves **Ignition, Sparkplug, Maglev** and **Turbofan**, and readers can infer the intensive work to make this entire optimization cycle to work well:



[Figure 45]: V8: optimization cycle | Credit: Google

07. Security

This section is the last one, but it will serve as a bridge to the next articles in this series. Google offers a really intensive website, with many branches, where readers can follow all issues related to Chromium engine and its components. The following references are a few that can be an excellent source for learning new concepts, options, security issues and even exploitation techniques:

- **Chromium engine (source code):** https://chromium-review.googlesource.com/q/+-is:wip
- **Chromium open issues:** https://chromium-review.googlesource.com/q/project:chromium/src
- **Sparkplug:** https://issues.chromium.org/issues?q=componentid:1456567
- **Turbofan:** https://issues.chromium.org/issues?q=componentid:1456959
- **Maglev:** https://issues.chromium.org/issues?q=componentid:1456177
- **Ignition:** https://issues.chromium.org/issues?q=componentid:1456960
- **Sandbox:** https://issues.chromium.org/issues?q=componentid:1455745

There are many other ones, but readers have already understood how to find them. If you are searching for a list of exploited V8 Bugs, there is an interesting document on:

- https://docs.google.com/document/d/1njn2dd5_6PB7oZGTmkmoihYnVcJEgRwEFxhHnGoptLk/edit?tab=t.0#heading=h.1q56zf5othwu

Exploited V8 Bugs in 2024

Issue	First Exploited	Description	Exploit requires V8 Sandbox Bypass	Exploit requires JIT compilation	Variant	JavaScript or WebAssembly	Introduced by	Introduced in
41490332	ITWexploiexploiexplo	Incorrect fast-path for JS property deletion in runtime	Yes	Probably	No	JavaScript	Performance Work	2021 (?)
40941600	V8CTF	Type confusion after promise resolution in combination with async stack traces	Yes	No	No	JavaScript	Feature Work	2020
41482183	V8CTF	Incorrect structural optimization in Turboshaft	Yes	Yes	No	JavaScript	Performance Work	2023
41488920	V8CTF	UaF in Maglev during object creation	Yes	Yes	No	JavaScript	Performance Work	2023
330760873	Pwn2Own	Out-of-bounds access in JS enum cache	Yes	Yes	Yes	JavaScript	Performance Work	2017 (?)
330588502	Pwn2Own	Incorrect parsing of Wasm Types	Yes	Probably Not	No	WebAssembly	Feature Work	2023
330759272	Pwn2Own	Unclear ownership of DOMArrayBuffer	Yes	No	No	JavaScript	Feature Work	2023
330563095*	Pwn2Own	WebCodec ArrayBuffer UaF	Yes	N/A	N/A	N/A	N/A	2022

[Figure 46]: Exploited V8 Bugs in 2024 (truncated list)

Over time, Google has added different protection to prevent exploitation, mainly ones related to memory corruption vulnerabilities and variants, which can result in remote code execution (RCE) as we have seen happening with Chrome renderer component. Besides constant improvements, new languages such as Rust in certain areas (it can cause meaningful performance problems), new fuzzing techniques, better garbage collection approach, better architecture, security protections and measures such as **V8 Sandbox**, **memory tagging** (a hardware feature), recently **V8 CFI**, **libc++ hardening** (from LLVM, and adopted by Chrome) and other ones have been introduced into the project and are expected to prevent the exploitation of existing vulnerabilities. Even other measures that were adopted and not only for security purposes such as **pointer compressions** in heap also represents a good security fit because transform almost-all pointers (references) from an object inside the V8 heap to another object in a 32-bit offset (having as base address the own heap), and leaving few objects that hold 64-bit raw-pointers to another place, and are exactly these remaining 64-bit pointers that comes from ArrayBuffers and TypedArray backing store pointers that will be addressed by the sandbox protection.

About the sandbox protection, it is completely suited to mitigate memory corruption attacks, being that there are many available memory primitives to attackers and, worst, part of them originate from logic errors, which makes using protected languages not effective. In a few ways, the sandbox protection acts by limiting (isolating) the heap memory to prevent any corruption spills to other memory regions, and it is accomplished by moving V8 heap into a protected region (the sandbox) and banishing all 64-pointers. The fundamental idea is to replace all pointers (for V8 heap objects and any other one) with references inside the sandbox's memory (for example, 1 TB), where this new address points to a table with the pointer to the correct address (and data type), preventing type confusion vulnerabilities and it still limits any external arbitrary write to a well-established boundary (there are also guard pages around allocated regions). In other words, the sandbox protection is imposing a redirection to a table inside the sandbox using a relative address (offset) and not an absolute address. Therefore, potential attacks will change the offset, size of an object and even data inside the sandbox, but they can reach anywhere outside there.

