Juicing V8: A Primary Account for the Memory Forensics of the V8 JavaScript Engine

By:
Enoch Wang (University of New Haven), Samuel Zurowski (University of New Haven), Orion Duffy (University of New Haven), Tyler Thomas (University of New Haven), and Ibrahim Baggili (University of New Haven)

From the proceedings of
The Digital Forensic Research Conference
DFRWS USA 2022
July 11-14, 2022

Forensic Science International: Digital Investigation 42 (2022) 301400

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2022 USA - Proceedings of the Twenty-Second Annual DFRWS USA

Juicing V8: A primary account for the memory forensics of the V8 JavaScript engine

Enoch Wang [a, b], Samuel Zurowski [a, b], Orion Duffy [a, b], Tyler Thomas [a, b], Ibrahim Baggili [a, b, *]

[a] University of New Haven Cyber Forensics Research and Education Group (UNHcFREG), Samuel S. Bergami Jr. Cybersecurity Center, USA
[b] Connecticut Institute of Technology at the University of New Haven, USA

ARTICLE INFO

Article history:

Keywords:
Memory forensics
Volatility
V8
Javascript
Memory analysis
Object recovery

ABSTRACT

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, our 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. We analyzed the V8 engine and its garbage collection process. We then developed and validated a Volatility plugin – V8MapScan – to reconstruct V8 objects from a memory image. The runtime of the V8 engine is housed within the V8 isolate which contains its own heap manager and garbage collector. Within the heap of the isolate exists a root object map known as the MetaMap. By using the MetaMap and a *object-fitting* technique, we were able to extract objects, object-maps, and object properties. The V8MapScan plugin scans process memory for the MetaMap data structure contained within the V8 isolate using its data structure, references to objects can be found and extracted. Our findings were verified with Chrome DevTool's *Heap Profiler*. Our approach recovered the majority of objects indicated by the heap profiler with common types such as the ONE_BYTE_INTERNALIZED_STR type returning more than 98.9%. Lastly, we provide a case study using our tools on the Monero Cryptocurrency Miner.

1. Introduction

The V8 engine powers commonly used software such as Electron applications (e.g. Discord, Node.js (Node)). Currently, 95% of web browsing employs some form of JavaScript (JS) and 99% of websites use JS (Tiwari Yan, 2012). Billions of users everyday interact with V8 in some capacity whether in Chrome, Discord, or Electron applications (Google chrome statistics for 2022, 2022; Jargon, 2019). V8 is written in C++ and runs independently within its own thread (Mulazzani et al., 2013). The V8 thread may spawn worker threads to mark areas for Garbage Collection (GC), code optimization, and more (Concurrent marking in v8, 2018). Applications running JS maintain live objects within the heap and are managed solely by the V8 engine (Degenbaev et al., 2018). These live objects can provide critical information to some of the malicious activity preformed through JS.

Malicious threat actors abuse the usage of JS because most modern-day web browsers implicitly trust script code to execute. The majority of websites surfed by users contain JS that is executed. While most JS can be trusted, due to the implicit trust, attackers are enabled to lure victims to their websites and execute malicious code. Many websites have been found to employ the computational power of visitors to mine cryptocurrency without the knowledge of the visitor – known as cryptojacking (Jan et al., 2018).

Furthermore, web browsers have numerous Application Programming Interface (API) capabilities that can be abused by attackers, presenting many client-side attack vectors. Frameworks such as MarioNet have proven that browser-based attacks may run malicious code that persists even after closing the browser (Papadopoulos et al., 2018). There have even been instances of the V8 engine being exploited. Google disclosed a Type Confusion

* Corresponding author. University of New Haven Cyber Forensics Research and Education Group (UNHcFREG), Samuel S. Bergami Jr. Cybersecurity Center, USA.
E-mail addresses: ewang3@unh.newhaven.edu (E. Wang), szuro1@unh.newhaven.edu (S. Zurowski), oduff1@unh.newhaven.edu (O. Duffy), tthom10@unh.newhaven.edu (T. Thomas), ibaggili@newhaven.edu (I. Baggili).

https://doi.org/10.1016/j.fsidi.2022.301400

Vulnerability in V8 (CVE-2021-30551) that was exploited due to logic errors (Glazunov, 2021; Meadows, 2003). Vulnerabilities exploiting the nature of V8 are beneficial to investigate including client side attacks such as cross-site scripting (XSS), session hijacking, and click-jacking.

To counter these attacks, forensic investigators may rely on disk, memory, and network forensics. To gain full reconstruction of JS objects and user activity, post-mortem memory forensics becomes necessary. In this work, we conduct memory analysis of the V8 JS engine. We provide metrics of reliability and completeness of object recovery across Node versions and compare our results to heapdumps analyzed using Google's developer tool used to analyze heap snapshots. Our work makes the following contributions:

- We present the primary account of the memory forensics of the V8 JS engine.
- We present V8MapScan, a series of Volatility plugins capable of locating the V8 objects, their maps and descriptors, capable of displaying the information extracted from applications running Node. We also share a dataset that provides key metrics of how much data can be recovered. The plugin and dataset can be found at https://github.com/unhcfreg/V8-Memory-Forensics-Plugins.
- We outline key data structures within the V8 JS engine and their relationship to each other.
- We demonstrate that our approach is generalizable and works against other applications employing Node's V8 runtime (Discord).

In Sections 2 and 3 we discuss the intricacies of the V8 engine and related work. Section 4 describes the reverse engineering process and the functionality of V8MapScan. Sections 5 presents and discusses our performance metrics. Sections 8 and 7 speculate how this work may be continued and present the implications of our work.

2. Background

In this section, we discuss the memory layout of objects in V8, the V8 GC process, JS forensics, and the llnode plugin. The memory layout for the data structures was discovered through source code analysis (Bak), which is publicly available, and verified through the use of llnode. We present summary details about the tools used and the information we extracted from relevant data structures. We also elaborate on GC and its implications on our work.

2.1. V8 objects

Objects within V8 are organized through hidden classes that record both the offset for a given object property and the address of the object (Ahn et al., 2014). 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 (Chambers et al., 1989). 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.

2.2. 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 Unicode Transformation Format (UTF).

2.3. 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 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.

2.4. Garbage collector

A key factor to consider when performing memory forensics is Garbage Collection (GC). GC varies from engine to engine which affects what objects remain in memory. GC in V8 is a complex process that manages background memory, enabling programs running JS applications to perform seamlessly without jitter or running out of memory. Changes to the Stop-The-World (STW) approach have moved the GC process into a concurrent, parallel, and incremental execution (Inführ; Krylov et al., 2020). STW refers to the algorithmic approach utilized in the Java Virtual Machine (JVM) to halt all running threads during GC (Flood et al., 2001). This prevents race conditions or other complications that may occur while objects are organized by the GC. Although effective, JS applications with STW observe a noticeable impact in real-time performance (Boehm and Demers Scott, 1991).

