Deconstructing the V8 Javascript Engine

Enoch Wang, Supervised by Ibrahim Baggili

This material is primarily based upon work supported by National Security Agency (NSA) and Department of Defense (DoD) under grant H98230-20-1-032. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Security Agency or Department of Defense.

1. Lab Overview

This exercise intends to teach students about the structure of the V8 Javascript engine by exploring the layout of it's data structures, and provides students with a Virtual Environment to play with. The Virtual Environment contains a memory image running a V8 application with forensically relevant data that expert users will be able extract.

In materials provided, a memory image was taken of machine containing a Monero cryptominer. The Monero cryptominer was actively using the system's resources to mine to a pool and generate revene to a certain wallet. Using the V8_MapScan plugin, try to extract forensically relevant objects about the person who installed the crytominer.

2. Materials Needed

- Volatility
- VM Ware
- Memory Dump: monero.vmem *Download Link:* `https://unhnewhaven-my.sharepoint.com/:u:/g/personal/hjohn5_unh_newhaven_edu/EQ-HGQxKP6lOlBMLF6v4uykBt7xMlSpQrKL1mqtqiNuhlQ?e=GOOvp3`

3. Learning Objectives

- Demonstrate a understanding of the memory dump acquisition process
- Demonstrate a understanding of the Data Structure layout of the V8 Javascript Engine
- Analyze the memory dump, using Volatility and associated plug-in
- Apply concepts of memory forensics to a real world problem
- Evaluate the results, determine forensically relevant data from a Node.js application

4. Lab Tasks

4.1 Task 1: Set Up

The first part involves porting a packaged VM into your system. The v8-dev VM contains an Ubuntu system with a copy of Volatility 2.6 already installed. Dependencies for Volatility and a developer version copy of the MapScan plugin have already been included in this virtual machine. Users will have to update this plugin with a more modern version of the plugin. A link to the published version of the plugin can be found here.

v8-dev virtual machine *Download Link*: `https://lsumail2-my.sharepoint.com/:u:/g/personal/tghara1_lsu_edu/ER3ba8vLFAhKtwVH_Urn7GEBWlvVLM0RHuD92Noq8F5CSg?e=UveUMu`

4.2 Task 2: Understanding V8

V8 is the open source interpreter developed by Google to enable JavaScript (JS) functionality in Chrome and power other software. Malicious threat actors abuse the usage of JS because most modern-day browsers implicitly trust script code to execute. To aid in incident response and memory forensics in such scenarios, this work introduces the first generalizable account of the memory forensics of the V8 JS engine and provides practitioners with a list of objects and their descriptors extracted from a memory image. These objects can be used to reveal key information about a user and their activity.