There are interesting sandbox options that can be evaluated using v8/d8:

- **sandbox-testing:** this option, which is useful to demonstrate and validate bypasses, exposes memory corruption API, and helps to detect violation of the sandbox protections (rules).
- **sandbox-fuzzing:** this option exposes the memory corruption API, which enables the sandbox crash filter, causing all crashes that are not sandbox violations to be ignored.

In terms of source code, the following places are recommended to be checked because they contain relevant details of sandbox implementation:

- **src\sandbox\sandbox.h:** it is a primary file to understand the goal of sandbox mechanisms, basic operations and first declarations.
- **src\sandbox\sandbox.cc:** it contains the actual sandbox implementation itself, and it is well-commented.
- **src\sandbox\sandboxed-pointer.inl.h:** reading and writing operations of sandboxed pointers have their own functions implemented here. Declarations (prototypes) are in **src\sandbox\sandboxed-pointer.h.**

In the next two figures are shown a piece of **src\sandbox\sandbox.h** and **src\sandbox\sandboxed-pointer.inl.h** files, which contain good details:

```
15: namespace v8 {
16: namespace internal {
17:
18: #ifdef V8_ENABLE_SANDBOX
19:
20: /**
21:  * The V8 Sandbox.
22:  *
23:  * When enabled, V8 reserves a large region of virtual address space - the
24:  * sandbox - and places most of its objects inside of it. It is then assumed
25:  * that an attacker can, by exploiting a vulnerability in V8, corrupt memory
26:  * inside the sandbox arbitrarily and from different threads. The sandbox
27:  * attempts to stop an attacker from corrupting other memory in the process.
28:  *
29:  * The sandbox relies on a number of different mechanisms to achieve its goal.
30:  * For example, objects inside the sandbox can reference each other through
31:  * offsets from the start of the sandbox ("sandboxed pointers") instead of raw
32:  * pointers, and external objects can be referenced through indices into a
33:  * per-Isolate table of external pointers ("sandboxed external pointers").
34:  *
35:  * The pointer compression region, which contains most V8 objects, and inside
36:  * of which compressed (32-bit) pointers are used, is located at the start of
37:  * the sandbox. The remainder of the sandbox is mostly used for memory
38:  * buffers, in particular ArrayBuffer backing stores and WASM memory cages.
39:  *
40:  * As the embedder is responsible for providing ArrayBuffer allocators, V8
41:  * exposes the virtual address space backing the sandbox to the embedder.
42:  */
43: class V8_EXPORT_PRIVATE Sandbox {
44:  public:
45:   // +-  ~~~  -+----------------------------------------  ~~~  -+-  ~~~  -+
46:   // |  32 GB  |                 (Ideally) 1 TB                 |  32 GB  |
47:   // |          |                                                |         |
48:   // | Guard    |      4 GB      :  ArrayBuffer backing stores,  | Guard   |
49:   // | Region   |    V8 Heap     :  WASM memory buffers, and     | Region  |
50:   // | (front)  |     Region     :  any other sandboxed objects. | (back)  |
51:   // +-  ~~~  -+----------------+-----------------------  ~~~  -+-  ~~~  -+
52:   //           ^                                                ^
53:   //           base                                             end
54:   //           < - - - - - - - - - - - - size - - - - - - - - - - >
55:   // < - - - - - - - - - - - - - reservation_size - - - - - - - - - - - - >
56:
```

[Figure 47]: src\sandbox\sandbox.h (truncated)

```
13: namespace v8 {
14: namespace internal {
15:
16: V8_INLINE Address ReadSandboxedPointerField(Address field_address,
17:                                             PtrComprCageBase cage_base) {
18: #ifdef V8_ENABLE_SANDBOX
19:   SandboxedPointer_t sandboxed_pointer =
20:       base::ReadUnalignedValue<SandboxedPointer_t>(field_address);
21:
22:   Address offset = sandboxed_pointer >> kSandboxedPointerShift;
23:   Address pointer = cage_base.address() + offset;
24:   return pointer;
25: #else
26:   return ReadMaybeUnalignedValue<Address>(field_address);
27: #endif
28: }
29:
30: V8_INLINE void WriteSandboxedPointerField(Address field_address,
31:                                           PtrComprCageBase cage_base,
32:                                           Address pointer) {
33: #ifdef V8_ENABLE_SANDBOX
34:   // The pointer must point into the sandbox.
35:   CHECK(GetProcessWideSandbox()->Contains(pointer));
36:
37:   Address offset = pointer - cage_base.address();
38:   SandboxedPointer_t sandboxed_pointer = offset << kSandboxedPointerShift;
39:   base::WriteUnalignedValue<SandboxedPointer_t>(field_address,
40:                                                 sandboxed_pointer);
```