A full GC of the heap is unnecessary (Li et al., 2018). The V8 JS engine design adopts algorithms that break up the task into smaller sub-tasks managed by a background thread (Bhattacharya et al., 2017). Many of the improvements occur with the Minor GC which exclusively manages the *young generation*. Young refers to the type of objects maintained within the Young Space in memory (Fig. 1). This approach leverages the “Weak Generational Hypothesis” effect, which states that a majority of objects (about 98%), die within a short period (Oracle. 3 generations, 2015). To accommodate this, V8 copies the surviving objects into the old generation space, which consists of the Old Data Space, the Old Pointer Space, and the Large Objects indicated in Fig. 1.

The *old generation* objects reserve the expensive operation of copying for the remaining 2% of live objects that persist through 2 iterations of GC (Ren Ying, 2016). Any dead objects surviving past 2 iterations of GC are left as memory to be collected as implied garbage, as seen in *Object 3* in Fig. 1. This reduces the need for marking surviving objects, finding dead objects, and removing them from memory. Young objects surviving two iterations are promoted into old space, as seen in *Object 2* in Fig. 1. To improve the task of finding dead objects, developers have adopted the technique of root object tracing (Bikineev et al., 2021). Traditionally, dead objects were determined by reference counters which maintain a count of how many times the object is referenced using

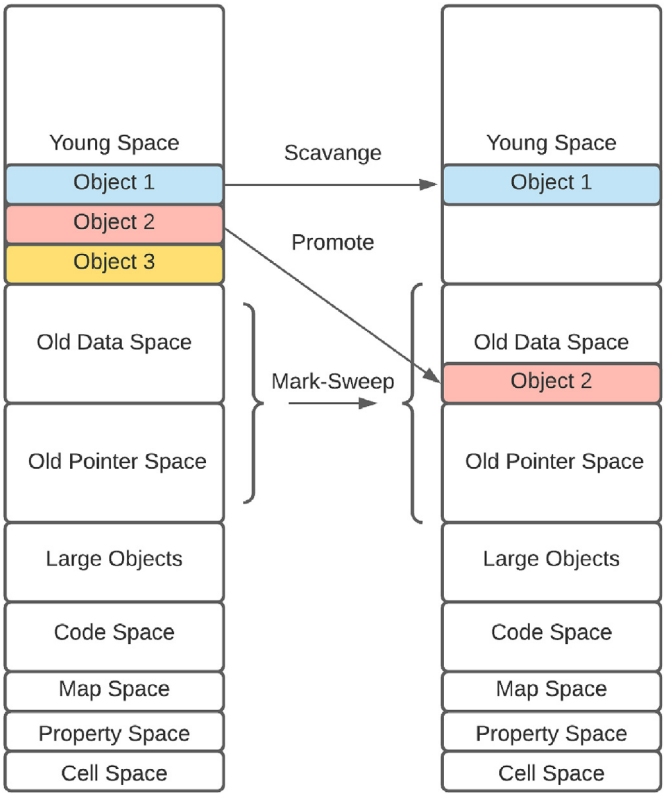

Fig. 1. Memory layout of garbage collector.

reference pointers. When the count reaches zero, the object is deallocated and marked as dead. One of the common issues with reference pointers are reference cycles, a situation where dead objects reference each other preventing the reference counter from ever reaching 0. Determining object lifeline by tracing reachable objects from root objects (Transitive closure) provides a dynamic and robust solution, reducing the amount of code required while recursively determining objects that are in use (Weninger et al., 2018; Levanoni and Petrank, 2001).

The primary effect of GC on our work and memory forensics include the periodic deletion of data stored objects. Thus, GC limits what can be recovered from memory. GC depends on a number of factors including the idle state, heap state, object size stored, type of GC (minor/major), and the application's own implementation of GC. This process makes GC both a nebulous and time restrained process and our results indicated significantly fewer found objects when GC occurs.

2.5. llnode

llnode is a plugin for the Low-Level Debugger (LLDB) for Node. It is the product of the Low-Level Virtual Machine (LLVM) project, a collection of compilers and tools created to support the analysis of applications (Lattner and Adve, 2004). Capable of breaking down high-level data types into low-level primitives, the compiler utilizes a multi-stage optimization system to represent complex data structures into words and tokens readable to humans. Due to the *compile-link-execute* model of high-level languages, binaries created by a specific architecture may not run on a different architecture due to varying instruction sets. LLVM is able to assist in this transition by transforming programs into low-level representations capable of being translated into any architecture (Arthur Lattner, 2002). As a result, LLVM is perfect for Just-In-Time (JIT) applications and has expanded into the development of platform-specific tools such as llnode for Node.

The llnode plugin specifically enables the LLDB to inspect JS objects, object properties, object maps, and the stack frame on active instances of Node. The plugin is able to do this with a *object fitting* method, where the stack frame is scanned and parsed for objects. Objects are determined by fitting a portion of the stack frame into various object-sized chunks with different parameters relative to the data type. If any of the parameters are not met, the object returns false for the object type. This process repeats until all objects within the stack are defined.

The information produced by llnode is useful for back-end developers but the restriction to live instances make it unsuitable for post-mortem forensic examination. We employed the plugin to observe each data structure layout and determine their relational pattern. By creating an instance of V8 within llnode, complete data structures could be pulled from memory. This assisted in the verification of objects extracted with V8MapScan, our Volatility memory forensics plugin. The *object fitting* method utilized in the scan portion of the llnode plugin was also instrumental in the development of V8MapScan.

The redevelopment of the scan process was necessary for our work as the llnode plugin currently does not support Windows and requires a running instance of Node to debug (Kim and Ryou, 2019). The V8MapScan Volatility plugin, on the other hand, is capable of extracting objects from a captured Windows memory image, making it suitable for forensic applications.

3. Related work

This section presents work that is related to the field of memory forensics.

3.1. Forensics related to V8

Discord is a popular messaging application built on Electron (which is built on V8). Discord enables users for Instant Messaging (IM) and Voice over IP (VoIP) which could be used for nefarious purposes such as spreading malware, harassment, and more. A tool called DisFor was developed to extract, analyze, and present Discord client-side artifacts in a forensically sound manner (Iqbal et al., 2021). While this approach can produce forensically sound evidence, it solely focused on disk artifacts rather than volatile memory.