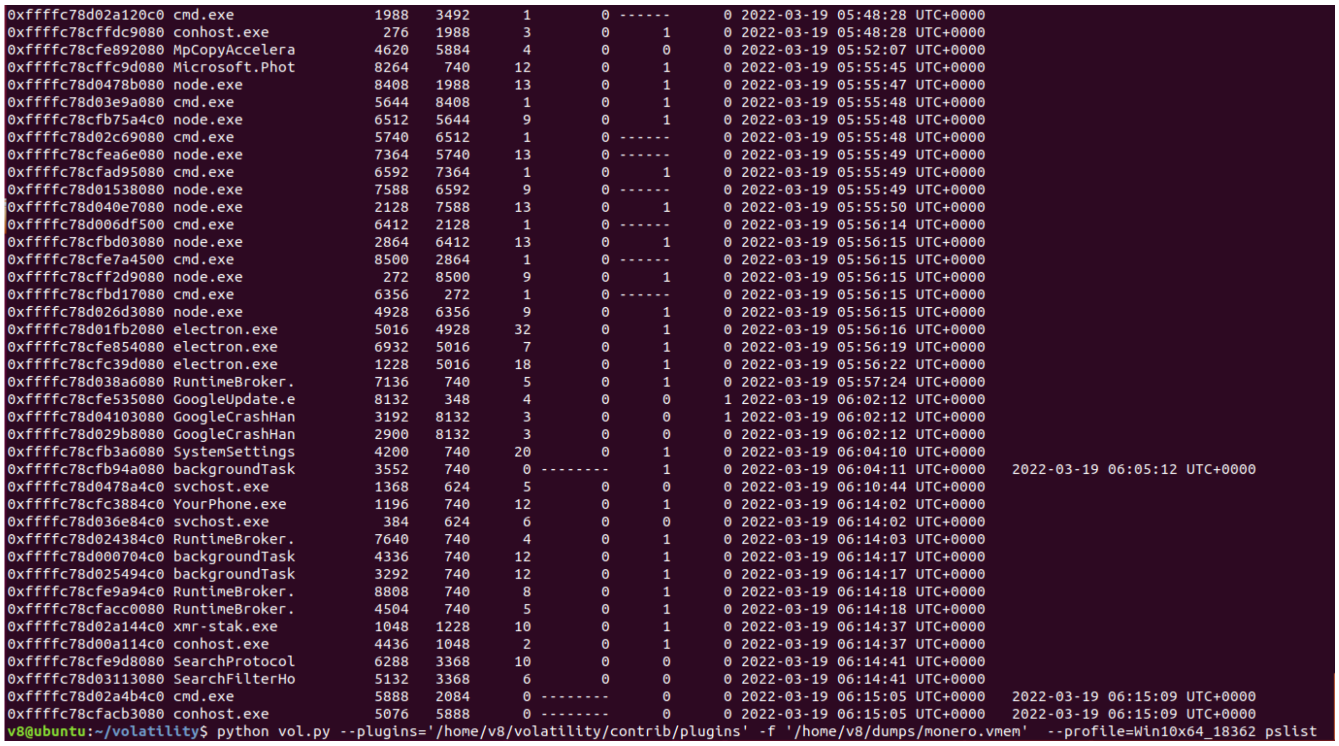
To understand how to effectively extract objects from an application running V8, we must first understand the data structure hierarchy of V8. Objects within V8 are organized through hidden classes that record both the offset for a given object property and the address of the object. These hidden classes can be referred to as object *maps*. V8 uses these maps to categorize various object types by including property descriptors that indicate type and length. These maps also contain a reference to the *SELF* object within its table. Objects vary from type to type and can be represented by an identification number found within the map, known as an *instance type*. The object structure itself can be broken down into three main components: map pointer, properties, and object elements. Object types can consist of basic strings, arrays, user-created objects, and more.

4.2.1 V8 String types

Several of the simplest string types start with a pointer to the map of the object, followed by an integer describing the length of the string, and the properties that create the string. The most basic string type is known as a *ONE_BYTE_INTERNALIZED_STRING_ TYPE*. It is primitive and often contains a short ASCII string. There is also the *CONS_ ONE_BYTE_STRING_ TYPE*, which contains two pointers to two separate one byte internalized string types that are then merged. The type that is directly called the string type contains chunks of code stored in *UTF*.