[Figure 48]: src\sandbox\sandboxed-pointer.inl.h (truncated)

The general concepts related directly or indirectly to V8 sandboxes bring us the following sequences and possibilities:

- V8 object → sandboxed pointer → sandboxed object (ArrayBuffer or TypedArray).
- V8 object → code pointer table → entry composed by a pointer (code pointer) → JIT Code
- V8 object → trusted pointer table → entry composed by type + trusted pointer → bytecode
- V8 object → external pointer table → entry composed by type + pointer → Dom node (external object)

A series of details provides us with a better understanding of the sandbox context:

- The key concept of V8 sandbox is to **place the object accessed directly by V8 into a sandbox and does its access through an offset rather using raw pointers**.
- These offsets are named **sandboxed pointers**, which can only access an object inside the sandbox.
- **Sandboxed pointers** are shifted by 40 bits, limiting them to less the 1 TB, which is the size of the V8 sandbox.
- The **pointer compression cage** (4 GB) is "moved" into the beginning of the sandbox heap, and V8 Objects refer to other one also using a sandboxed pointer.
- **Code pointers** are sandboxed to prevent any rogue execution, and this task is done using a code pointer table and ensuring that calls go to a valid function entry point and that the signature and calling convention match are in accordance with the expected.
- Objects that include **BytecodeArrays** and **DeoptmizationData** are natural target for attackers that aim to corrupt memory to escape from the V8 sandbox.
- Objects resident out of the sandbox are always referred through the **pointer table**.
- At the practical level, the sandbox introduction aims to **remove all raw pointers from v8 heap**, using two approaches such as either transforming the raw pointer into an offset that has as base address the start of the sandbox memory (it is known as sandboxed pointer) or turning them into a table of pointers (formally known as external pointer table) that is out of the sandbox.
- **Handles (indexes)** can eventually be corrupted, but the pointers cannot. Additionally, such indexes are shifted, which is similar to SST (System Service Table) from Windows operating system, but there the shifting is by 4 (a nibble) and on V8 sandbox (in special, the external pointer table) holds each pointer shift by 8. This shifting by 8 protects the table of out-of-bound access vulnerabilities by limiting the unshifted handle (index) to access beyond the number of entries.
- The accessed value of an **external pointer table** must be valid, or a crash happens.
- Following the same approach, every access done through the handle, which can be manipulated, is submitted to a type checking when it is dereferenced.
- The **garbage collector** manages the pointer table, which holds the object pointers and respective tags (types).
- Objects referenced by the pointer table can be **executable code**, **objects managed** by the V8 garbage collector and **non-V8 objects** that are not managed by the V8 garbage collector.
- Referred buffers that are out of the sandbox usually have length information associated with them.
- **External objects** accessed from the sandbox must be thread-safe.
- **Trusted objects** are part of the V8 that do not need to be kept inside the V8 sandbox and prevent any direct pointer corruption.
- **Trusted objects** are referred to by the **trusted pointer table**, which contains a full pointer to a **HeapObject** that resides in a trusted space.

- Pointers that are inside the **trusted space** are compressed (there is a pointer compression cage).
- These **trusted pointers** cannot be referred through direct pointers from the sandbox, and any reference to them must be done through **pointer table indirection**.
- Due to the trusted space, a **new trusted heap space** is added to V8.

