Digital Investigation 29 (2019) S13–S21

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2019 USA — Proceedings of the Nineteenth Annual DFRWS USA

Inception: Virtual Space in Memory Space in Real Space — Memory Forensics of Immersive Virtual Reality with the HTC Vive

Peter Casey*, Rebecca Lindsay-Decusati, Ibrahim Baggili, Frank Breitinger

University of New Haven, 300 Boston Post Rd, West Haven, CT, 06516, USA

ARTICLE INFO

Article history:

Keywords:
Memory forensics
Data recovery
Virtual reality
Reverse engineering

ABSTRACT

Virtual Reality (VR) has become a reality. With the technology's increased use cases, comes its misuse. Malware affecting the Virtual Environment (VE) may prevent an investigator from ascertaining virtual information from a physical scene, or from traditional "dead" analysis. Following the trend of anti-forensics, evidence of an attack may only be found in memory, along with many other volatile data points. Our work provides the primary account for the memory forensics of Immersive VR systems, and in specific the HTC Vive. Our approach is capable of reconstituting artifacts from memory that are relevant to the VE, and is also capable of reconstructing a visualization of the room setup a VR player was immersed into. In specific, we demonstrate that the VE, location, state and class of VR devices can be extracted from memory. Our work resulted in the first open source VR memory forensics plugin for the Volatility Framework. We discuss our findings, and our replicable approach that may be used in future memory forensics research.

1. Introduction

With the rise in popularity of Virtual Reality (VR) devices, and their new found residence in the consumer marketplace, these complex systems will soon become an important source of digital evidence. Hardware improvements, price reductions and a boom of content has led to over 1 million VR headsets being sold in 2017 (Lamkin, 2017). As end users have readily adopted Augmented Reality (AR), VR, and Mixed Reality (MR) alike, the forensics community has lagged in preparing for a new environment, where crimes and evidence are a coalescence of virtual and physical realities.

Virtual Environments (VEs), hosting new types of social interaction, may also be the locale for misconduct. Reports of sexual harassment in VR, where women have been groped have caught both digital investigators and lawmakers off guard (Wong, 2016). Fully immersive systems, which envelope the entirety of a user's vision, force the user to impart their trust to the VR system's safety mechanisms. Users may injure themselves using the systems, or defraud each other in the VEs. Moreover, social features of online gaming have recently become a haven for money laundering (Editor, 2015). Ethical and legal controversy have kept a legal precedence at bay, however, these concerns should not preclude a thorough forensic evaluation (Lemley and Volokh, 2017).

Preliminary work has been conducted on disk and network artifacts of VR systems, however, volatile memory is a source yet to be considered (Yarramreddy et al., 2018). Large amounts of information, that may not be available in traditional storage can be recovered from volatile memory. This may include running processes, network connections, chat messages and encryption keys (Case and Richard, 2017). VR systems are no exception; the complex tracking system and device events are largely handled in memory. Traditional "dead" analysis will not recover such evidentiary data.

As forensic techniques continue to improve and adversary tactics evolve, memory forensics has become increasingly important. Malware and exploitation tools are progressively reducing their non-volatile footprint and thus evidence collection must keep pace. As the field was once primarily focused on operating system level forensics (Case and Richard, 2017), application and device specific data collection is too finding relevance (Anglano et al., 2017; Sylve et al., 2012). Today, our homes have been filled with an ecology of devices. We have seen the epic rise of smart phones, eReaders, smart watches, fitness and health trackers, gaming devices, smart TVs, Internet of Things devices, etc. Virtual Reality devices are now moving into this complex playing field and require a forensic evaluation of volatile memory.

With MR systems blending together virtual and physical experiences, malware targeting these devices may have a slew of

* Corresponding author.
E-mail addresses: pgrom1@unh.newhaven.edu (P. Casey), rlind2@unh.newhaven.edu (R. Lindsay-Decusati), IBaggili@newhaven.edu (I. Baggili), Fbreitinger@newhaven.edu (F. Breitinger).
URL: http://www.unhcfreg.com, http://www.unhcfreg.com, http://www.Baggili.com/, http://www.fbreitinger.de

https://doi.org/10.1016/j.diin.2019.04.007

additional capabilities. An immersed user, who trusts both their vision and hearing to the VR device, is a perfect target for an attack that could even bring about physical harm (Casey et al., 2019; UNHcFREG, 2018). Should the VE be maliciously modified to disrupt or deceive an immersed user, the calibration between the physical and virtual space will inevitably be compromised. Thus, an onlooker cannot reliably ascertain the user's virtual orientation from their physical. *This discrepancy would only be found in the live copy of the system's configuration, residing only in memory.*

The HTC Vive, which provides a fully immersive VR experience and utilizes the most popular gaming platform, Steam, is a fitting candidate for study (Bailey, 2018). In this work, we conduct an analysis of the runtime of the HTC Vive to identify potential sources of digital evidence unique to volatile memory. We develop a methodology to locate, and extract these data structures from a memory dump. Finally, we automate the process and form a 3-dimensional reconstruction of the VE via a plugin for the Volatility Framework. Our work provides the following contributions:

1. To the best of our knowledge this is the primary account for specifically examining the memory forensics of VR systems.
2. We share our analysis and findings that may impact future investigations involving VR systems.
3. We employ and share a reusable methodology that may be adopted by others to create similar plugins.
4. We construct an open source tool *Vivedump*, that may be used in the analysis of memory dumps of HTC Vive VR systems and share related datasets. The tool is a plug-in for the widely adopted Volatility framework.