3.2. Memory forensics

Memory forensics involves the acquisition and analysis of volatile Random-access Memory (RAM) on a system. The memory is considered volatile because the data requires a power source to maintain its state. Once a system shuts down, the capacitors begin draining their voltage. These voltages represent the state of a binary bit effectively storing data. Over time, memory forensic techniques, such as cold boot attacks, have been developed to target this specific mechanism. With direct physical access to the RAM, it is possible to slow this process by freezing the RAM, enabling forensic investigators to create a memory image (Ooi and Kam, 2009). Alternatively, memory forensic acquisition tools such as DumpIt, FTK Imager, and Winpmem can be mounted on an external Universal Serial Bus (USB) port to acquire volatile memory from a live system. However, acquisition exists as only the first part of a much greater challenge: interpreting raw memory data.

With access to forensically sound memory images, live memory analysis can be conducted on multiple levels. The first level involves finding all human readable strings from a memory dump with the

use of *grep* (Case and Richard, 2017). Doing so may provide immediate information, such as a password stored in plain-text. However, information stored in applications are subject to structures and symbols which may appear as nonsense without proper translation. Development of memory forensics tools capable of reverse engineering structures to carve relevant data has become an imperative requirement. Similar frameworks have been developed for password managers (Frank and Dewald, 2017), login shells, crypto wallets (Van Der Horst et al., 2017), hypervisors (Graziano et al., 2013) and other applications. This method of custom designing a unique plugin for a specific platform is a necessity as data structuring varies between applications.

The process of developing a plugin on top of a memory forensics tool to reconstruct high-level structures is still at its infancy (Pagani and Balzarotti, 2019). The development of memory forensics tools will continue to provide investigators with meaningful information carved from raw data (Schatz and Cohen, 2017; Tyler et al., 2020, 2021). The techniques of similar projects have been reviewed and inspiration has been taken from their methodology. However, the unique applications for each plugin require a different approach for each project, and a universal approach has yet to be discovered. Furthermore, updates to applications and operating systems continue to break algorithms employed to extract data structures from memory (Lewis et al., 2018). Exploring the memory structures of application engines leads to general approaches applicable to a large number of applications. For example tracing hooks within processes may provide forensically relevant data across multiple programs rather than one type of application (Case et al., 2019).

4. Methodology

In this section, we present the reverse engineering process of V8 and how creating a controlled instance of V8 enabled us to identify magic bytes. We demonstrate how the information obtained from reverse engineering the engine could be leveraged to create an algorithm capable of reconstructing various V8 memory data structures. Finally, this section ends with an explanation of how images were created to simulate a system hosting Node. These images were used to observe V8 running live applications, and to evaluate the efficacy of our approach.

4.1. Reverse engineering V8

The steps involved in reverse engineering V8 consisted of the following:

- Creating a controlled instance of V8 within llnode
- Reading through the code base and identifying indicated data structures within a live instance V8
- Validating relational patterns within the memory structure

The code base revealed two key factors. The first enabled the discovery of objects through a similar tracing methodology performed by V8's GC. The second enabled the verification of each data structure. Each object within V8 could be tied to an object map that identifies information about the object such as descriptions, instance type, and property fields (See Fig. 2). Each object map also contains a reference to the MetaMap, a root map that all maps inherit from. The address of the MetaMap contains within its immediate value a reference to itself +1 (See Fig. 3).

Understanding the object maps enables data extraction about each object including object type code, object type name, and object properties. Each object holds a pointer to its object map and values which are described by the object properties. Thus to properly parse an object, both the object and the object map are necessary. Determining the location of both the object and the object map is possible through a series of recursive Yara-scans. These recursive scans follow the structure of: 1) finding the MetaMap, 2) finding each object map, 3) finding each object.

Determining each field within each data structure was done through correlating llnode outputs with the V8 code base (Bak). Addresses of entire objects could be extracted using llnode but understanding the layout of each object required the code base. Information such as the various sizes of each field was indicated by the developers, resulting in the visual representation provided in Fig. 2. Each field indicated in the object map represents 4 bytes with the exception of the Prototype Tag field which represents 8 bytes. The second field in the object map contains the first and second integer fields which are also subdivided into 1-byte fields. These fields contained identifying information, the most important being the Instance Type. The Instance Type is contained within the second integer in a 2-byte field. Descriptors for the objects matching the map could be extracted from the sixth field in the object map. The majority of objects follow the layout indicated in Fig. 2 but certain types such as custom-defined arrays are exceptions to this layout with primitive types seeing minor changes from the indicated Fig. 2.

4.2. Plugin development

The discovery of the MetaMap facilitates the V8 data structure as it indicates a set of addresses within the V8 isolate. The isolate is an independent copy of the V8 runtime which includes: a heap manager, a garbage collector, and other various functionalities. The MetaMap is the root map that is referenced by a pointer from all object maps. The location of the MetaMap is referenced at the start of all object maps as well as within its own structure. Thus, it is possible to find the MetaMap by scanning for values that match the address +1. Additionally, finding the MetaMap enables searching for object maps that contain a reference to the MetaMap. Fig. 3 shows the layout of a object, object map, and MetaMap's relation through a series of reference pointers. Each reference pointer indicates the address of the next structure. Starting from the value stored within the first address of the object and subtracting 1 from it produces the object map. The value stored in the object map - 1 also contains the MetaMap.

Algorithm 1. Algorithm for V8MapScan

Algorithm 1 Algorithm for V8MapScan

```
rule ← "FF 03 (20 | 40)"
y ← yara.compile(rule)
for process ∈ processList do
    if process ≠ V8_PROCESS then
        continue
    end if
    m ← y.match(process)
    if m ≠ V8_MetaMap then                ▷ Locate MetaMap
        continue
    end if
    yref ← yara.compile(m)                ▷ References to MetaMap
    mapRefs ← yref.match(process)
    for objMap ∈ mapRefs do
        V8Obj ← parseObj(objMap)          ▷ Extract objects
    end for
end for
```

The V8MapScan plugin begins by scanning for a series of magic bytes that are stored within the MetaMap structure. In the

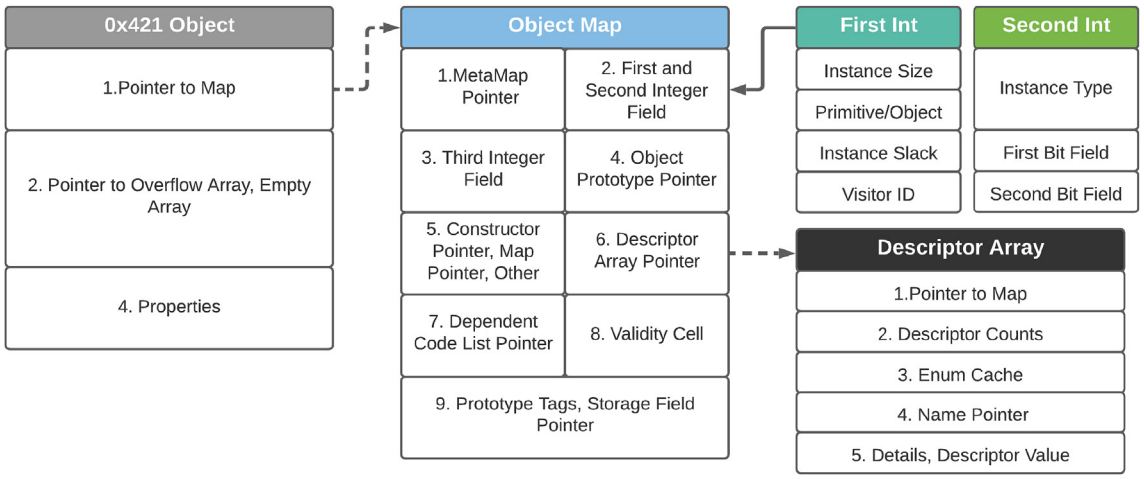

Fig. 2. V8 data structures.