If there is not an appropriate code pointer protection, it would be possible to corrupt the pointer and jump to another unexpected function, middle of code or even using JIT spraying followed by execute arbitrary code. Concisely, **V8 exploits access to user-mode applications and mechanisms through raw pointers (mainly with RWX access), and that is the real motivation for implementing a sandbox.** Readers can verify the following files that contain trusted pointer table definitions and implementations:

- **src\sandbox\trusted-pointer-table.h**
- **src\sandbox\trusted-pointer-table.cc**
- **src\sandbox\trusted-pointer-table-inl.h**

A piece of code from **trusted-pointer-table.h** is shown below:

```
22: class Isolate;
23: class Counters;
24:
25: /**
26:  * The entries of a TrustedPointerTable.
27:  *
28:  * Each entry contains an (absolute) pointer to a TrustedObject.
29:  */
30: struct TrustedPointerTableEntry {
31:   // Make this entry a "regular" entry, containing an absolute pointer to a
32:   // TrustedObject.
33:   inline void MakeTrustedPointerEntry(Address pointer, IndirectPointerTag tag,
34:                                       bool mark_as_alive);
35:
36:   // Make this entry a freelist entry, containing the index of the next entry
37:   // on the freelist.
38:   inline void MakeFreelistEntry(uint32_t next_entry_index);
39:
40:   // Make this entry a zapped entry. Zapped entries contain invalid pointers.
41:   inline void MakeZappedEntry();
42:
43:   // Retrieve the pointer stored in this entry. This entry must be tagged with
44:   // the given tag, otherwise an inaccessible pointer will be returned.
45:   // This entry must not be a freelist entry.
46:   inline Address GetPointer(IndirectPointerTag tag) const;
47:
48:   // Store the given pointer in this entry while preserving the marking state.
49:   // This entry must not be a freelist entry.
50:   inline void SetPointer(Address pointer, IndirectPointerTag tag);
```

[Figure 49]: Trusted pointer table: src\sandbox\trusted-pointer-table.h (truncated)

In terms of vulnerabilities, many bug classes have been explored over time, but as readers will be inspecting a running code using debuggers (GDB or WinDbg) to analyze them, it is really critical to remember V8 can have doubles, SMI (small integers) and pointers, and that it uses pointer tagging to differentiate among them:

- **smi: value << 32: 0xabcdabcd → 0xabcdabcd00000000**
- **double: value is represented as a 64-bit number.**
- **pointer: value & 1: 0x112233445566 → 0x112233445567**

Therefore, **for addresses you receive from debugger, you must remember to subtract 1 before starting any analysis because they are pointers that follow the rule above**. Instead of providing you with a new script, it is better to elaborate a fast example to show a few further details. Open a terminal and execute the command from the V8 folder:

- (windows terminal) **out\x64.debug\d8.exe --allow-natives-syntax**
- Attach the WinDbg to d8.exe process: **File → Start Debugging → Attach to Process → pickup d8.exe and click on Attach.**
- Enter **g + ENTER** on WinDbg
- On d8.exe prompt, run: **var hacker = [1.2, 3.9, 5.4, 6.7]**
- On d8.exe prompt, run: %**DebugPrint(hacker)**

You will see the following result:

```
C:\articles\chromium\v8>out\x64.debug\d8.exe --allow-natives-syntax
V8 version 13.2.0 (candidate)
d8> var hacker = [1.2, 3.9, 5.4, 6.7]
undefined
d8> %DebugPrint(hacker)
DebugPrint: 00000286002887A1: [JSArray]
 - map: 0x02860008d145 <Map[16](PACKED_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x02860008cab1 <JSArray[0]>
 - elements: 0x028600288779 <FixedDoubleArray[4]> [PACKED_DOUBLE_ELEMENTS]
 - length: 4
 - properties: 0x028600000775 <FixedArray[0]>
 - All own properties (excluding elements): {
    0000028600000DC1: [String] in ReadOnlySpace: #length: 0x02860024fecd <Access
orInfo name= 0x028600000dc1 <String[6]: #length>, data= 0x028600000069 <undefine
d>> (const accessor descriptor, attrs: [W__]), location: descriptor
 }
 - elements: 0x028600288779 <FixedDoubleArray[4]> {
           0: 1.2
           1: 3.9
           2: 5.4
           3: 6.7
 }
000002860008D145: [Map] in OldSpace
 - map: 0x02860008183d <MetaMap (0x02860008188d <NativeContext[301]>)>
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - unused property fields: 0
 - elements kind: PACKED_DOUBLE_ELEMENTS
 - enum length: invalid
 - back pointer: 0x02860008d101 <Map[16](HOLEY_SMI_ELEMENTS)>
 - prototype_validity cell: 0x028600000ab1 <Cell value= 1>
 - instance descriptors #1: 0x02860008d0c9 <DescriptorArray[1]>
 - transitions #1: 0x02860008d16d <TransitionArray[5]>
   Transitions #1:
     0x028600000e85 <Symbol: (elements_transition_symbol)>: (transition to HOLEY
_DOUBLE_ELEMENTS) -> 0x02860008d189 <Map[16](HOLEY_DOUBLE_ELEMENTS)>
 - prototype: 0x02860008cab1 <JSArray[0]>
 - constructor: 0x02860008c78d <JSFunction Array (sfi = 000002860025548D)>
 - dependent code: 0x028600000785 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0

[1.2, 3.9, 5.4, 6.7]
d8>
```