The remainder of this paper is organized as follows. We familiarize the reader with VR related components and discuss related work in Section 2. We present the details of our apparatus and applications used in Section 3. In Section 4, we present our methodology from scenario creation through plugin development. Our findings are presented and discussed in Sections 5 and 6 respectively. Finally, we identify areas of future work (Section 7) and make concluding remarks (Section 8).

2. Background information and related work

VR has been the subject of research in many different academic disciplines for a very long time. There is much research on what technology would be required for such a system, and on potential future uses for this technology. Only recently have consumer grade VR devices become available. State-of-the art work has only scratched the surface on the forensics and security of VR. de Guzman et al. (2018) provided a survey of privacy and security research and approaches in MR. Most relevant to our work is the work by Yarramreddy et al. (2018) which considered disk and network artifacts of social VR applications, while Casey et al. (2019) evaluated the security of these systems and implemented proof-of-concept attacks.

Currently there are three types of consumer VR systems available. Firstly, systems that operate through a small scale device, typically a smart phone with a peripheral such as Google Cardboard or the Samsung Gear VR. Secondly, standalone systems that do not require additional hardware, such as the Oculus Go and Google Daydream. Finally, systems driven by a desktop or laptop computer, which allow for a more robust experience. These systems are typically fully immersive and allow room-scale play, requiring a tracking system. One such system is the HTC Vive and is the focus of our work. Readers familiar with this system, may want to skip the next two paragraphs, but we chose to describe the details of the HTC Vive system, and it's Application Programming Interface (API) below.

The HTC Vive base configuration consists of a headset or Head Mounted Display (HMD) which is tethered to a computer, two wireless handheld controllers, and two wall mounted base stations that are responsible for tracking the other devices. The base stations employ Valve Corporation's lighthouse technology to sweep the room with infrared light and triangulate the tracked devices. The final computed location in virtual space is referred to as a *pose*. Aside from HDMI image processing on the HMD, the computer driving the system handles the bulk of the workload, allowing for convenient memory collection and analysis. Applications are typically launched through Steam, a digital distribution platform developed by Valve, with SteamVR responsible for the VR runtime. To promote third-party application development, Valve established a framework that can be expanded to suit any VR system known as OpenVR.

OpenVR is an API designed to free developers from relying on hardware specific Software Development Kits (SDKs). Simply a collection of virtual functions matched to the appropriate VR interface, OpenVR allows applications to maintain compatibility among several systems and SDKs (Valve Software, 2018). Although OpenVR can power a variety of systems, OpenVR's developer Valve partnered with the development of the HTC Vive. Utilizing the API, we can access hardware and tracking information otherwise unavailable to the user.

One such component of the VE is the Chaperone. Because the system is fully immersive and encapsulates the entirety of the user's vision, there must be some protection from physical obstacles. When in close proximity to the user defined boundaries, a virtual wall (Chaperone) will appear and notify the user.

2.1. Memory forensics

Formal memory forensics arrived on the scene as early as 2001, with the first structured analysis tools surfacing in 2004 (Case and Richard, 2017), and a wave of work in 2005 and 2006 with the Digital Forensics Research Workshop (DFRWS) Forensics Challenges from those years that spurred several analysis tools such as the Volatility Framework (DFRWS, 2006a, b; Volatility Foundation, 2018).

Factors such as disk encryption and the rise of memory resident malware have contributed to the need for memory forensics tools. Initially, memory forensics focused on malware analysis, until the benefits to other types of investigations were realized. Much work has subsequently been conducted in the areas of memory acquisition, and memory analysis. Gruhn and Freiling (2016) provided an evaluation of methods of acquisition and analysis, comparing the correctness, atomicity and integrity of each. Work focusing on acquisition includes Petroni et al. (2006) with the introduction of FATKit, acquisition of persistent systems (Huebner et al., 2007), and the use of rootkits to limit acquisition (Bilby, 2006).

A presentation on the subversion of memory forensic acquisition, the work by Schatz (2007) suggests a minimal operating system injected into a host system to acquire memory dumps without the need to be concerned with the problems of trust where a rootkit may be present in the host operating system. Balogh (2014) demonstrated the use of a network interface controller to use direct memory access for memory acquisition. Memory forensics has changed the traditional digital paradigm such that modus operandi of digital forensics has changed: One can no longer simply pull the plug immediately. Memory forensics seems to add complexity, but it is an approach needed to deal with an increasingly complex digital world.

Analyzing a memory dump can be as equally complex, as volatile memory is transient, with each component or application having its own address space. In addition, paging and redundancy can complicate the availability of information. A great deal of work

Table 1
Workstation details.

System Details		Application Details	
Device	Details	Application Name	Version
Processor	Intel Core i7-6700 CPU	Cheat Engine	6.7
System Type:	64-bit OS, x64 processor	DumpIt	3.0.20170909.1
Graphics Card	NVDIA GeForce GTx 1070	Microsoft Visual Studio	10.0.16299.0
Manufacturer	iBUYPOWER	OpenVR	v1.0.13
Installed Memory (RAM)	8.00 GB	SteamVR	1523310137
		Volatility	2.6

has been conducted on the forensics of operating systems. Dolan-Gavitt (2008) demonstrated analysis of the registry in main memory, and van Baar et al. (2008) targeted files mapped in memory. Zhang et al. (2009) showed Windows kernel processor control region based analysis and the work by Maartmann-Moe et al. (2009) demonstrate the use of memory forensics to recover cryptographic keys. Hejazi et al. (2009) provided further analysis of main memory in Windows through work with security specific calls and the call stack. The technique allowed for data recovered with previous string search methods. Data structures used by DOSKEY were examined by Stevens and Casey (2010), which allowed the command line history to be discovered in memory.

