browser-pwn-v8基础知识

作者：raycp

V8简介

v8引擎是一种js引擎的实现。JavaScript引擎是执行JavaScript代码的程序或解释器。javaScript引擎可以实现为标准解释器或即时编译器，它以某种形式将JavaScript编译为字节码。V8 - 开源，由Google开发，用C ++编写

v8曾经有两个编译器(v5.9之前)：

- full-codegen— 一个简单且速度非常快的编译器，可以生成简单且相对较慢的机器码
- Crankshaft— 一个更复杂的（Just-In-Time）优化编译器，生成高度优化的代码

v8有多进程，主进程负责获取代码，编译生成机器码，有专门负责优化的进程，，还有一个监控进程负责分析那些代码执行比较慢，以遍Crankshaft 做优化，最后还有一个就是GC进程，负责内存垃圾回收。

v5.9之后，TurboFan (https://github.com/v8/v8/wiki/TurboFan)成了新的优化器，中间语言(bytecode)的Ignition，代码性能更高。

参考链接

1. How JavaScript works: an overview of the engine, the runtime, and the call stack (https://blog.sessionstack.com/how-does-

javascript-actually-work-part-1-b0bacc073cf)

2. How JavaScript works: inside the V8 engine + 5 tips on how to write optimized code (https://blog.sessionstack.com/how-javascript-works-inside-the-v8-engine-5-tips-on-how-to-write-optimized-code-ac089e62b12e)

环境安装

ccls+vscode源码审计环境安装

mac 上ccls安装：

```
brew install ccls
```

ubuntu上安装：

```
# 下载ccls源码
git clone https://github.com/MaskRay/ccls
cd ccls
# 在ccls根目录下执行
# 第0步，下载第三方依赖
git submodule update --init --recursive
# 第一步，下载llvm的二进制包
# 这一步可以用任何下载工具代替，只要使用的是这个网址的结果即可
wget -c http://releases.llvm.org/8.0.0/clang+llvm-8.0.0-x86_64-linux-gnu-ubuntu-18.04.tar.xz
# 解压二进制包
tar xf clang+llvm-8.0.0-x86_64-linux-gnu-ubuntu-18.04.tar.xz
# 在当前文件目录下执行cmake 执行结果保存到Release文件夹中
cmake -H. -BRelease -DCMAKE_BUILD_TYPE=Release -DCMAKE_PREFIX_PATH=$PWD/clang+llvm-8.0.0-x86_64-linux-gnu-ubuntu-18.04
cmake --build Release
# 开始编译并安装
cd Release
# 这里使用4线程编译，当然如果你的电脑够强的话，可以直接-j或者使用更搞核数加快编译
make -j4
# 编译完成，安装
sudo make install
```

安装vscode的 `ccls` 插件，在插件中搜索 `ccls` 安装。

配置vscode的 `setting.json` ， `shift+command+p` 输入 `settings` ，选择 `>Preferences: Open Settings (JSON)` ，进行自定义设置，加入：

```
"ccls.launch.command": "/path/to/ccls/Release/ccls",
"ccls.cache.directory": "${workspaceFolder}/.ccls-
cache/"
```

编译v8

因为很多资源都是谷歌域名下面的，访问不到，所以需要设置http代理来下载，因此第一步是配置git的http代理：

```
git config --global http.proxy
http://proxyUsername:proxyPassword@proxy.server.com:po
rt
git config --global http.proxy http://ip:port # 如果没
有密码
```

配置git的代理还不够，因为配置环境的时候也会用到 `curl` 这条命令，所以还得在环境变量设置一个代理。
在 `~/.bashrc` 或 `~/.zshrc` （取决于用的shell）最后面加上这两行命令，其中 `ip:port` 换成http代理：

```
echo 'export http_proxy="http://ip:port"' >> ~/.zshrc
echo 'export https_proxy=$http_proxy' >> ~/.zshrc
```

配置好以后命令行里执行 `source ~/.zshrc` 加载配置文件，然后执行 `curl -v google.com` 看能否得到返回，测试是否配置成功。

首先下载谷歌源码管理器：

```
git clone
https://chromium.googlesource.com/chromium/tools/depot
_tools.git /path/to/depot_tools
```

也可以可能会更快：

```
proxychains git clone
https://chromium.googlesource.com/chromium/tools/depot
_tools.git /path/to/depot_tools
```