[Figure 50]: d8 session controlled by WinDbg

Nothing is really different from the previous script, except that this time I have created a simple array on runtime with integer and floating pointer numbers. On WinDbg you have to press break and now you will examine some values shown on d8 debugger, but from WinDbg point of view. Once again, remember to subtract 1 from pointers:

```
0:046> dd 0x00000286002887A1-1
00000286`002887a0  0008d145 00000775 00288779 00000008
00000286`002887b0  0000010d 00bab932 00000adc 7566280a
00000286`002887c0  6974636e 29286e6f 220a7b20 20657375
00000286`002887d0  69727473 3b227463 2f2f0a0a 6d204120
00000286`002887e0  2065726f 76696e75 61737265 7473206c
00000286`002887f0  676e6972 20796669 74616874 70757320
00000286`00288800  74726f70 6f6d2073 74206572 73657079
00000286`00288810  61687420 534a206e 0a2e4e4f 55202f2f
0:046> dq 0x028600288779-1
00000286`00288778  00000008`000008d1 3ff33333`33333333
00000286`00288788  400f3333`33333333 40159999`9999999a
00000286`00288798  401acccc`cccccccd 00000775`0008d145
00000286`002887a8  00000008`00288779 00bab932`0000010d
00000286`002887b8  7566280a`00000adc 29286e6f`6974636e
00000286`002887c8  20657375`220a7b20 3b227463`69727473
00000286`002887d8  6d204120`2f2f0a0a 76696e75`2065726f
00000286`002887e8  7473206c`61737265 20796669`676e6972
0:046> .printf "%f\n",3ff33333`33333333
1.200000
0:046> .printf "%f\n",400f3333`33333333
3.900000
0:046> .printf "%f\n",40159999`9999999a
5.400000
0:046> .printf "%f\n",401acccc`cccccccd
6.700000
```

[Figure 51]: Examining values from d8 output on WinDbg

Maybe the highlighted picture above is enough to show the most important points because I have followed the same color from the previous image, and we should remember that each address is composed with the prefix marked in pink, except when it is a value (in the case of the elements). Note a few details:

- **hacker** is a **JSArray** while its elements point to a **FixedDoubleArray**, which has its own Map.

- After the last element of the **FixedDoubleArray**, we have a repetition of values from the JSArray that are not easy to see the correct order in this image because I printed values as 64-bit, but if I repeat the output with **dd** command we will have:

```
0:046> dd 0x028600288779-1
00000286`00288778  000008d1 00000008 33333333 3ff33333
00000286`00288788  33333333 400f3333 9999999a 40159999
00000286`00288798  cccccccd 401acccc 0008d145 00000775
00000286`002887a8  00288779 00000008 0000010d 00bab932
00000286`002887b8  00000adc 7566280a 6974636e 29286e6f
00000286`002887c8  220a7b20 20657375 69727473 3b227463
00000286`002887d8  2f2f0a0a 6d204120 2065726f 76696e75
00000286`002887e8  61737265 7473206c 676e6972 20796669
```

[Figure 52]: Examining additional values

- This time it is clearer to understand that repeated fields are **JSArray Map's address, properties' address,** and **elements' address.**

- The **Map** defines that type of object, how it should be accessed and where elements are in memory. Additionally, if tagged as **FastProperties**, it is faster accessing its properties than other forms of representation such as, for example, dictionaries. Additional information later.

- The value 8d1 seems to be part of the back pointer (it points to a previous object), but it needs to be shifted left by 8. Therefore, when we do it and check the destination of this back pointer, we have JSArray's Meta Map address:

```
0:048> dd 0x02860008d100
00000286`0008d100  0008183d 31040404 05000847 0a0007ff
00000286`0008d110  0008cab1 0008c83d 0008d0c9 00000785
00000286`0008d120  00000ab1 0008d129 00000ba9 0000000a
00000286`0008d130  00000000 00000000 00000002 00000e85
00000286`0008d140  0008d147 0008183d 31040404 11000847
00000286`0008d150  0a0007ff 0008cab1 0008d101 0008d0c9
00000286`0008d160  00000785 00000ab1 0008d16d 00000ba9
00000286`0008d170  0000000a 00000000 00000000 00000002
0:048> dd 0x02860008d145-1
00000286`0008d144  0008183d 31040404 11000847 0a0007ff
00000286`0008d154  0008cab1 0008d101 0008d0c9 00000785
00000286`0008d164  00000ab1 0008d16d 00000ba9 0000000a
00000286`0008d174  00000000 00000000 00000002 00000e85
00000286`0008d184  0008d18b 0008183d 31040404 15000847
00000286`0008d194  0a0007ff 0008cab1 0008d145 0008d0c9
00000286`0008d1a4  00000785 00000ab1 0008d1b1 00000ba9
00000286`0008d1b4  0000000a 00000000 00000000 00000002
```