Data extraction, being an important goal of memory analysis can be heavily reliant on reverse engineering in the absence of source code and further complicated by the diversity of applications and their frequent updates. Huebner et al. (2007) presented a method of creating profiles from C source code to automate this process. Manual searches can be time consuming and lag behind the most current operating system version, yet several tools benefit from this (Betz, 2005; Jr. and Mora, 2005). With YARA, a useful pattern matching tool for malware detection, textual and binary patterns can be quickly identified (Alvarex, 2018). Cohen (2017) showed how YARA can be applied to memory to maintain context between Virtual Address Descriptors (VADs). Our work employs YARA scanning across a process's virtual address space, however, signatures do not span multiple VADs. At the time of writing, there was no literature addressing the memory forensics of VR systems, and as such, literature in that domain is lacking.

3. Apparatus

In this section, we describe the devices and software used in our methodology. A desktop workstation was used to drive the VR system, the details of which can be found in Table 1. Post-collection memory analysis was conducted using Volatility and the resulting plugin was designed to function within the Volatility Framework. Because the Windows profiles necessary for memory analysis were not included in the latest release (at the time of writing), namely, Windows 10 × 64 Build 16299, the most up-to-date profiles were obtained directly from the source code.[1] The VR system used in this study was an HTC Vive with the standard components. Details of the Vive and individual devices are presented in Table 2.

4. Methodology

This section describes the process used to identify memory locations associated with our targeted data structure and the means of reliable access. A high level overview of the methodology is presented in Fig. 1. Creating a scenario with predictable values, we conduct memory analysis to locate potential data containing memory regions. Pointer scans are conducted to identify routes to the data and the machine instructions are inspected to ensure static references. Using the machine instructions as a YARA signature and the aforementioned routes, we constructed a Volatility plugin to automatically extract the data from a memory dump.

[1] https://github.com/volatilityfoundation/volatility (last accessed 2018-10-25).

Table 2
HTC Vive details.

Device	Device ID	Firmware	Hardware
Headset	LHR-941D5CC1	1462663157	0x80020100
Basestation	LHB-BD6F4BB6	436	0x09000009
Basestation	LHB-016BDC7A	436	0x09000009
Vive Controller	LHR-FDDA1B43	1465809478	0x81010500
Vive Controller	LHR-FFE6DB47	1465809478	0x81010500

4.1. Documentation analysis

We began our study with an analysis of the OpenVR API. The API documentation was surveyed to identify data structures found to contain information that may be of interest to a forensic investigator or hold Sensitive Personal Information (SPI). Table 3 shows the data structures of interest for this study, some of which are tracked device pose information, device class, device state/activity, hardware and firmware versions. The objective of this portion of the study was to determine the extent to which information is readily available in memory and understand the layout and data type of the structures we aim to extract. It should be noted that data structures found in the OpenVR documentation may not perfectly translate to the underlying VR runtime. Both SteamVR and OpenVR are authored by Valve Corporation, thus we find many similarities.

We will not present the details of each data structure we investigated, rather, we will use a single example to trace our methodology. Listing 1 shows the OpenVR header declaration of TrackedDevicePose_t. Not depicted, each component of the HMD matrices or vectors are float values and ETrackingResult is an enumerator. With this in mind, while conducting our memory analysis we can expect the structure to be laid out as follows: $(3{*}4)float_{HmdMatrix34\_t} + 3{*}float_{vVelocity} + 3{*}float_{vAngularVelocity} + int_{eTrackingResult} + 2{*}bool$. The C++ standard states that the complier cannot reorder struct members, thus we can also expect to find the data in the order shown in Listing 1 (International Organization for Standardization, 2017, 12.2.17).

Listing 1 : TrackedDevice Pose_t

```
1  struct TrackedDevicePose_t
2  {
3      HmdMatrix34_t mDeviceToAbsoluteTracking;
4      HmdVector3_t vVelocity;
5      HmdVector3_t vAngularVelocity;
6      ETrackingResult eTrackingResult;
7      bool bPoseIsValid;
8      bool bDeviceIsConnected;
9  };
```

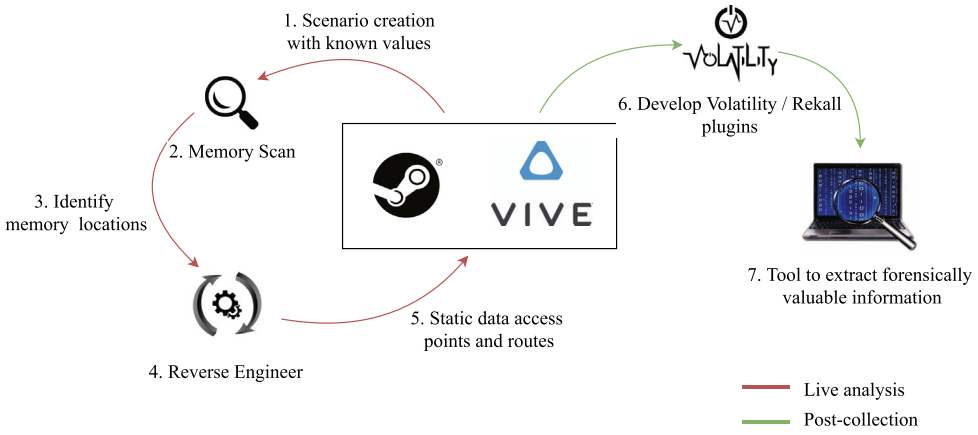

Fig. 1. Methodology.

Table 3
OpenVR data structures and enumerators.