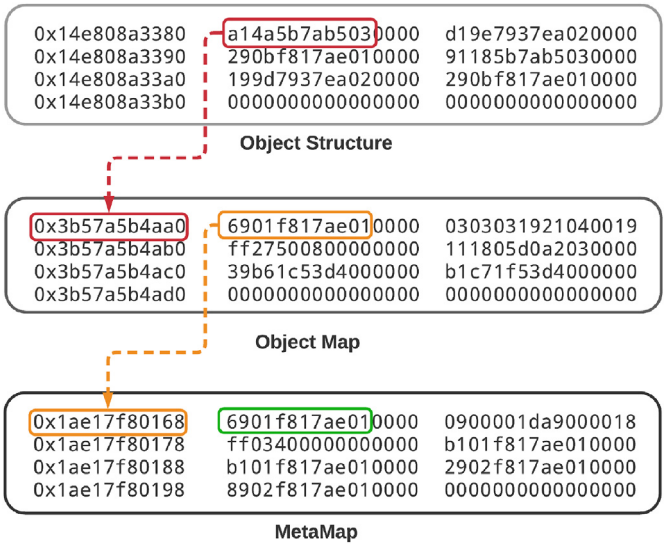

Fig. 3. Memory layout: V8 object, object map and MetaMap

MetaMap structure, shown in Fig. 3 and Algorithm 1, the bytecodes on line one provide a signature that can be directly scanned for. Various versions of Node contain unique bytecode identifiers located four bytes from the base address of the MetaMap. To support multiple versions of Node, the bytecodes *FF* 03 (20 | 40) are used to locate the MetaMap in versions 12 through 15. To verify that the structure is a MetaMap, the base address +1 is compared to the value stored at the address itself. The determination of these magic bytes were acquired using the llnode debugger to simulate an instance of V8. The first set of magic bytes were found from a simulated version of Node version 14.15.1. This method worked for every version of Node after 12.0 with the exception of 15.1.

For finding the MetaMap in version 15.1, Cheat Engine was utilized to conduct a series of pointer scans on known objects. Cheat Engine is an open-source debugger that attaches itself to processes memory in order to manipulate values. These values can directly correlate to variables used within the process. Pointer scans are conducted by scanning for a value located within the process. Scanning for a value will provide the addresses where the value is found. However, not all the addresses contain the value to the desired variable (Casey et al., 2019; Cano, 2016; Feng et al., 2008). To identify the address that correlates to a value, a series of repeat scans using the addresses that stored a manipulated value must be performed until a single address remains. This technique was used to identify the address of custom strings within Node version 15.1 and enabled discovery of the MetaMap by recursively iterating through the reference pointers as demonstrated in 3.

With the MetaMap obtained, the plugin can locate all object maps by scanning for values containing the address of the MetaMap +1. Each map is enumerated and extracted for information relating to the object (descriptors and instance types). To find the objects themselves, another full process scan is conducted with the address of the object map +1. Like the maps, the objects also contain valuable information such as value and properties. The step-by-step procedure is indicated in Algorithm 1 and demonstrates a $O\ (n^2)$ time complexity. Line 1 includes the magic bytes used for identifying the MetaMap structure. Line 2 compiles them into the Yara-scan ruleset. Lines 3 through 10 scan for the MetaMap within the Node process space. Lines 11 and 12 locate all object maps by searching for references of the MetaMap. Lines 13 to 15 locate and extract all objects by searching for references to the object maps. Real-time execution of the plugin is primarily determined by the number of object maps detected within the image.

4.3. Experimentation

To establish evidence that the V8 object recovery approach outlined is accurate, a robust experiment was conducted. Additionally, to produce evidence, an environment had to be created with a custom Node application that enables:

- Creation of Objects
- Deletion of Objects
- Unique Identification of Objects
- Predictability of the total number of V8 Objects within the heap.

A custom application was created with properties that could be instantiated within the source code. A user would define an object in the source code to increment the total number of user objects by one. However, because the class contained other properties, other object types would also increment.

In the experiment, 10 memory dumps were created by employing Windows 10 virtual machines using VMware Workstation. The memory dumps used in this experiment are listed in

Table 1
Memory dump dataset.

Name	Size	Node Version	Application
V8_1_objects.vmem	2 GB	14.15.1	Custom
V8_2_objects.vmem	2 GB	14.15.1	Custom
V8_3_objects.vmem	2 GB	14.15.1	Custom
V8_4_objects.vmem	2 GB	14.15.1	Custom
V8_5_objects.vmem	2 GB	14.15.1	Custom
V8_6_objects.vmem	2 GB	14.15.1	Custom
V8_7_objects.vmem	2 GB	14.15.1	Custom
V8_8_objects.vmem	2 GB	14.15.1	Custom
V8_9_objects.vmem	2 GB	14.15.1	Custom
V8_10_objects.vmem	2 GB	14.15.1	Custom
Agari_ver11_0.vmem	2 GB	11.0	Custom
Agari_ver13_0.vmem	2 GB	13.0	Custom
Agari_ver15_0.vmem	2 GB	15.0	Custom
Discord.vmem	2 GB	N/A	Discord

Table 1 and may be downloaded from our GitHub. The memory images labeled custom were specifically used for the experiment. The number represents how many user objects were created by the custom application written. This was conducted by, executing the Node application, and then taking a snapshot containing all user objects which were created, including all the others created by the V8 runtime. Furthermore, another snapshot was taken using the heapdump Node library to analyze the total number of each object type (Noordhuis). The heapdump library takes advantage of V8's built-in functionality to take a heap snapshot using the heap profiler. The heap snapshot tool can also be loaded into the Chrome DevTools heap profiler for analysis. Furthermore, heapdump is immediately executed within the application once all objects are created. It is almost important to note that invoking this within the application can potentially create more objects. The heapdump approach was employed to quantify the amount of data our proposed approach recovered. The purpose of the experiment was to evaluate the following: 1) statics of the number recovered objects via heapdump, 2) the number of recovered objects, and 3) how garbage collection can affect data recovery in volatile memory.