4.2.2 V8 Structure of arrays

The primary organizational structure of arrays is similar to the basic strings. It is stored as a pointer to the map of the object, followed by an integer describing the length, and succeeded by the properties themselves. Depending on the array type in question, multiple bytes in memory may be allocated for the properties. However, in the simplest types, each property is a single word. *JS* objects are somewhat different. The number of properties must be found through the map, as it is



```
0xffffc78d02a120c0 cmd.exe              1988   3492      1        0 ------      0 2022-03-19 05:48:28 UTC+0000
0xffffc78cffdc9080 conhost.exe           276   1988      3        0      1      0 2022-03-19 05:48:28 UTC+0000
0xffffc78cfe892080 MpCopyAccelera       4620   5884      4        0      0      0 2022-03-19 05:52:07 UTC+0000
0xffffc78cffc9d080 Microsoft.Phot       8264    740     12        0      1      0 2022-03-19 05:55:45 UTC+0000
0xffffc78d0478b080 node.exe             8408   1988     13        0      1      0 2022-03-19 05:55:47 UTC+0000
0xffffc78d03e9a080 cmd.exe              5644   8408      1        0      1      0 2022-03-19 05:55:48 UTC+0000
0xffffc78cfb75a4c0 node.exe             6512   5644      9        0      1      0 2022-03-19 05:55:48 UTC+0000
0xffffc78d02c69080 cmd.exe              5740   6512      1        0 ------      0 2022-03-19 05:55:48 UTC+0000
0xffffc78cfea6e080 node.exe             7364   5740     13        0 ------      0 2022-03-19 05:55:49 UTC+0000
0xffffc78cfad95080 cmd.exe              6592   7364      1        0      1      0 2022-03-19 05:55:49 UTC+0000
0xffffc78d01538080 node.exe             7588   6592      9        0 ------      0 2022-03-19 05:55:49 UTC+0000
0xffffc78d040e7080 node.exe             2128   7588     13        0      1      0 2022-03-19 05:55:50 UTC+0000
0xffffc78d006df500 cmd.exe              6412   2128      1        0 ------      0 2022-03-19 05:56:14 UTC+0000
0xffffc78cfbd03080 node.exe             2864   6412     13        0      1      0 2022-03-19 05:56:15 UTC+0000
0xffffc78cfe7a4500 cmd.exe              8500   2864      1        0 ------      0 2022-03-19 05:56:15 UTC+0000
0xffffc78cff2d9080 node.exe              272   8500      9        0      1      0 2022-03-19 05:56:15 UTC+0000
0xffffc78cfbd17080 cmd.exe              6356    272      1        0 ------      0 2022-03-19 05:56:15 UTC+0000
0xffffc78d026d3080 node.exe             4928   6356      9        0      1      0 2022-03-19 05:56:15 UTC+0000
0xffffc78d01fb2080 electron.exe         5016   4928     32        0      1      0 2022-03-19 05:56:16 UTC+0000
0xffffc78cfe854080 electron.exe         6932   5016      7        0      1      0 2022-03-19 05:56:19 UTC+0000
0xffffc78cfc39d080 electron.exe         1228   5016     18        0      1      0 2022-03-19 05:56:22 UTC+0000
0xffffc78d038a6080 RuntimeBroker.       7136    740      5        0      1      0 2022-03-19 05:57:24 UTC+0000
0xffffc78cfe535080 GoogleUpdate.e       8132    348      4        0      0      1 2022-03-19 06:02:12 UTC+0000
0xffffc78d04103080 GoogleCrashHan       3192   8132      3        0      0      1 2022-03-19 06:02:12 UTC+0000
0xffffc78d029b8080 GoogleCrashHan       2900   8132      3        0      0      0 2022-03-19 06:02:12 UTC+0000
0xffffc78cfb3a6080 SystemSettings       4200    740     20        0      1      0 2022-03-19 06:04:10 UTC+0000
0xffffc78cfb94a080 backgroundTask       3552    740      0 --------      1      0 2022-03-19 06:04:11 UTC+0000   2022-03-19 06:05:12 UTC+0000
0xffffc78d0478a4c0 svchost.exe          1368    624      5        0      0      0 2022-03-19 06:10:44 UTC+0000
0xffffc78cfc3884c0 YourPhone.exe        1196    740     12        0      1      0 2022-03-19 06:14:02 UTC+0000
0xffffc78d036e84c0 svchost.exe           384    624      6        0      0      0 2022-03-19 06:14:02 UTC+0000
0xffffc78d024384c0 RuntimeBroker.       7640    740      4        0      1      0 2022-03-19 06:14:03 UTC+0000
0xffffc78d000704c0 backgroundTask       4336    740     12        0      1      0 2022-03-19 06:14:17 UTC+0000
0xffffc78d025494c0 backgroundTask       3292    740     12        0      1      0 2022-03-19 06:14:17 UTC+0000
0xffffc78cfe9a94c0 RuntimeBroker.       8808    740      8        0      1      0 2022-03-19 06:14:18 UTC+0000
0xffffc78cfacc0080 RuntimeBroker.       4504    740      5        0      1      0 2022-03-19 06:14:18 UTC+0000
0xffffc78d02a144c0 xmr-stak.exe         1048   1228     10        0      1      0 2022-03-19 06:14:37 UTC+0000
0xffffc78d00a114c0 conhost.exe          4436   1048      2        0      1      0 2022-03-19 06:14:37 UTC+0000
0xffffc78cfe9d8080 SearchProtocol       6288   3368     10        0      0      0 2022-03-19 06:14:41 UTC+0000
0xffffc78d03113080 SearchFilterHo       5132   3368      6        0      0      0 2022-03-19 06:14:41 UTC+0000
0xffffc78d02a4b4c0 cmd.exe              5888   2084      0 --------      0      0 2022-03-19 06:15:05 UTC+0000   2022-03-19 06:15:09 UTC+0000
0xffffc78cfacb3080 conhost.exe          5076   5888      0 --------      0      0 2022-03-19 06:15:05 UTC+0000   2022-03-19 06:15:09 UTC+0000
v8@ubuntu:~/volatility$ python vol.py --plugins='/home/v8/volatility/contrib/plugins' -f '/home/v8/dumps/monero.vmem'  --profile=Win10x64_18362 pslist
```

Figure 1: pslist output.

not stored within the object itself. *JS* objects begin with a pointer to their map, similar to the other object types. It is followed by two pointers to arrays. The second pointer will always be to an empty array, while the first may point to the same empty array or an overflow array in a case where there was not enough memory available in the location where the object was stored to hold all properties of the object.