Data	Contents
TrackedDevicePose_t	Pose and status of tracked device
HmdMatrix34_t	Tracked device tranformation matrix
ETrackedDeviceClass	Type of Device
ETrackedDeviceProperty	Static device properties
EVRState	Status of the overall system
EVREventType	Event types
EDeviceActivityLevel	Level of Hmd activity
VREvent_Notification_t	Notification related events
VREvent_Overlay_t	Overlay Events
VREvent_Ipd_t	Ipd change

4.2. Scenario creation

In order to search for the targeted data in memory, we must be able to either manipulate the variable to a known value or acquire the value directly from the VR runtime. We can control and record the state of the VR system as it is in physical space, yet much of the underlying tracking and hardware information is transparent to the user. Leveraging OpenVR, we developped a minimal helper program to output each of the components of TrackedDevicePose_t to the console. We then manipulated the physical position of the HMD and observed the changes to the data structure. In doing so we verified our observed output, but also establish known location in physical space. However, the helper program's interaction with the VR runtime would not be typical of a normal user experience and would thus taint future memory collection. For our post-collection analysis, we must be able to verify the correctness of our location without the aid of our helper program. Thankfully, the VE and pose units are in meters, allowing a measurable conversion from physical location to virtual pose location.

Observing the values of each component of TrackedDevicePose_t under device manipulation, we find mDeviceToAbsoluteTracking to be suitable for memory searching. Representing the 3-dimensional transformation, we can expect the value to be unique and relatively constant. The most intuitive and verifiable data point to search for in this structure is the *x*, *y*, and *z* positional coordinates of the tracked device, found in `mDeviceToAbsoluteTracking [0][3], [1][3], [2][3]` respectively. Both velocity data vectors are not practical because of the difficulty associated with maintaining a constant value. Furthermore, eTrackingResult and the boolean values may not be distinct enough to narrow memory search results.

4.3. Memory scanning

Once the pose coordinate information is readily available via our helper program, we can conduct a memory search to locate the data. We elected to use Cheat Engine to locate and identify pointers to the data (Byte, 2018). Targeting the VR runtime, we limit our search to memory allocated to the processes child to SteamVR. Future work may expand our search to specific user VR applications. As we might expect, the VR tracking system will constantly be updating the location, however, Cheat Engine allows for rounded float searching, providing the necessary tolerances for the data to be found. Filtering results for memory regions that are rapidly updating, we were able to identify a single range of addresses aligned with the expected size of the structure.

Manual inspection of the address and the following values adhered to what would be expected. To eliminate false positives and directly correlate memory location to purpose, we manipulated the physical location of the HMD while continuing to monitor the candidate memory region. If the changes to the physical system were synchronous with the values in memory and values corresponded to that of the helper program, we were certain the candidate memory region was representative of the targeted data structure.

4.4. Working backwards to a static reference

Over multiple collections, data was found to be dynamically stored, therefore, to reliably locate in the absence of scenario information we must tie the address to a static location. By conducting a pointer scan in Cheat Engine we worked our way backwards to a static reference in the execution block. The VR runtime includes several cooperative processes, however, most information regarding the hardware was found to originate from *vrmonitor. exe*, thus, we selectively filtered for these base addresses and routes. In doing so, we ensured that the data was locatable regardless of its placement in memory.

Sorting the results of the pointer scan by number of hops, we found the shortest path to the data block:

```
Base: “vrmonitor.exe"+00206950 Offset: 68
```

The pointer simply originating from the executable as opposed to the Thread Stack or a Dynamically Linked Library (DLL) does not alone indicate that base address will be reliable. Cheat Engine indicates if an address is located in the *DATA* portion of the executable, and thus “hard coded”. We can further strengthen our ability

to locate the base address of our route, which will in turn aid in plugin development by identifying the machine instructions which assess our base address. This was achieved by attaching a debugger to the process *vrmonitor. exe*, and observing accesses and writes to the base address. It should be noted that we observed the frequency of accesses to correlate to the frequency of tracking pose updates.

4.5. YARA signatures and memory collection

We can then use accessing or writing instructions as a signature to locate our base address. A disassembly of this portion of memory yielded is shown in Listing 2. Line 1 being the location at which the pointer to the data structure is accessed. Because the pointer is accessed using an offset from the instruction pointing register, we confirm this will be a reliable point of access.

Listing 2 : Disassembly of Base Address Access

```
1  48 8b 05 25 d6 10 00
   MOV RAX, [RIP+0x10d625]
2  48 85 c0                TEST RAX, RAX
3  75 2a                   JNZ + 2a
4  snip
```

Memory was then collected using the tool Dumpit. Information regarding the state of the system and location of each device was recorded at the time of collection. The helper program, Cheat Engine and the debugger would not be typical of a normal user experience, thus, prior to collection the system is restarted. Therefore, we cannot use the VR tracking system to record the device location, however, as previously discussed, known virtual locations and their coordinates have been established.

Conducting a YARA scan on a memory dump for the machine instructions is shown in Listing 2, Line 1 we can ensure that the signature is unique, and will not produce a false positive for the base address. Additionally this provided us with the physical address of our signature in the memory dump. Volatility's interactive memory image explorer, *volshell*, allowed us to manually trace the path to the data block, and validate its contents. Dereferencing the contents of `RIP+0x10d625` and applying the offset 0x68, we observed an array of float values. These were confirmed, over multiple memory collections, to accurately represent the location of the HMD.

4.6. Plugin development

The Volatility plugin *Vivedump*, automates the process of locating the base address and dereferencing the chain of pointers. Pseudo-code for the general process of locating the data is presented in Algorithm 1. The example used to trace the methodology is blunt, requiring only a single dereference, however, other data structures may require several steps. Enumerating Python lists allows for easy handling of variable length routes to the data (Lines 2, 7).

