Duck Hunt: Memory Forensics of USB Attack Platforms

Tyler Thomas[1], Mathew Piscitelli[1], Bhavik Ashok Nahar[1], Ibrahim Baggili[1]

Abstract

To explore the memory forensic artifacts generated by USB-based attack platforms, we analyzed two of the most popular commercially available devices, Hak5's USB Rubber Ducky and Bash Bunny. We present two open source Volatility plugins, *usbhunt* and *dhcphunt*, which extract artifacts generated by these USB attacks from Windows 10 system memory images. Such artifacts include driver-related diagnostic events, unique device identifiers, and DHCP client logs. Our tools are capable of extracting metadata-rich Windows diagnostic events generated by any USB device. The device identifiers presented in this work may also be used to definitively detect device usage. Likewise, the DHCP logs we carve from memory may be useful in the forensic analysis of other network-connected peripherals. We also quantify how long these artifacts remain recoverable in memory. Our experiments demonstrated that some Indicators of Compromise (IOCs) remain in memory for at least 24 hours.

Keywords: Memory Forensics, Volatility, Bash Bunny, Rubber Ducky, USB Devices, Network Devices

1. Introduction

USB-based attacks present a new and unique challenge to cyberforensics investigators. They allow an attack to be carried out inconspicuously after a compromise of physical security. After a breach in physical security, an attack can be carried out by simply plugging the device into the victim machine. Additionally, the devices are designed to limit the amount of information left on the disk by operating entirely within memory. This introduces a unique challenge because memory forensics, while potentially yielding evidence not found on disks, has its own limitations. Devices that are specifically designed to carry out such attacks are readily available on the market. Their availability and low price point provide low-skill attackers with a straightforward and economical method for conducting targeted attacks.

Contrary to the perception of USB-based attacks being low-profile, these devices do leave evidence. All programs must run in memory and therefore every action the USB device invokes the host system to carry out happens entirely in memory. Additionally, the device drivers must interact with the host Operating System (OS) in order for communication to take place. This activity also has the potential to generate evidence in both volatile memory and disk. These memory-based artifacts may remain extractable for an extended period of time after the attack is over and even after the device is unplugged.

Our objective was to explore the types of Indicators of Compromise (IOC) present on victim machines after attacks by consumer USB attack platforms. Additionally, in order to determine the viability of using these IOCs in the real world, we also quantify how long these IOCs remain in a recoverable state.

In this work, we summarize digital artifacts in volatile memory generated by the two most popular USB-based attack platforms on the market, the Bash Bunny and the USB Rubber Ducky. We measure how long these artifacts remain in memory and how they change over time in relation to user activity. Finally, we present tools and methods for recovering the digital evidence generated by these devices from post-mortem memory dumps in the form of open source[1] plugins for the Volatility Framework. Our plugins are not specific to the examined devices, and may also be useful in the analysis of other devices and applications. We present the following contributions:

- To the best of our knowledge, this is the primary account of memory forensic analysis of USB-based attack platforms.
- We share methods for detecting Rubber Ducky and Bash Bunny use in Windows 10 memory dumps, *which have broader impact on peripheral device forensics.*
- We present *usbhunt*, a Volatility plugin using a novel technique for memory-based USB device detection using Windows 10 telemetry data.
- We present *dhcphunt*, a Volatility plugin using a novel technique for memory-based DHCP log data recovery from the Windows *netsh* utility.
- We share our artifact findings in the Artifact Genome Project [1].

Email addresses: tthom10@unh.newhaven.edu (Tyler Thomas), mpisc1@unh.newhaven.edu (Mathew Piscitelli), bnaha1@unh.newhaven.edu (Bhavik Ashok Nahar), ibaggili@newhaven.edu (Ibrahim Baggili)

[1] https://github.com/unhcfreg/DuckHunt

In Section 2 we provide an overview of necessary background information including how these devices interact with the Windows OS, as well as a brief overview of existing work in memory and USB forensics. Section 3 details our processes and methods for artifact discovery and tool creation. Sections 4 and 5 present and discuss our findings. We describe paths forward in Section 6 and conclude with Section 7.

2. Background Information and Related Work

2.1. Definition of a USB-Based Attack Platform

Figure 1 illustrates how a USB-based attack platform interfaces with the victim machine to carry out its attack. First, the physical security of the target machine is compromised such that a USB device may be inserted and recognized by the OS. Once the device has been connected to the system, it will expand its functionality by spoofing one or more trusted or known device types, such as a serial, storage, Human Interface Device (HID) keyboard, or Remote Network Driver Interface Specification (RNDIS) Ethernet gadget. The attacker device uses the spoofed virtual media to conduct the malicious activity on the victim machine or network. Examples of common objectives include: reconnaissance, exfiltration of data, and delivering and executing a payload. After carrying out the malicious activity, the device is unplugged from the victim machine allowing the attacker to leave the area. The attack is completed with minimal interaction with the target machine.