Once the data was collected, our Volatility plugin was tested. The plugin would create a CSV file containing the following properties: number of user-created objects, object type, and total count of the object type. Once CSV files were created for each memory dump, they were concatenated together for analysis and analyzed within a Jupyter Notebook. The results of the experiments can be found in graphs 4, 5 and Table 2.

4.4. Version testing

To ensure the robustness of the plugin, multiple versions of Node were tested using the images indicated in Table 1. A sample Node application, called Agari (Ronitsinha) was run on multiple versions of Node. A memory snapshot was taken of each version which can be seen Table 1. The tests showed the plugin was able to extract objects, identified by unique strings, in all versions of Node after 13.0. The plugin was not able to identify the MetaMap nor a V8 isolate in version 11.0. Inspection of the process memory from version 11.0 also revealed that certain Node processes could not be dumped from memory.

A version of the plugin was also modified for 32-bit applications such as Discord. Discord is only launched in 32-bit and currently does not have a 64-bit version. The modified plugin was able to consistently locate the MetaMap from the Discord image indicated in Table 1. The modified plugin scanned through every instance of Discord located within the image to determine the structure. The current state of the modified plugin contains no further implementation and does not locate object maps or objects.

Table 2
Object discovered by method statistics.

Instances Type	Heap Count	Plugin Count	Discovered Percent
All String Types	8180	8127	99.3%
Internalized String	5583	5527	98.9%
Array	1677	2699	>100%
ArrayBuffer	23	28	>100%
Global	2	1	50%
User Objects	10	10	100%
Symbols	111	154	>100%

5. Findings and evaluation

5.1. Overview

The plugin developed during this research was tested on Windows 10 (−profile = Win10 × 64_18362). Testing was performed on memory images created specifically with both custom Node applications and Discord. Memory images were captured using snapshot support in VMWare Workstation. Anticipated data was used compare with the plugin output through V8's source code on GitHub. V8's functions that created the JS objects were analyzed to determine all relevant data. This information was then compared to the information recovered by the plugins.

5.2. Plugin output and usability

This section outlines how to use the V8MapScan plugin to analyze memory dumps. Only partial output is shown to provide insight on how it works since the amount of data depending on each memory dump can differ.

5.2.1. v8_extractprops

v8_extractprops can be useful for providing insight on the total objects that can be potentially recovered. Listing 1 shows a sample of a property found in an object and the data residing in it. This plugin was primarily used for dataset creation and analysis. However, it can serve to provide properties of objects that may be forensically interesting to a practitioner.

```
$ 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]
]
```

5.2.2. v8_findalltypes

v8_findalltypes returns all the Object Maps and the descriptors. This includes how many maps there are of each type. v8_findalltypes output can be found in Listing 2 showing the name of the type, the decimal number of the type, and the total map count of the respective type.

```
$ 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
```

5.2.3. v8_extractobjects

v8_extractobjects takes the object Map address and returns all the objects that are inferenced from this address. This can be used to collect relevant information and extracting object information. Example output shown in Listing 3 provides the type, map address, and the address of the object map.

```
$ python vol.py -f dump.vmem v8_extractobjects
Type    Map Address        String Address
String      0x1ae17f80408       0x6963708ce8
String
bindServerHandle
```

5.2.4. v8_instancetypeaddr

v8_instancetypeaddr works by supplying in an instance type number. These numbers can be found in the v8_findalltypes plugin output. The output can be found in Listing 4 where it provides the MetaMap address, and the object address of the instance type. Once the number has been inputted, this plugin will return the address of the object maps found. The purpose of this plugin is to enable investigators to look for specific types and then use other plugins to then conduct deeper analysis on the instance types to see the data it holds. It can contain arrays, strings, and various other information of potential forensic relevance.

```
$ python vol.py -f dump.vmem v8_instancetypeaddr
Enter InstanceType: 1066
Map found: 0x34697840d80L
Number Object Address
     1     0x344bd60bad8
```

5.3. Recovery analysis

It is important to note that metrics in memory forensics is an understudied topic, and a difficult problem since memory is volatile. However, we conducted our analysis given the reasonable constraints memory forensics imposes. The graphs in Figs. 4 and 5 show metrics of the total amount of objects that were discovered by our Volatility plugin. Examined were two specific instance types. The red portion of the bar indicates approximately how much data was not found due to potential GC. This is an approximation, based on the number of increasing objects from all the dumps for each instance type. For example, in Fig. 5, each time a new object was introduced in each memory dump, it would increase by two with the anomaly in which GC potentially occurred. Based on the *y* values, we linearly approximated on the amount of objects that should be found in memory (assuming GC did not occur). It is important to note that it is almost impossible to have 100% certainty that all objects were found because of the indeterminate nature of GC.

For JSObjectType (hex value 0 × 421), most cases seemed to have the maximum value of how many objects should be recoverable. JSObjectType uses a constant approximation since the total number of objects is based on the number of instance types. Additionally, effects of GC can also be seen in Fig. 4. On the *x* axis, 1 and 7 in the all memory dumps had significantly less retrievable data because when those memory images were collected major GCs occurred. This resulted in less retrievable data using our approach. One of the major impacts in object recovery in V8 in GC attribution. When major GCs occurs, it can result in thousands of unrecoverable objects. As a forensic practitioner, this data could be relevant for a case and may pose challenges. In our testing, memory dumps that contained major GC show a substantial smaller amount of recoverable data. However, even if GC occurs the V8MapScan plugin can still show a number of recoverable JSObjectType. Many of the memory images created (specifically 3,4,8,10) show indications that minor GC occurred showing that minor sweeps of GC can result in some unrecoverable data. If GC does not occur, all JSObjectType instances would have been recorded as approximately 4870.

One of the most common string types used in V8 is ONE_BYTE_INTERNALIZED_STR. Results on this type were analyzed and shared in Fig. 5. This type was shown to have an increase of recovered data. The number of user objects were increased in the custom application. The figure also indicates V8MapScan was able to recover more data as the number of user objects increased. Two memory images (one and seven user objects) show major and minor GCs occurred resulting in majority of the data not being able to be recovered. Six user objects created contained the same number of objects retrievable for five user objects. This suggests that a minor GC occurred before the snapshot was taken.

Using the heapdump approach, Table 2 shows some of the recovery percentages on specific V8 types (or a combination of specific types). For some types, our plugin could recover more than the heapdump library within Node meaning that either that memory was potentially going to be GCed or the library is unable to recover that specific data. Some data not found with heapdump may be due to sometimes data being set to null. However, within the heap it should reside because it has not been fully deleted. For most types, results show that in most instances almost 100% of the data is recoverable.