Considering only the process space for *vrmonitor* has its benefits (Line 4), being limited signature collisions and reduced computing time. This is particularly favorable as VR capable machines typically have a large amount of Random-access Memory (RAM). We integrated command line options to allow the search space to be expanded, as we anticipate evidence will originate from other Steam processes or third party applications.

Enumerating the VAD tree for the process, we conduct a YARA scan for the signatures determined in Section 4.5 (Lines 4–5). Finally, the pointer is dereferenced and the offset applied in accordance with the specified route *r* (Lines 7–8). The result is the address of the targeted data structure. Still the plugin's output can be improved by handling data interpretation as well.

Algorithm 1 Vivedump: Locate target data

```
1:  y ← yara.compile(rule)            ▷ Base address signature
2:  r ← [offset1, offset2, ...]                          ▷ Route
3:  for task ∈ processList do
4:      if task = vrmonitor.exe then
5:          for vad ∈ getProcessVad do
6:              m ← y.match(vad)          ▷ Address of YARA match
7:              for offset ∈ r do                  ▷ Traverse route
8:                  m ← dereference(m) + offset
9:              end for
10:             return m                  ▷ Address of target data
11:         end for
12:     end if
13: end for
```

4.6.1. Reconstruction and interpretation

Remembering that the first location coordinates we searched for was located at *HmdMatrix*34_*t*[0][3], the base of the data structure must be 3∗*sizeof*(*Float*) prior. We can then proceed to process 12 consecutive 32 bit values as type float, into our reconstructed mDeviceToAbsoluteTracking matrix. Following the structure described in Listing 1, the succeeding group of three floats are assigned to vVelocity and vAngularVelocity.

We are fortunate that the enumerator declaration is provided in the OpenVR documentation. We can then reiterate the enumerator as a Python dictionary to aid in interpretation. The ETrackingResult was observed to be a 32-bit unsigned integer and read accordingly. Finally the remaining boolean values occupy the upper and lower bytes of the concluding DWORD. A bit mask was applied to isolate each value.

5. Findings

5.1. Data availability

Repeating the methodology for items in Table 3, we are able to find routes to access many of the data structures. A summary of our findings and extractable evidentiary data by our plugin is presented in Table 4. Again, with enumerated type data such as tracked device state, activity and class we are able to use the OpenVR documentation to ease our interpretation of the data.

Our focus was directed towards acquiring information that would allow for the reconstruction of the VE and have every reason to believe that with a thorough investigation all desired data could be obtained from memory. Elements of Table 4 are complementary in painting a picture of the VR system at the time of memory collection. Utilizing the information given from the tracked device state, we can detect the number of devices present, while the activity level allows us to determine which tracked devices were recently moved.

Most VR applications, aside from particular utility type OpenVR applications, require an HMD to be present. By default, the HMD will always occupy the first position of the TrackedDevicePose array, however, we did not observe a trend in array position for other devices, namely the controllers and base stations. There is

Table 4
Summary of routes to data structures.

Data Structure	YARA Signature	Route
TrackedDevicePose Array	48 8b 05 25 D6 10 00	0x10D62C
HmdMatrix34_t		+0 × 5C
ETrackingResult		+0xA4
bPoseIsValid		+0xA8
bDeviceIsConnected		+0xAA
HMD State	48 8b 05 55 A4 13 00	0x13A45C, 0x1E0, 0 × 0B8, 0x68
HMD Activity		+0x10
TrackedDevice Class	48 8b 05 55 A4 13 00	0x13A45C, 0x190, 0x8, 0x510, 0x3E0, 0x120

likely a race condition at startup during the initialization of these Bluetooth devices. For this reason we must be able to detect the type of device in the latter pose array positions. Generally, the base stations can be identified from their location, as they are typically wide spread, elevated and at the outskirts of the room. Nonetheless, we can definitively determine the device class using the described route.

The methodology could repeated for the controllers and base stations, however for most data structures this is not necessary. Often the structures are allocated in an array, meaning following the corresponding HMD data we can process each subsequent device. For this reason we only include routes for the HMD in Table 4.

5.2. The chaperone

The data structure containing the Chaperone information was easily found in memory, yet no path from the VR runtime executable was identified. This is likely due to the Chaperone being immediately loaded on start-up, preventing the debugger in our analysis from tracing the access of this information. Other applications accessing the Chaperone do so through the *vrclient_x64* DLL. Utilizing the application specific routes we can extract the data, however anticipating the specific application is not practical nor necessary. Rather, this information is stored to the disk in json format (chaperone_info.vrchap) and often remains in memory. The persistence of the file particularly holds true should the play area configuration be modified at any point during the session. Because this information is available both on disk and regularly in memory we did not deem it necessary to further pursue extraction. The plugin will conduct a file scan for the json in memory and if not found the file can be manually provided to the visualizer.

Akin to the Chaperone, several elements of Table 3, such as device hardware properties and Interpupillary Distance (IPD) are generally static or infrequently changed. This increases the difficulty associated with identifying routes and signatures, should the data even be present. Often these properties did not appear in memory searches unless called upon by a user VR application. Be that as it may, this information is not unique to volatile memory and can be easily obtained from the disk. For this reason we do not consider this information as a worthwhile endeavor and if required handled on a case by case basis.

5.3. Vivedump: usage

The plugin, utilizing the Volatility Framework, requires the memory dump, operating system profile, and the plugin directory as input. Command line options allow for the user to specify the number of tracked device poses to extract, expand the breadth of search, and manually provide the Chaperone file. The plugin will output a textual representation of all evidentiary data, a wavefront *obj* formatted mesh of the VE, and pass the 3-dimensional coordinates to a Python OpenGL instance. Sample textual output is presented in Fig. 2 and source code can be found at https://github.com/unhcfreg/VR4Sec.