[Figure 53]: Examining the JSArray Map and MetaMap object

- There are a few details to be commented on here:

    - The 0x2860008183d address suggests that the Map object (located at 0x2860008d145) has a map pointer that points to a map object at 0x2860008183d (something like map of the map).
    - MetaMap describes the structure of a Map object.
    - From back pointer or the Map's address we get the same content.

- The next element of this **FixedDoubleArray** (that does not exist, but it would be the fifth element, and it would be represented as float point) would point to the Map of the JSArray.

- If an attacker had an **out-of-bound read/write primitive**, it would be possible to read the Map of the JSArray. Additionally, if this attacker could overwrite this Map with another one, it would probable cause **a type-confusion vulnerability** because the JavaScript engine would be waiting for a JSArray, but there would be some other Map there. Even though it is not the same case, it is similar to two BufferView points to the same address, where the first one has a type and the second one expects for another type, so triggering a **type-confusion bug**. It is always interesting to refresh:

- **var array1 = [1.5, 4.8]** **(array of floating values)**
- **var obj1 = {"X":2.5}** **(object)**
- **var obj_array1 = [obj1]** **(array of objects)**

- The Map of an object has the following structure: **dynamic type of the object**, **size of the object**, **properties of the object** (and where such objects are stored in memory), **type of the array elements** and **prototype** of the object.

- Having an out-of-bound read/write primitive, an attacker could overwrite the **object array's map** with **an array of float array's map** (float array's map → object's array's map) ,which should be leaked. In this case, it would be accessing a float element as if it were an object address (**addrof primitive**). The nomenclature **addrof** is because we can retrieve the address of an object (as a float) through the index 0 of a float array. At the end, it works like an operator **addrof(my_array)**, where my_array is a float array and the object's address that I want to retrieved is the first one (index 0). In other words, **addrof** primitive provides us the address of a JavaScript object.

- On the other hand, an attacker could **overwrite float array's map with an object array's map** (object array's map → float array's map) In this case, it would be accessing a memory address of an element (an object address) as if it were a float element (**fakeobj primitive**). The nomenclature **fakeobj** is because we can create (having the correct writing primitive) an object anywhere if we control **object's map** and the **element's pointer**. In this case, the attacker could retrieve the object's address using addrof(obj), and then create a fake object at the top of the retrieved address. At the end, it works like an operator **fakeobj(target_addr)**. In other words, **fakeobj** primitive returns a pointer to an address memory that, according to the context, will be taken and interpreted as an object.

- For example, when an attacker (with an oob-read primitive) accessed array1[2] he/she would be retrieving {"X":2.5} because actually the 2nd index (third element), according to the first bullet on the top of the previous page, is exactly the element's pointer that, in this case, would be the first element of the array of the objects.

- Returning to our example, observe a few addresses before the JSArray's map:

```
0:048> dd 0x000002860028A6A1-1-0x30
00000286`0028a670  3498468c 00099683 000008d1 00000008
00000286`0028a680  33333333 3ff33333 33333333 400f3333
00000286`0028a690  9999999a 40159999 cccccccd 401acccc
00000286`0028a6a0  0008d145 00000775 0028a679 00000008
00000286`0028a6b0  0000010d 322f9b86 00000013 62654425
00000286`0028a6c0  72506775 28746e69 6b636168 00297265
00000286`0028a6d0  0000010d 42216942 00000004 29386428
00000286`0028a6e0  000005bd 00000004 e3e5e7e0 0028a6d1
```

[Figure 53]: Examining the JSArray Map and previous addresses

- Based on the figure above, we have that from JSArray's address (0x2860028A6A1-1) it would be possible to subtract 0x20 and reach FixedDoubleArray's elements (light blue).

- Having read and write primitives allows us to interact with user-mode programs and use such primitives to perform real execution.