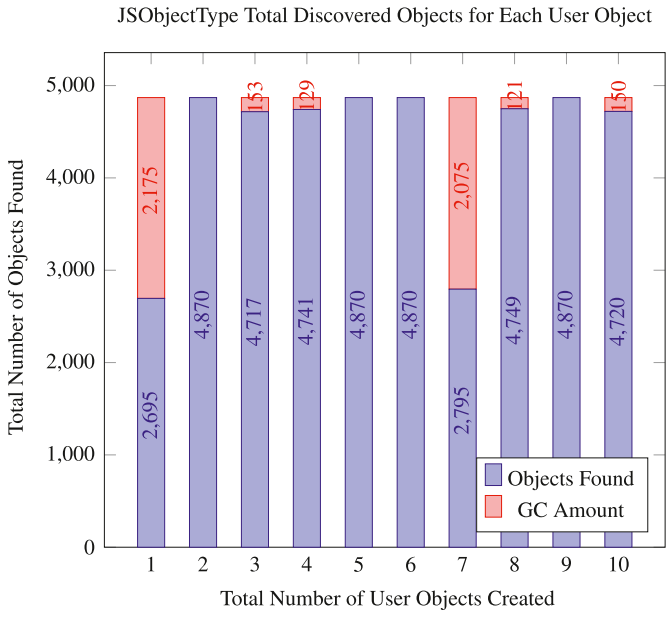

Fig. 4. V8 JSObjectType recovery data.

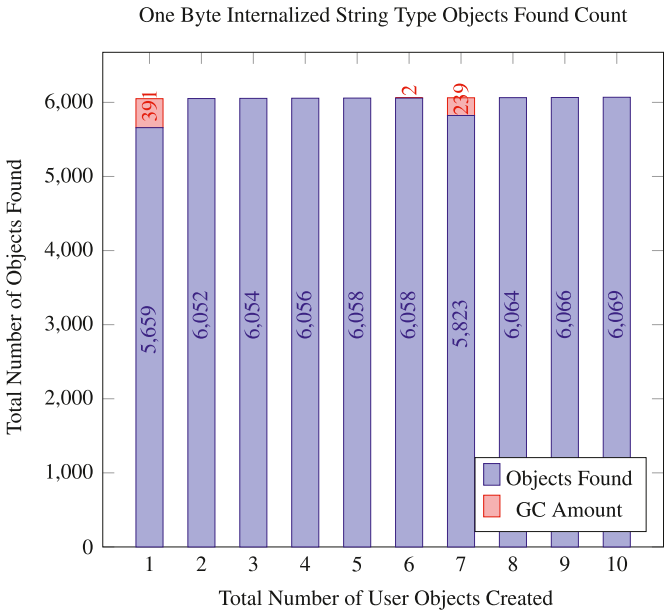

Fig. 5. One Byte internalized string recovery data.

6. Monero miner case study

To show forensic usefulness of our approach we conducted a case study on a cryptocurrency miner (Node Miner). Cryptocurrency miners have been exploited by adversaries in recent times, and thus their forensic examination is appropriate for real-world investigations.

A memory snapshot on the Monero miner was taken and analyzed with our suite of V8 plugins. The Monero (XMR) Miner was running on Node. To setup the case study, we employed a virtual machine with the following properties:

- Windows 10 (Build, 18362)
- Node (14.15.1)
- node-miner designed to easily mine Monero (XMR) and Electroneum (ETN) https://www.npmjs.com/package/node-miner
- The pool pool.minexmr.com:4444
- MyMonero wallet

A wallet was created using the MyMonero application to generate a receiving address for the Monero miner. The node-miner was then run on the pool pool.minexmr.com:4444. Upon the execution of the application, a memory snapshot was taken of the virtual machine. The image was analyzed using the v8_extractprops tool of our plugin which prints all objects that may be useful to the investigator.

In a real world scenario, a practitioner may first employ the utility tool *instance-types-info.py* found on our project's GitHub to locate specific string types that could be forensically relevant (Fig. 6). This utility allows the user to reference the values of the string types that reside within V8. Upon doing this, an investigator can run the *v8_findinstancetypes* on the memory image to verify that JS object exists (Listing 5). Practitioners may then note that instance type "8" exists, therefore, they can attempt to extract the objects. We note that V8 string types do not have a symbolic name which is why they are known as "Invalid Typename". Upon validation, an investigator may then employ v8_extractprops (Listing 1) to query a specific type referencing the *instance-types-info.py* script output. This allows an examiner to correlate specific types of JS objects to their associated values.

```
$ python vol.py -f monero.vmem v8_findalltypes
Name                 Instance Type  Map Count
Invalid Typename                 0          1
Invalid Typename                 2          1
Invalid Typename                 8          1
Error                         1069          2
Uint16Array                   1051          1
ArrayBuffer                   1061          2
```

Next, an investigator extracts the information as shown in Fig. 7 which shows forensically relevant artifacts such as the wallet, URLs, modules used within the V8 process etc. With the increasing hijacking of npm modules, knowing which modules are used in a node application can be extremely useful. The line following "pool.minexmr.com" contains the wallet address used by the cryptocurrency miner. This quick case study illustrates the efficacy of using our constructed techniques in real-world memory forensic scenarios.

7. Conclusion

This paper explored the internal workings of the V8 JS engine and its Garbage Collector. Applications utilizing the V8 JS engine were employed in our evaluation. Objects created in these

```
*************** INSTANCE TYPES FOR STRING ***************
INTERNALIZED_STRING_TYPE                                : 0x00 0000000000000000     0
ONE_BYTE_INTERNALIZED_STRING_TYPE                       : 0x08 0000000000001000     8
EXTERNAL_INTERNALIZED_STRING_TYPE                       : 0x02 0000000000000010     2
EXTERNAL_ONE_BYTE_INTERNALIZED_STRING_TYPE              : 0x0A 0000000000001010    10
UNCACHED_EXTERNAL_INTERNALIZED_STRING_TYPE              : 0x12 0000000000010010    18
UNCACHED_EXTERNAL_ONE_BYTE_INTERNALIZED_STRING_TYPE     : 0x1A 0000000000011010    26
STRING_TYPE                                             : 0x20 0000000000100000    32
ONE_BYTE_STRING_TYPE                                    : 0x28 0000000000101000    40
CONS_STRING_TYPE                                        : 0x21 0000000000100001    33
CONS_ONE_BYTE_STRING_TYPE                               : 0x29 0000000000101001    41
SLICED_STRING_TYPE                                      : 0x23 0000000000100011    35
SLICED_ONE_BYTE_STRING_TYPE                             : 0x2B 0000000000101011    43
EXTERNAL_STRING_TYPE                                    : 0x22 0000000000100010    34
EXTERNAL_ONE_BYTE_STRING_TYPE                           : 0x2A 0000000000101010    42
UNCACHED_EXTERNAL_STRING_TYPE                           : 0x32 0000000000110010    50
UNCACHED_EXTERNAL_ONE_BYTE_STRING_TYPE                  : 0x3A 0000000000111010    58
THIN_STRING_TYPE                                        : 0x25 0000000000100101    37
THIN_ONE_BYTE_STRING_TYPE                               : 0x2D 0000000000101101    45
FIRST_NON_STRING_TYPE                                   : 0x40 0000000001000000    64
```

Fig. 6. Types an investigator should prioritize to investigate.