5.4. Vivedump: visualization

We cannot expect a forensic examiner to easily interpret the data as found in memory, therefore it is appropriate to present the VE in an intuitive format. The pose matrix represents the devices rotation and translation with respect to the center of the tracked area. We can then transpose the matrix from its 3*x*4 arrangement to the standard OpenGL 4*x*4 transformation matrix and present the user with a 3-dimensional representation of the room. The tracked device class can then be used to apply an appropriate mesh to the device for visualization. If acquired, the Chaperone will be added to the resulting visualization. This provides context to the tracked device locations in the physical room, as the Chaperone boundaries typically outline physical walls. An example of the visualization produced is provided in Fig. 3.

6. Discussion

The data structures that were acquired in this study were all

```
python vol.py --plugin=vive-dump -f mem_dump.dmp --profile Win10x64_16299 vivedump
Volatility Foundation Volatility Framework 2.6

HMD
-0.988565862179,   -0.0586262568831, -0.138926059008, 0.323732972145
-0.0173488464206,  0.959426701069,   -0.281423956156, 1.71743369102
0.149788171053,    -0.275795906782,  -0.949473559856, 0.0478134155273
ETrackingResult: TrackingResult_Running_OK
Bool values: 0x344e0101
bPoseIsValid: True
bDeviceIsConnected: True

HMD activity: k_EDeviceActivityLevel_UserInteraction
HMD state: VRState_Ready
```

Fig. 2. Sample output of HMD data extracted from memory using Vivedump. Additional tracked devices can be extracted using command line options (-N).

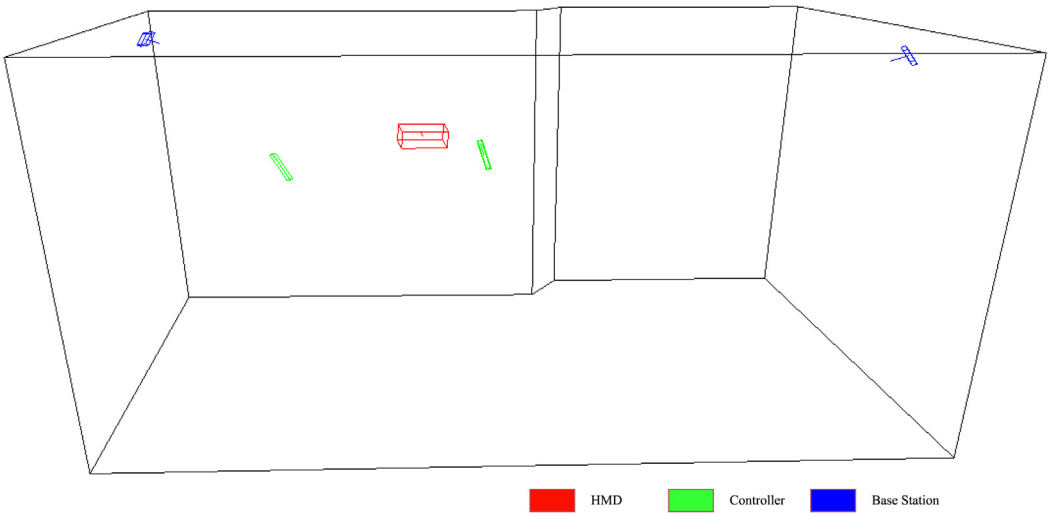

Fig. 3. Sample output of the visualization provided by Vivedump.

similar in that the underlying information could be easily manipulated. This played an important role in locating the proper memory region. Because many enumerators were used to describe device states and events and several processes are storing identical information, determining the nature of the data is impossible without some feedback. For instance, a search for a byte containing the value corresponding to a active controller state would yield an enormous amount of potential memory regions. Although, allowing the controller to enter an idle state eliminates all regions that do not respond to the change.

Furthermore, information that is not persistent is equally difficult to locate. In particular, an event object will occupy a new location with every event and unprocessed events will form a queue. This too precludes direct manipulation of the underlying memory region and drastically complicates the method to locate the information. During our analysis we did observe candidate regions, seemingly in a sequential log format, but were not able to observe any correlation to purposefully triggered events. Unfortunately, processed events and retired poses were not maintained in memory thereby incurring a low probability of recovery.

Much like event objects, the Chaperone could be found in memory but no obvious path to the data was identified. Data structures that are not frequently accessed or updated pose another challenge when searching for a route to the memory region. Our memory scanning tool can easily identify pointers, but to reliably extract the data from memory the base address of the route must also be tied to a static location. This was achieved if a pointer scan yielded a route from the executable, where if not directly referenced the pointer could be traced back to a offset from the instruction pointer. A feature of Cheat Engine, breaking the program when the target memory region is access, allowed us to quickly identify the necessary op code signatures. In contrast, the use of DLLs complicate the route as the base pointers are passed to the DLL functions, meaning the state of the registers must be further reversed.

This does not entirely prevent this information from being obtained, however the incentive is not always worth the effort. As Steam frequently is pushing updates to their VR runtime, there is a significant chance that the new builds will render the current signatures useless. During the course of this study, we encountered three SteamVR updates that had such an effect. This is of course a challenge associated with application specific memory forensics. We will discuss several potential methodologies which may help to overcome this obstacle in Section 7. Unfortunately, many of the VR related data structures are better manually retrieved on a case by case basis.

6.1. Reliability

All data structures of which we identified routes from a static location to (Table 4) except for TrackedDevice Class were consistently obtainable. This was tested over multiple memory captures, with varying degrees of VR usage, connected devices, and applications. Accessing the TrackedDevice Class information was occasionally misleading or fragmented. Either the path would diverge from the targeted region or simply fail due to a null pointer. Although the data is persistent, the intermediate pointers may be dependent on the state of the runtime. A small program was implemented to frequently request tracked device classes which in turn improved the likelihood of the pointer route remaining intact. This suggests that the intermediate steps may be rerouted to point to other data structures.