As you can see, it is easy to spot all values, and the reason I wanted to remember these debugging steps is that you will use multiple times while investigating vulnerabilities.

Although I have not touched on other concepts and even mechanisms, there are a wide range of them that, eventually, might lead to vulnerabilities when manipulated in a unique way:

- The **Inline Cache (IC)**, which is used to make access to properties faster, might be manipulated because it acts as a mechanism to avoid constant type checking for operands that rarely change. This change is known as **map transition**.

- In a general way, as IC assumes (based on information collected on object as its hidden class | Map) that an operand has a type (hidden class|Map), it caches this type to avoid new verification later. However, if this type has changed for some reason, and the cache has not updated its content, a type-confusion vulnerability can arise.

- There are distinct types of **Inline Cache (IC)**:

    - **Monomorphic:** this type of acts on objects with the same class (Map) that have been repeatedly accessed.

    - **Polymorphic:** this type of cache manages diverse types of objects that are known in advance. of course, the performance is not equivalent to monomorphic because as entries' type can change, they must be checked first.

    - **Megamorphic:** this type of cache handles with a wide range of object types whose access cannot be predicted and evaluated in advance.

    - On **page 40**, I mentioned the code of **src\ic\ic.cc** file (also check **src\ic\ic.h**), which mentions these types of **Inline Cache (IC)** and other ones:

```
53: // Aliases to avoid having to repeat the class.
54: // With C++20 we can use "using" to introduce scoped enums.
55: constexpr InlineCacheState NO_FEEDBACK = InlineCacheState::NO_FEEDBACK;
56: constexpr InlineCacheState UNINITIALIZED = InlineCacheState::UNINITIALIZED;
57: constexpr InlineCacheState MONOMORPHIC = InlineCacheState::MONOMORPHIC;
58: constexpr InlineCacheState RECOMPUTE_HANDLER =
59:     InlineCacheState::RECOMPUTE_HANDLER;
60: constexpr InlineCacheState POLYMORPHIC = InlineCacheState::POLYMORPHIC;
61: constexpr InlineCacheState MEGAMORPHIC = InlineCacheState::MEGAMORPHIC;
62: constexpr InlineCacheState MEGADOM = InlineCacheState::MEGADOM;
63: constexpr InlineCacheState GENERIC = InlineCacheState::GENERIC;
```

[Figure 54]: Inline Cache types (src\ic\ic.cc)

- I recommend readers examine code from classes (and respective functions) such as **IC, LoadIC, StoreIC, KeyedStoreIC** and **KeyedLoadIC**.

While debugging scripts using d8.exe you will be using a series of commands that have been proved useful while investigating vulnerabilities and testing proof-of-concepts (PoCs) because they influence directly or indirectly on the V8 execution:

- **%SetBreakpoint(target, 15):** it sets a breakpoint at line 15 of the target function.
- **%ClearBreakpoint(target, 15):** it clears the breakpoint at line 15 for the target function.
- **%PrepareFunctionForOptmization(target):** it prepares the target function for V8 optimization.
- **%OptimizeMaglevOnNextCall(target):** it enables V8 engine to optimize the target function using Maglev on the next call.
- **%GetOptimizationStatus(target function**): this command returns the optimization status of a given function (target).
- **%NeverOptimizeFunction(target):** this command prevents any optimization of the target.

In terms of key concepts, the structure of a JavaScript object will be in play all the time, and a refresh about properties and elements are suitable for clearing pending doubts. Given the following objects:

- obj_1 = {x: “exploit”, k: “again”}
- array_1 = [“cyber”, “security”]

The following observations can help you:

- “x” and “k” are **named properties**.
- the key (x a k) does not tell about the position.
- “cyber” and “security” are **elements**, where 0 and 1 as array-indexed properties.
- **properties** and **elements** are stored in different data structures.
- **properties** and **elements** can either be arrays or dictionaries.
- the composition of an object is given by the **hidden class**.
- if a property is added then that object’s hidden class is modified, but this effect does not happen whether an object is created with a new additional property, and in this case a sub-class of the hidden classes is created.
- **in-object properties** are stored in the object itself while normal properties are stored in a separated data structure.
- **fast properties** are stored in the properties store and accessed by index.
- as **slow properties** are stored in a dictionary, their access is slower than usual but adding and removing them are easier because meta-information is not stored in the hidden class.
- **fast elements** are internal arrays whose indexes map to index in the element store.
- **fast elements** can be packed or holey (with holes between two elements) and containing **smi values** (integers) or double.
- an array containing only smi elements is **Smi Packed**.
- an array containing double elements is **Double Packed**.
- there are special methods named **ElementsAccesso**r to access elements from a backing store, which are represented in several versions for each Array function.