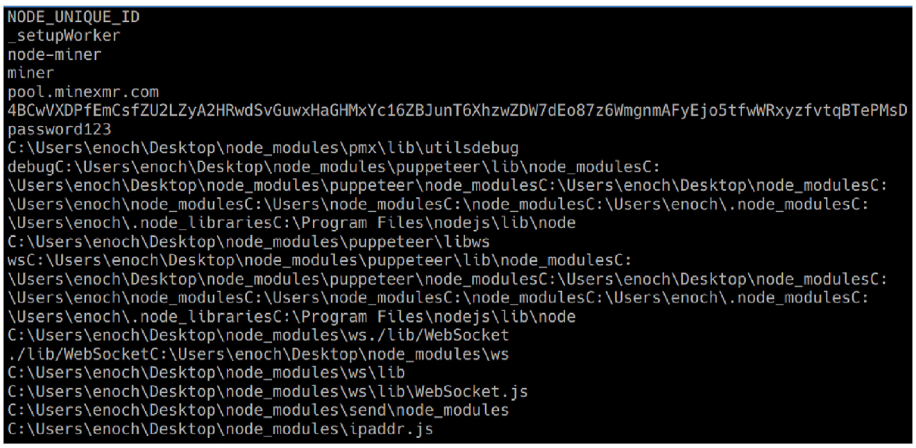

Fig. 7. Truncated output of V8 Strings extracted from memory.

environments were traced through each state of the GC and memory images were taken. These memory images were then analyzed and aided in the development of a new memory forensic tool.

With the information gathered by reviewing the llnode code, V8 code, V8 documentation, and methodologies demonstrated (Van Der Horst et al., 2017), we were able to develop a Volatility plugin designed to extract objects from the V8 JS engine. With it, we managed to extract objects from Node and applications built on Node.

The results were discussed and analyzed to evaluate our constructed plugin. We produced the primary account and tool for the comprehensive memory forensics of the V8 JS engine, as well as a dataset that may be employed in future research.

8. Future work

Support into the applications Discord and Chrome are current plans for the future. The modified 32-bit version supporting Discord can only locate the MetaMap structure because it requires all other functions to convert to 32 bit. Additionally, Chrome implementation of V8 differs from Node and will require a new method capable of parsing multiple sub-processes for multiple V8 isolates.

Furthermore, several case studies can be conducted against client-side attacks, server-side attacks, and extracting messages from Discord. The plugins can provide investigators insight on how to discover relevant forensics artifacts in memory. The case study will also analyze the time to determine how long JS artifacts reside in memory.

One of the plugins developed v8_findalltypes, instances of the type names will be indicated as null because the current location searched for these names does not always contain the symbolic name. To accommodate this, properties identified as null require a different search algorithm to find the proper name. This approach would require more memory analysis with lldb as they reside in a different location in V8's engine.

In addition, a significant improvement includes accommodating changes to Volatility. The plugin in the future should be upgraded to Volatility3 as Python2 is at the end of its life. Volatility3 provides many feature enchantments that could be used and result in better performance and long-term support. Volatility3 uses Intermediate Symbol Format (ISF) files which are standard JSON. The V8's memory structure could be written to an ISF to provide symbol tables that allow for easier manageability. Additionally, the V8MapScan will become better suited towards any future versions of V8, as ISF files can be provided to accommodate versioning. ISF will enable parsing of V8 objects to become more manageable. Currently, V8 object fields are parsed in Python class members. These Python class members can also be error-prone and difficult to understand.

Acknowledgements

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. Reverse engineering of data structures was provided by Mathew Piscitelli, documentation of structures, and assistance of developing the volatility plugin was greatly appreciated. Additionally, the assistance provided by Hailey Johnson, who helped manage the project, testing the plugin, and keeping track of the team's progress was critical to the project's success.

References

Ahn, Wonsun, Choi, Jiho, Shull, Thomas, María, J., 2014. Garzarán, and Josep Torrellas. Improving javascript performance by deconstructing the type system. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14. Association for Computing Machinery, New York, NY, USA, ISBN 9781450327848, pp. 496–507. https://doi.org/10.1145/2594291.2594332, 10.1145/2594291.2594332.

Arthur Lattner, Chris, 2002. LLVM: an Infrastructure for Multi-Stage Optimization. University of Illinois at Urbana-Champaign. PhD thesis.

Bak, Lars. The official mirror of the v8 git repository. https://github.com/v8/v8.

Bhattacharya, Dev, Kent, Kenneth B., Aubanel, Eric, Daniel, Heidinga, Shipton, Peter, Micic, Aleksandar, 2017. Improving the performance of jvm startup using the shared class cache. In: 2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM). IEEE, pp. 1–6.

Bikineev, Anton, Katz, Omer, Lippautz, Michael, Nov 2021. Oilpan library. https://v8.dev/blog/oilpan-library.

Boehm, Hans-J, Demers, Alan J., Scott, Shenker, 1991. Mostly parallel garbage collection. ACM SIGPLAN Not. 26 (6), 157–164.

Cano, Nick, 2016. Game Hacking: Developing Autonomous Bots for Online Games. No Starch Press.

Case, Andrew, Richard III, Golden G., 2017. Memory forensics: the path forward. Digit. Invest. 20, 23–33.

Case, Andrew, Jalalzai, Mohammad M., Firoz-Ul-Amin, Md, Maggio, Ryan D., Ali-Gombe, Aisha, Sun, Mingxuan, Richard III, Golden G., 2019. Hooktracer: a system for automated and accessible api hooks analysis. Digit. Invest. 29, S104–S112.

Casey, Peter, Lindsay-Decusati, Rebecca, Baggili, Ibrahim, Frank, Breitinger, 2019. Inception: virtual space in memory space in real space—memory forensics of immersive virtual reality with the htc vive. Digit. Invest. 29, S13–S21.