One may expect that a valid device class would correlate with bPoseIsValid, bDeviceIsConnected, and HMDState, however, disconnecting the device does not affect the device class. Meaning, if a controller is in the fifth position of the pose array and disconnects, a query to that position will still return the corresponding controller enumerator. That is until another device fills that pose array position. Thankfully, we observed the byte following the device class in memory to also correlate to the device state. With regards to reliability, we maintain the preferred route to ascertain whether a device is connected is via the above data structures.

6.2. Version detection

Much like Volatility requires the profile be determined to properly interpret operating system level information, the application version holds great weight. As frequent updates are a significant challenge for application specific memory forensics, detecting and handling a broad history of application versions is also necessary (Case and Richard, 2017). Often, with each update to the VR runtime, a new set of signatures were required to access the data. Throughout this study, three significant updates caused changes in the opcodes that were used for our YARA scans.

A built in feature of Volatility, *verinfo*, pulls version information embedded in portable executable files (Volatility Foundation, 2018). In many cases this will suffice to determine the application

release. Unfortunately SteamVR developers do not update this information to match the current build. Memory dumps spanning multiple versions were compared, analyzing SteamVR and its dependents. No direct corollary was observed using the output of verinfo and the SteamVR version obtained directly from the software.

The signatures used for each SteamVR build that spanned this study did not produce any conflicts. No false positive YARA matches were produced, simply reducing in lost computation time. Allowing searching for all signatures may be practical as our catalog of SteamVR versions remains small, yet, with continued updates, conflicts and decreased efficiency may accrue. Following the model used by Volatility, we intend to remedy this problem with profiles determined by additional wider signatures tailored for each SteamVR release. Each profile specifying the routes and signatures needed to access each data structure.

7. Future work

Our analysis of the HTC Vive runtime likely only encompasses a small portion of the available data. Further work is needed to determine the full breadth of information that can be extracted. Furthermore, 32-bit applications and other VR systems or workstations have yet to be considered. Although the OpenVR documentation serves as a good reference for data structures and availability, many other sources of data may exist. Our methodology can be applied for additional structures found in the OpenVR documentation that are easily manipulated. Small adjustments can be made to expand our context to non-documented points and different VR systems.

This study was focused towards the VR runtime and global data, but did not consider the user application. The user VR application may inherit much of the same data and provide an additional source. Furthermore, application specific information may be of value to an investigator. A potentially worthwhile use case involves social VR applications, where user interactions and conversations may be retrievable.

7.1. Scalability and automation

The frequent SteamVR updates dictated our Volatility plugin be designed to be easily adapted. Although profiles and version detection too remain part of our future work, the plugin requires three essential inputs. The YARA signature, route and type or arrangement of the data structure. Should a particular case require a new data point, minimal changes to the plugin are needed. Conveniently, two of these inputs directly originate from our memory search tool, Cheat Engine. Regardless of the targeted application, our plugin could be adapted to accept the file output of Cheat Engines pointer scan. Quite often, there are many routes to the data structure returned by the scan. Automation of traversing these routes would strengthen the probability of reliably obtaining data. A heuristic approach to determining appropriate base addresses from the pointer scan result will drastically reduce the man hours and complexity of plugin development.

Future work should consider building a framework to potentially generate application specific Volatility plugins with the following compressed methodology:

- Identifies the targeted memory region
- Conduct pointer scan
- Pass pointer scan output to framework
- Framework outputs a fully functional Volatility plugin

Application specific memory forensics will continue to be a difficult problem to tackle given the sheer volume of application production. Automation of plugin development, although not as reliable as manual investigation, may alleviate much of the tedious labor.

7.2. Stepping through memory

Our ability to extract information relies on a static reference to a memory location. Which in turn points us to the desired data. Depending on the state of the application while conducting our memory searches, varying pointer scan results will be produced. Should no route originating from the executable be found, identifying a static reference may be difficult. In contrast, we observed a large selection of routes from the Thread Stack and DLLs. This is particularly relevant for information sparsely accessed after initialization, such as device class.

Creating small applications and stepping through them on a live system, reveal how simple it may be to leverage *vrclient_x64. dll* to pull information even from a memory dump. Initializing an OpenVR instance returns a pointer representing the VR system, which points to the base of the DLL. When utilizing a virtual OpenVR function, a *call* is made to the location of the system pointer with the offset corresponding to the function. This system pointer can be extracted in the same manner as all entries of Table 4 from an OpenVR instance, or rather ascertained using the Volatility plugin *dlllist*.

Rather than relying on the routes derived from this study, which intermediate pointers may be subject to reuse, the DLL contains the instructions and pointers to access the data. We suspect that by emulating the system from which the memory dump was obtained, we can call these DLL functions. Preliminary observations have led us to believe that *vrclient_x64. dll* does not directly probe the VR system, rather access the location of where the data is already stored. Because of this, we expect these functions will continue to successfully execute in absence of the VR system. This proposed strategy may be useful in a variety of other applications aside from VR and may be more resistant to application updates.

8. Conclusion

VR, AR, and MR technologies will continue to improve leading to additional uses. With increased adoption their forensic importance will further escalate. In this work, we initiate the forensic analysis of the volatile memory of VR systems. We demonstrate that the VE, location, state and class of devices can be extracted from memory. To aid in a forensic investigation, we have also developed a Volatility plugin to automate the extraction of data. Our plugin and methodology has been designed with future VR systems and applications in mind, as we hope to stimulate further research in both VR and memory forensics.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1748950. 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 Science Foundation.