导入路径，建议把命令放入 `~/.bashrc` 或着 `~/.zshrc`：

```
export PATH=$PATH:/path/to/depot_tools
或
echo 'export PATH=$PATH:"/path/to/depot_tools"' >>
~/.bashrc
如果用的zsh则是
echo 'export PATH=$PATH:"/path/to/depot_tools"' >>
~/.zshrc
```

然后拉取源码并安装依赖工具，切换到有漏洞版本的v8则使用 `git reset --hard [commit hash with vulnerability]`：

```
mkdir ~/work/browser_pwn/v8
cd ~/work/browser_pwn/v8
fetch v8
# 如果中断了则 gclient sync同步
cd v8
git reset --hard [commit hash with vulnerability] # 如果要切换到有漏洞版本的v8
gclient sync # 同步更新
sudo ./build/install-build-deps.sh # 只需在linux系统中使用
# 安装字体比较慢，可以用sudo ./build/install-build-deps.sh --no-chromeos-fonts
```

编译：

```
cd /path/to/v8  # 进入目录
git pull && gclient sync # 同步更新
tools/dev/gm.py x64.release # 编译 release 版本
tools/dev/gm.py x64.debug # 编译 debug 版本
tools/dev/gm.py x64.release.check # 测试
```

`fetch v8` 的时候文件下下来差不多要2个g，用了代理可能也比较慢，所以较好的方法可能是在主机上 `fetch` 好了然后打包下来再进行后续操作。

编译好后就在 `./out/x64.debug/` 或 `./out/x64.release/` 看到d8的存在。

参考链接

1. [原创]V8环境搭建，100%成功版 (https://bbs.pediy.com/thread-252812-1.htm)
2. v8调试环境搭建(解决遇到的一些问题)

(http://eternalsakura13.com/2018/06/26/v8_environment/)
3. Building V8 from source (https://v8.dev/docs/build)
4. Building V8 with GN (https://v8.dev/docs/build-gn)
5. ccls编译及其在vscode中使用 (https://blog.csdn.net/u013187057/article/details/99323985)
6. ccls–Visual Studio Code (https://github.com/MaskRay/ccls/wiki/Visual-Studio-Code)

v8的调试

v8的调试功能极为丰富，几乎是想要的调试信息都能有。 `--allow-natives-syntax` ，调试v8必加的一个flag，可以使用用很多native 函数，方便调试（release版本中也存在）。

常用用的api：

- `%DebugPrint(obj)` ，打印 `object` 的一切信息(debug版)或地址(release版)
- `%SystemBreak()` ，在 `javascipt` 中下断点 。

v8提供了一个 `gdbinit` 文件，在gdb中通过 `source ./tools/gdbinit` ，就能可视化显示v8的对象结构，可以使用 `job` 等命令显示结构体信息。

示例：

创建 `demo.js` ：

```
var test_array = [1.1, 2.2, 3,3];
%DebugPrint(test_array);
%SystemBreak();
```

进入gdb进行调试：

```
gdb ./out/x64.release/d8
...
pwndbg> set args --allow-natives-syntax ../../demo.js
//设置flag，并指定demo.js
pwndbg> source ./tools/gdbinit  // 加载gdbinit
pwndbg> r
Starting program:
/home/raycp/work/browser_pwn/v8/v8/out/x64.release/d8
--allow-natives-syntax ../../demo.js
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-
gnu/libthread_db.so.1".
[New Thread 0x7ffff661c700 (LWP 93846)]
0x17965204df79 <JSArray[4]>  ## test_array对象地址

Thread 1 "d8" received signal SIGTRAP,
Trace/breakpoint trap.
ERROR: Could not find ELF base!
0x0000555556622f41 in v8::base::OS::DebugBreak() ()
ERROR: Could not find ELF base!
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────────────────────────[
REGISTERS
]──────────────────────────────────────────────────
 RAX  0x0
 RBX  0x5555567fbae0 —▸ 0x7fffffffdf40 ◂—
0x5555567fbae0
 RCX  0x5555565a57a0 (Builtins_CallRuntimeHandler) ◂—
push   rbp
 RDX  0x5555567fbae0 —▸ 0x7fffffffdf40 ◂—
0x5555567fbae0
 RDI  0x0
 RSI  0x7fffffffd7d8 —▸ 0x708ca7804d1 ◂— 0x708ca7805
 R8   0x30fb95241869 ◂— 0x708ca780f
 R9   0x63
```

```
 R10  0xa
 R11  0xfffffffffffffffb
 R12  0x555556882f50 ◂— 0x0
 R13  0x5555567fbb60 —▸ 0x708ca780751 ◂—
0xa600000708ca7807
 R14  0x0
 R15  0x5555568812d8 ◂— 0x1baddead0baddeaf
 RBP  0x7fffffffd760 —▸ 0x7fffffffd788 —▸
0x7fffffffd7a8 —▸ 0x7fffffffd800 —▸ 0x7fffffffd828 ◂—
...
 RSP  0x7fffffffd738 —▸ 0x55555628cbb5 ◂— mov    r14,
qword ptr [rbx + 0x58]
 RIP  0x555556622f41 (v8::base::OS::DebugBreak()+1) ◂—
ret
————————————————————————————————————————————[
DISASM
]————————————————————————————————————————————
 ► 0x555556622f41 <v8::base::OS::DebugBreak()+1>
ret    <0x55555628cbb5>  // 断在了%SystemBreak();
```