On this subject, readers can read details in different and informative files:

- **src\objects\elements.cc**
- **src\objects\elements.h**

08. References

For readers that might be interested in learning details about topics mentioned here, a brief list of valuable resources follows below:

- **Chromium project:** https://www.chromium.org/Home/
- **Chrome V8 documentation:** https://v8.dev/
- **Maps (Hidden Classes) in V8:** https://v8.dev/docs/hidden-classes
- **Pointer Compression in V8:** https://v8.dev/blog/pointer-compression
- **Maglev - V8's Fastest Optimizing JIT**: https://v8.dev/blog/maglev
- **Sparkplug:** https://v8.dev/blog/sparkplug
- **Ignition Design Doc:** https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/edit?tab=t.0
- **Getting garbage collection for free:** https://v8.dev/blog/free-garbage-collection
- **Concurrent marking in V8**: https://v8.dev/blog/concurrent-marking
- **Sparkplug — a non-optimizing JavaScript compiler:** https://v8.dev/blog/sparkplug
- **Turbofan documentation:** https://v8.dev/docs/turbofan
- **An overview of the Turbofan compiler:** https://docs.google.com/presentation/d/1H1lLsbclvzyOF3IUR05ZUaZcqDxo7_-8f4yJoxdMooU/edit
- **Turbofan TechTalk presentation:** https://docs.google.com/presentation/d/1sOEF4MlF7LeO7uq-uThJSulJlTh--wgLeaVibsbb3tc/edit#slide=id.p
- **Deoptimization in V8:** https://docs.google.com/presentation/d/1Z6oCocRASCfTqGq1GCo1jbULDGS-w-nzxkbVF7Up0u0/edit#slide=id.p
- **Turbolizer:** https://github.com/v8/v8/tree/lkgr/tools/turbolizer/
- **Chromium security:** https://www.chromium.org/Home/chromium-security/
- **The V8 Sandbox:** https://v8.dev/blog/sandbox
- **Enable "safe libc++ mode" for hardening:** https://issues.chromium.org/issues/40228527
- **V8 Sandbox - Trusted Space:** https://docs.google.com/document/d/1IrvzL4uX_Zv0k2Iakdp_q_z33bj-qlYF5IesGpXW0fM/edit?tab=t.0#heading=h.diogznvalour
- **V8 Sandbox - Code Pointer Sandboxing:** https://docs.google.com/document/d/1CPs5PutbnmI-c5g7e_Td9CNGh5BvpLleKCqUnqmD82k/edit?tab=t.0
- **V8 Sandbox - External Pointer Sandboxing:** https://docs.google.com/document/d/1V3sxltuFjjhp_6grGHgfqZNK57qfzGzme0QTk0IXDHk/edit?tab=t.0#heading=h.xfofnqanirh0
- **V8 Sandbox - Sandboxed Pointers:** https://docs.google.com/document/d/1HSap8-J3HcrZvT7-5NsbYWcjfc0BVoops5TDHZNsnko/edit?tab=t.0
- **V8 Sandbox - Address Space:** https://docs.google.com/document/d/1PM4Zqmlt8ac5O8UNQfY7fOsem-6MhbsB-vjFI-9XK6w/edit?tab=t.0#heading=h.xzptrog8pyxf
- **Compile Your Own Type Confusions:** https://phrack.org/issues/70/9#article
- **Attacking JavaScript Engines:** https://phrack.org/issues/70/3#article

- **A Deep Dive into V8 Sandbox Escape Technique Used in In-The-Wild Exploit:** https://blog.theori.io/a-deep-dive-into-v8-sandbox-escape-technique-used-in-in-the-wild-exploit-d5dcf30681d4
- **Fast properties in V8:** https://v8.dev/blog/fast-properties
- **Browser Series:** https://seal9055.com/blog/?p=browser_architecture&d=browser

09. Conclusion

This article is only an introduction to Chrome research, where certainly concepts have not been explored in depth, but such explanations can help you and mainly encourage you to dedicate time and effort to learning much more about browsers.

I hope I have enough time in the near future to write a second article about the topic.

Just in case you want to stay connected:

- **Twitter: @ale_sp_brazil**
- **Blog:** https://exploitreversing.com

Keep learning, reversing, and exploiting everything, and I will see you next time!

Alexandre Borges