References

Alvarex, V.M., 2018. Yara. https://virustotal.github.io/yara/.
Anglano, C., Canonico, M., Guazzone, M., 2017. Forensic analysis of telegram messenger on android smartphones. Digit. Invest. 23, 31–49.
Bailey, D., 2018. With $4.3 billion in sales, 2017 was steams biggest year yet. https://www.pcgamesn.com/steam-revenue-2017.

Balogh, Š., 2014. Memory acquisition by using network card. J. Cyber Secur. Mobil. 3 (1), 65–76.
Betz, C., 2005. Memparser analysis tool. http://old.dfrws.org/2005/challenge/memparser.shtml.
Bilby, D., 2006. Low Down and Dirty: Anti-forensic Rootkits. https://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Bilby-up.pdf.
Byte, D., 2018. Cheat Engine. http://www.cheatengine.org/.
Case, A., Richard, G.G., 2017. Memory forensics: the path forward. Digit. Invest. 20, 23–33 (Special Issue on Volatile Memory Analysis).
Casey, P., Baggili, I., Yarramreddy, A., 2019. Immersive virtual reality attacks and the human joystick. IEEE Trans. Dependable Secure Comput. http://doi.acm.org/10.1109/TDSC.2019.2907942.
Cohen, M., 2017. Scanning memory with yara. Digit. Invest. 20, 34–43 (Special Issue on Volatile Memory Analysis).
de Guzman, J.A., Thilakarathna, K., Seneviratne, A., 2018. Security and Privacy Approaches in Mixed Reality: A Literature Survey. CoRR abs/1802.05797. URL: http://arxiv.org/abs/1802.05797.
DFRWS, 2006a. Digital Forensics Research Workshop: Forensics Challenge 2006. http://old.dfrws.org/2005/challenge/index.shtml.
DFRWS, 2006b. Digital Forensics Research Workshop: Forensics Challenge 2006. http://old.dfrws.org/2006/challenge/index.shtml.
Dolan-Gavitt, B., 2008. Forensic analysis of the windows registry in memory. Digit. Invest. 5, S26–S32 (The Proceedings of the Eighth Annual DFRWS Conference).
Editor, 2015. https://www.welivesecurity.com/2015/12/24/online-gaming-new-frontier-cybercriminals/.
Gruhn, M., Freiling, F.C., 2016. Evaluating atomicity, and integrity of correct memory acquisition methods. Digit. Invest. 16, S1–S10. DFRWS 2016 Europe.
Hejazi, S., Talhi, C., Debbabi, M., 2009. Extraction of forensically sensitive information from windows physical memory. Digit. Invest. 6, S121–S131 (The Proceedings of the Ninth Annual DFRWS Conference).
Huebner, E., Bem, D., Henskens, F., Wallis, M., 2007. Persistent systems techniques in forensic acquisition of memory. Digit. Invest. 4 (3), 129–137.
International Organization for Standardization, 2017. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.
Jr, G.M.G., Mora, R.-J., 2005. Kntlist analysis tool. http://old.dfrws.org/2005/challenge/kntlist.shtml.
Lamkin, P., 2017. Virtual Reality Headset Sales Hit 1 Million. https://www.forbes.com/sites/paullamkin/2017/11/30/virtual-reality-headset-sales-hit-1-million/#2337ed02b617.
Lemley, M.A., Volokh, E., 2017. Law, Virtual Reality, and Augmented Reality.
Maartmann-Moe, C., Thorkildsen, S.E., rnes, A., 2009. The persistence of memory: forensic identification and extraction of cryptographic keys. Digit. Invest. 6, S132–S140 (The Proceedings of the Ninth Annual DFRWS Conference).
Petroni, N.L., Walters, A., Fraser, T., Arbaugh, W.A., 2006. Fatkit: a framework for the extraction and analysis of digital forensic data from volatile system memory. Digit. Invest. 3 (4), 197–210.
Schatz, B., 2007. Bodysnatcher: towards reliable volatile memory acquisition by software. Digit. Invest. 4, 126–134.
Stevens, R.M., Casey, E., 2010. Extracting windows command line details from physical memory. Digit. Invest. 7, S57–S63 (The Proceedings of the Tenth Annual DFRWS Conference).
Sylve, J., Case, A., Marziale, L., Richard, G.G., 2012. Acquisition and analysis of volatile memory from android devices. Digit. Invest. 8 (3), 175–184.
UNHcFREG, 2018. Human joystick immersive virtual reality attack. https://www.youtube.com/watch?v=iyK94jFuniM.
Valve Software, 2018. Openvr Sdk. https://github.com/ValveSoftware/openvr.
van Baar, R., Alink, W., van Ballegooij, A., 2008. Forensic memory analysis: files mapped in memory. Digit. Invest. 5, S52–S57 (The Proceedings of the Eighth Annual DFRWS Conference).
Volatility Foundation, 2018. Volatility. https://github.com/volatilityfoundation/volatility.
Wong, J.C., 2016. Sexual harassment in virtual reality feels all too real 'it's creepy beyond creepy. https://www.theguardian.com/technology/2016/oct/26/virtual-reality-sexual-harassment-online-groping-quivr.
Yarramreddy, A., Gromkowski, P., Baggili, I., 2018. Forensic analysis of immersive virtual reality social applications: a primary account. In: 2018 IEEE Security and Privacy Workshops. SPW, IEEE, pp. 186–196.
Zhang, R., Wang, L., Zhang, S., 2009. Windows memory analysis based on kpcr. In: Information Assurance and Security, 2009. IAS'09. Fifth International Conference on, vol. 2. IEEE, pp. 677–680.