4.3 Task 3: Using the plugin on V8 Processes

Currently the plugin can only be used on Node.js applications. Future iterations of the plugin will have support for more mainstream applications such as Discord and Chrome! First, we want to find the Node.js process containing the core V8 isolate. This can require some trial and error. First we need to know which applications are running V8, we can generate a list of running process by running pslist as shown below. The profile of the memory image will be required when running Volatility commands in order to let Volatility know which version of Windows we are running the command on.

Note that there are multiple processes of Node.js and electron. Both electron and Node.js processes use the V8 engine and thus our plugin would work on all of these processes; However, as mentioned before, not all of these processes will contain the core V8 isolate, which is typically found within the parent process. The easiest way to determine the parent process is by looking for the Parent Process ID (PPID) indicated in pslist. In this example the PPID of electron is 5016. With this PPID we can now run the plugin. The command layout for the plugin should look similar to

```
python vol.py --plugins='<location of plugin>' -f '<location of dump>'
--profile=<image profile> <Plugin Command>
```

To start lets run v8_findalltypes. This command displays the object maps types found in the process and displays their ID in decimal and the number of maps found of each type. The command for this will be as follows.

```
python vol.py --plugins='/home/v8/volatility/contrib/plugins' -f
'/home/v8/dumps/monero.vmem' --profile=Win10x64_18362 v8_findalltypes
```

The output should be a table similar to whats shown below. The name of each object map will be indicated on the left followed by the instance type and map count on the right. This will enable us to identify which object maps are within the process, their instance type, and the number found.

```
$ python vol.py -f dump.vmem v8_findalltypes
Name          Instance Type        Map Count
Error                  1069                1
URIError               1069                1
Object                 1092               28
has                    1052                1
delete                 1053                1
toISOString            1066                1
```

Listing 1: Sample Output from Volatility Plugin

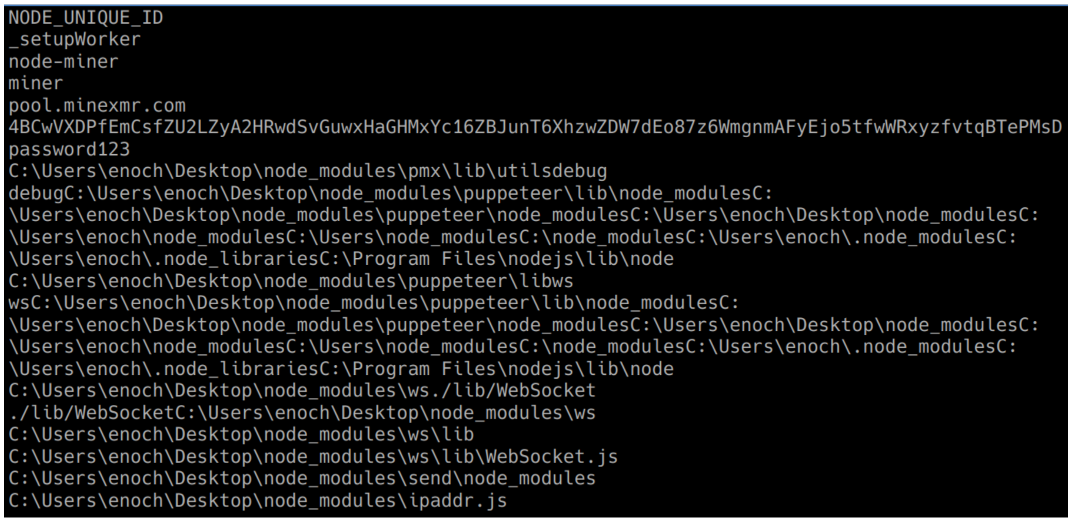
Before we can begin extracting objects, we must first select a object map instance type. There are a large number of object map types but the one we will be focusing on is the *ONE_BYTE_INTERNALIZED_STRING_TYPE*, or 0x8 in hex. To extract the 0x8 types we will simply run the following command.

```
python vol.py --plugins='/home/v8/volatility/contrib/plugins' -f
'/home/v8/dumps/monero.vmem' --profile=Win10x64_18362 v8_extractprops
```