Chambers, Craig, Ungar, David, Lee, Elgin, 1989. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. ACM Sigplan Not. 24 (10), 49–70.
Concurrent marking in v8. https://v8.dev/blog/concurrent-marking, June 2018.
Degenbaev, Ulan, Eisinger, Jochen, Hara, Kentaro, Hlopko, Marcel, Lippautz, Michael, Payer, Hannes, 2018. Cross-component garbage collection. In: Proceedings of the ACM on Programming Languages, 2(OOPSLA), pp. 1–24.
Feng, Wu-chang, Kaiser, Ed, Schluessler, Travis, 2008. Stealth measurements for cheat detection in on-line games. In: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games, pp. 15–20.
Flood, Christine H., Detlefs, David, Shavit, Nir, Zhang, Xiolan, 2001. Parallel garbage collection for shared memory multiprocessors. In: Java Virtual Machine Research and Technology Symposium.
Frank, Block, Dewald, Andreas, 2017. Linux memory forensics: dissecting the user space process heap. Digit. Invest. 22, S66–S75. https://doi.org/10.1016/j.diin.2017.06.002. ISSN 1742-2876.
Glazunov, Sergei, 2021. Cve-2021-30551: chrome type confusion in v8. https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2021/CVE-2021-30551.html.
Google chrome statistics for 2022, Mar 2021. https://backlinko.com/chrome-users.
Graziano, Mariano, Lanzi, Andrea, Balzarotti, Davide, 2013. Hypervisor memory forensics. In: Stolfo, Salvatore J., Angelos Stavrou, Wright, Charles V. (Eds.), Research in Attacks, Intrusions, and Defenses. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 21–40.
Dominik Inführ. Generational and Parallel Garbage Collection.
Iqbal, Farkhund, Motyliński, Michał, MacDermott, Áine, 2021. Discord server forensics: analysis and extraction of digital evidence. In: 2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS). IEEE, pp. 1–8.
Jan, Rüth, Zimmermann, Torsten, Wolsing, Konrad, Oliver, Hohlfeld, 2018. Digging into browser-based crypto mining. In: Proceedings of the Internet Measurement Conference 2018, pp. 70–76.
Jargon, Julie, Jun 2019. The dark side of discord, your teen's favorite chat app. https://www.wsj.com/articles/discord-where-teens-rule-and-parents-fear-to-tread-11560245402.
Kim, Min-jeong, Ryou, Jae-cheol, 2019. Development of lldb module for potential vulnerability analysis in ios application. J. Internet Comput. Serv. 20 (4), 13–19.
Krylov, Georgiy, Patrou, Maria, Dueck, Gerhard W., Siu, Joran, 2020. The evolution of garbage collection in v8: google's javascript engine. In: 2020 9th Mediterranean Conference on Embedded Computing (MECO). IEEE, pp. 1–6.
Lattner, Chris, Adve, Vikram, 2004. Llvm: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, pp. 75–86.
Levanoni, Yossi, Petrank, Erez, 2001. An on-the-fly reference counting garbage collector for java. In: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 367–380.
Lewis, Nathan, Case, Andrew, Ali-Gombe, Aisha, Richard, Golden G., 2018. Memory forensics and the windows subsystem for linux. Digit. Invest. 26, S3–S11. https://doi.org/10.1016/j.diin.2018.04.018. ISSN 1742-2876. https://www.sciencedirect.com/science/article/pii/S1742287618301944.
Li, Haoyu, Wu, Mingyu, Chen, Haibo, 2018. Analysis and optimizations of java full garbage collection. In: Proceedings of the 9th Asia-Pacific Workshop on Systems, pp. 1–7.
Meadows, Catherine, 2003. A procedure for verifying security against type confusion attacks. In: 16th IEEE Computer Security Foundations Workshop. IEEE, pp. 62–72. Proceedings, 2003.
Mulazzani, Martin, Reschl, Philipp, Huber, Markus, Leithner, Manuel, Schrittwieser, Sebastian, Weippl, Edgar, Wien, F.C., 2013. Fast and reliable browser identification with javascript engine fingerprinting. In: Web 2.0 Workshop on Security and Privacy (W2SP), vol. 5. Citeseer.
Noordhuis, Ben. bnoordhuis/node-heapdump: make a dump of the v8 heap for later inspection. https://github.com/bnoordhuis/node-heapdump.
Ooi, Joo Guan, Kam, Kok Horng, 2009. A proof of concept on defending cold boot attack. In: 2009 1st Asia Symposium on Quality Electronic Design. IEEE, pp. 330–335.
Oracle. 3 generations. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html, Feb 2015.
Pagani, Fabio, Balzarotti, Davide, August 2019. Back to the whiteboard: a principled approach for the assessment and design of memory forensic techniques. In: 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA, ISBN 978-1-939133-06-9, pp. 1751–1768. https://www.usenix.org/conference/usenixsecurity19/presentation/pagani.
Papadopoulos, Panagiotis, Ilia, Panagiotis, Polychronakis, Michalis, Markatos, Evangelos P., Ioannidis, Sotiris, Vasiliadis, Giorgos, 2018. Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation. *arXiv preprint arXiv:1810.00464*.
Ren, Xin, Ying, Zhangxu, 2016. Generational garbage collection algorithm based on lifespan prediction. In: 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), pp. 183–187. https://doi.org/10.1109/W-FiCloud.2016.47.
Ronitsinha. Ronitsinha/node-js-game: a node.js agar.io clone. https://github.com/ronitsinha/node-js-game.
Schatz, Bradley, Cohen, Michael, 2017. Advances in volatile memory forensics. Digit. Invest. 20. https://doi.org/10.1016/j.diin.2017.02.008, 03.
Tiwari, Devesh, Yan, Solihin, 2012. Architectural characterization and similarity analysis of sunspider and google's v8 javascript benchmarks. In: 2012 IEEE International Symposium on Performance Analysis of Systems & Software. IEEE, pp. 221–232.
Tyler, Thomas, Piscitelli, Mathew, Shavrov, Ilya, Baggili, Ibrahim, 2020. Memory foreshadow: memory forensics of hardware cryptocurrency wallets – a tool and visualization framework. Forensic Sci. Int.: Digit. Invest. 33, 301002. https://doi.org/10.1016/j.fsidi.2020.301002. ISSN 2666-2817. https://www.sciencedirect.com/science/article/pii/S2666281720302511.
Tyler, Thomas, Piscitelli, Mathew, Ashok Nahar, Bhavik, Baggili, Ibrahim, 2021. Duck hunt: memory forensics of usb attack platforms. Forensic Sci. Int.: Digit. Invest. 37, 301190. https://doi.org/10.1016/j.fsidi.2021.301190. ISSN 2666-2817. https://www.sciencedirect.com/science/article/pii/S2666281721000986.
Van Der Horst, Luuc, Raymond Choo, Kim-Kwang, Le-Khac, Nhien-An, 2017. Process memory investigation of the bitcoin clients electrum and bitcoin core. IEEE Access 5, 22385–22398. https://doi.org/10.1109/ACCESS.2017.2759766.
Weninger, Markus, Gander, Elias, Mössenböck, Hanspeter, 2018. Utilizing object reference graphs and garbage collection roots to detect memory leaks in offline memory monitoring. In: Proceedings of the 15th International Conference on Managed Languages & Runtimes, pp. 1–13.