通过 %DebugPrint(test_array); 打印出来了地址 0x17965204df79 <JSArray[4]> ，这个是 test_array 对象的信息，如果使用的是debug版本的v8会得到更为详细的信息；因为 %SystemBreak(); 存在的缘故，可以看到程序断在了 OS::DebugBreak() 。

可以在gdb中输入 job 0x17965204df79 看 test_array 的详细信息：

```
pwndbg> job 0x17965204df79  // 查看test_array
0x17965204df79: [JSArray]
 - map: 0x248fe5902ed9 <Map(PACKED_DOUBLE_ELEMENTS)>
[FastProperties]
 - prototype: 0x30fb95251111 <JSArray[0]>
 - elements: 0x17965204df49 <FixedDoubleArray[4]>
[PACKED_DOUBLE_ELEMENTS]
 - length: 4
 - properties: 0x0708ca780c71 <FixedArray[0]> {
    #length: 0x106d6d4401a9 <AccessorInfo> (const
accessor descriptor)
 }
 - elements: 0x17965204df49 <FixedDoubleArray[4]> {
           0: 1.1
           1: 2.2
         2-3: 3
 }

pwndbg> job 0x17965204df49  // 查看test_array的elements
0x17965204df49: [FixedDoubleArray]
 - map: 0x0708ca7814f9 <Map>
 - length: 4
           0: 1.1
           1: 2.2
         2-3: 3
```

同时看到地址是 0x17965204df79 ，实际上地址是 0x17965204df78 ，如下所示：

```
pwndbg> x/10gx 0x17965204df78 // test_array
0x17965204df78: 0x0000248fe5902ed9
0x00000708ca780c71
0x17965204df88: 0x00017965204df49
0x0000000400000000
0x17965204df98: 0x0000000000000000
0x0000000000000000
0x17965204dfa8: 0x0000000000000000
0x0000000000000000
0x17965204dfb8: 0x0000000000000000
0x0000000000000000
pwndbg> x/10gx 0x17965204df48 // test_array的element
0x17965204df48: 0x00000708ca7814f9
0x0000000400000000
0x17965204df58: 0x3ff199999999999a
0x400199999999999a
0x17965204df68: 0x4008000000000000
0x4008000000000000
0x17965204df78: 0x0000248fe5902ed9
0x00000708ca780c71
0x17965204df88: 0x00017965204df49
0x0000000400000000
```

v8因为地址都是八字节对齐，所以最低的三位是没有地址的，因此可以用最地位来表示数据。当表示为指针时会给地址加1，具体信息如下所示（copy自*CTF 2019 oob-v8 (https://changochen.github.io/2019-04-29-starctf-2019.html)）：

```
Value B is an 8 bytes long value //in x64.
If B is a double:
    B is the binary representation of a double
Else:
    if B is a int32:
        B = the value of B << 32 // which mean
0xdeadbeef is 0xdeadbeef00000000 in v8
    else: // B is a pointer
        B = B | 1
```

v8利用的常见步骤

1. 实现两个原语： addressOf 和 fakeObj ： * addressOf ：给定任意 object ， addressOf 能返回出它的地址。* fakeObj ：给定任意地址， fakeObj 将其解析成一个 object 指针并返回。（一般通过数组越界或类型混淆实现）。

2. 实现任意地址读写

3. ROP/Wasm