Depending on the number of objects selected the output may vary. Many of the objects extracted by the plugin are printed in raw data and do not have human readable characters. The output of all the objects can be found within the a text document in the directory where Volatility is run. Output of each object printed may look similar to the listing below. The command line is unable to parse the content of the raw object. Hence, You will need hex-viewer to read the content of the file that will be written to the same path where the command line is operating.

```
$ python vol.py -f dump.vmem v8_extractprops
[
    [
        'Property 1',
        'string',
        900186240977,
        'C:\\Users\\Bob\\Desktop\\app.js'
    ],
    ['Property 3', 'smi ', 0],
    ['Property 4', 'smi ', 3]
]
```

Listing 2: Sample Output From Volatility Plugin



```
NODE_UNIQUE_ID
_setupWorker
node-miner
miner
pool.minexmr.com
4BCwVXDPfEmCsfZU2LZyA2HRwdSvGuwxHaGHMxYc16ZBJunT6XhzwZDW7dEo87z6WmgnmAFyEjo5tfwWRxyzfvtqBTePMsD
password123
C:\Users\enoch\Desktop\node_modules\pmx\lib\utilsdebug
debugC:\Users\enoch\Desktop\node_modules\puppeteer\lib\node_modulesC:
\Users\enoch\Desktop\node_modules\puppeteer\node_modulesC:\Users\enoch\Desktop\node_modulesC:
\Users\enoch\node_modulesC:\Users\node_modulesC:\node_modulesC:\Users\enoch\.node_modulesC:
\Users\enoch\.node_librariesC:\Program Files\nodejs\lib\node
C:\Users\enoch\Desktop\node_modules\puppeteer\libws
wsC:\Users\enoch\Desktop\node_modules\puppeteer\lib\node_modulesC:
\Users\enoch\Desktop\node_modules\puppeteer\node_modulesC:\Users\enoch\Desktop\node_modulesC:
\Users\enoch\node_modulesC:\Users\node_modulesC:\node_modulesC:\Users\enoch\.node_modulesC:
\Users\enoch\.node_librariesC:\Program Files\nodejs\lib\node
C:\Users\enoch\Desktop\node_modules\ws./lib/WebSocket
./lib/WebSocketC:\Users\enoch\Desktop\node_modules\ws
C:\Users\enoch\Desktop\node_modules\ws\lib
C:\Users\enoch\Desktop\node_modules\ws\lib\WebSocket.js
C:\Users\enoch\Desktop\node_modules\send\node_modules
C:\Users\enoch\Desktop\node_modules\ipaddr.js
```

Figure 2: Extracted Strings Output

Try to evaluate your understanding of the plugin's capabilities by answering some of the following questions.

1. What processes can the plugin be used on?
2. Which of these processes contain a core V8 isolate?
3. What is the MetaMap for the process we are looking at?
4. How many object maps can be extracted from the plugin?

4.4 Step Four: Analysis

Now that we've extracted the strings, it is time to analyze some of the objects we've extracted. As mentioned before, some objects may not show up in a Linux terminal and may require a third party data analysis tool to view. HexViewer and UltraEdit are excellent choices. The choice of tool may also be limited to how many objects were extracted, as some tools may have a line limit. Once a tool has been selected and our file has been uploaded, we browse the strings for useful artifacts. We can also grep for keywords such as *pool, wallet address, miner, or password.* Doing so may provide us hints on some of the behavior that occurred on this system.

Depending on your string searches, you may find some interesting information about the system's intruder. Below is a sample output of the strings extracted from the process using the V8MapScan plugin. With this try to answer some of the following questions.

1. What was installed on the system
2. What is the pool address?
3. What is the wallet address?
4. Are there any useful strings correlated to this?

5. Submission

A detailed lab report with descriptions of what has been done, what was observed, and what was achieved. Provide screenshots and detailed explanations.

References

[1] Peter Casey, Ananya Yarramreddy, Ibrahim Baggili, *Immersive Virtual Reality Attacks and the Human Joystick*,Manuscript submitted for Publication,2017

[2] Ligh, Michael Hale and Case, Andrew and Levy, Jamie and Walters, Aaron, *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*,2014.

[3] Mike Auty et al. *https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Mal*, 2017

[4] Case, Andrew and Richard III, Golden G, *Memory forensics: The path forward*, Digital investigation, 1-11, 2016