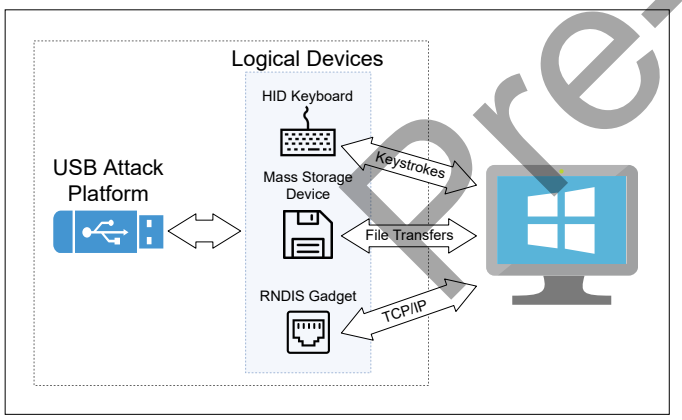

Figure 1: USB Threat Model

2.2. Why Does This Pose a Unique Threat?

The aforementioned approach is appealing to attackers for a variety of reasons. Foremost, plugging a USB device into a machine and waiting is a non-intrusive and inconspicuous action that is not likely to attract attention. Additionally, the entire process may be automated and executed with minimal user interaction. Finally, the attacker footprint can theoretically be minimized by utilizing the device as an external computing resource and executing parts of the malicious code on the USB device rather than the host system.

In achieving this level of automation and self containment, an attacker is able to keep the number of operations and amount of time required to complete an attack at an absolute minimum. This speed and efficiency allows the attacker to craft payloads to operate in ways that attempt to circumvent traditional forensic methods. For example, a payload may exfiltrate passwords directly to the device via a TCP/IP connection over an RNDIS Ethernet interface. Thereby, credentials are exfiltrated via a network connection without generating traffic on the local network. Because of the unique capabilities of these devices, it is necessary for researchers to devise forensic techniques able to detect their use regardless of the anti-forensic techniques they might employ.

2.3. Existing Methods for USB Attack Platform detection

[2] and [3] present a comprehensive overview of USB-based attacks. The devices investigated in this work operate primarily by spoofing HID peripherals, RNDIS Ethernet gadgets, and mass storage devices.

The majority of existing research on detection methods is timing based. By observing typical human typing patterns, it is possible to use an anomaly based approach for detecting spoofed HID's [4, 5]. This approach leverages the significant body of work on using keystroke dynamics for biometric authentication [6, 7, 8]

The anomaly detection model presented by [9] takes into account not only the USB packet timing, but several other factors including the content of the packets and metadata related to the transfer. Most notably, [10] took a radically different approach by demonstrating that temporal anomaly detection methods can be fooled. The authors then implemented a side channel analysis detection method taking into account power usage, keystroke sound, and error correction idiosyncrasies to produce a significantly more robust detection mechanism.

As demonstrated by [10], if a model can learn how a person types to detect anomalies or verify their identity, then a model can learn how to impersonate that person's keystroke dynamics to defeat it.

Outside of keystroke dynamics analysis, other work on real time prevention and detection of USB device activity has been focused on packet analysis and establishing secure and trusted devices [11, 12]. [13] presented a method for detecting potentially malicious devices by analyzing Windows event logs. Our work takes a different approach by inspecting the latent forensic artifacts generated in memory by these devices.

2.4. Memory Forensics

Since the domain began to gain notoriety after the 2005 Digital Forensics Workshop (DFRWS) forensics challenge, memory forensics has established itself as a valuable tool for forensic professionals [14].

Memory analysis tools for Windows and Mac OSX operating systems, such as the Volatility Framework, have

advanced rapidly in their ability to traverse process management data structures [15, 16, 17, 18]. Likewise, acquisition methods have improved to combat anti-forensic countermeasures [19, 20, 21].

While the most significant progress has been made at the systems level, there has been little work conducted at the application level. [22] discussed the capability gap in memory analysis techniques for userland based malware detection. Expanding upon this, we posit that robust userland memory analysis would further aid forensic investigators by providing them access to data that might not otherwise be available with traditional network for filesystem forensics. Analysis by [23] demonstrated that poor coding practices can lead to sensitive application data remaining in memory for an extended period of time.

The greatest challenge in making tools that recover forensic artifacts from application memory is that most applications differ wildly in their implementations and runtime environments. Because of this, such tools are typically highly specialized and must overcome significant technical challenges in reverse engineering and traversing closed source data structures, such as presented by [24].

Creating an additional layer of abstraction between the raw process memory and the analyst or tool developer may be a solution. [25] and [26] demonstrated that Java objects could be reconstructed, in some cases even after they were garbage collected. Tools that take advantage of the predictable object structure in managed memory runtimes such as Java and JavaScript show promise in furthering the state of application based userland memory forensics.

3. Methodology

3.1. Scenario Creation

Adversarial activity carried out with the two USB devices was simulated on the lab machine using publicly-available payloads provided by Hak5 [27, 28].

All adversarial activity was conducted within a VMWare Workstation Pro virtual machine running Windows 10. A complete apparatus of the hardware and software utilized in this phase can be seen in Table 1.

Although several Hak5-created payloads were used throughout testing to ensure that artifacts left by several payloads did not differ, the majority of data collection was performed after running one payload on each device. Reverse shell[2] and document exfiltration[3] payloads were used for the Bash Bunny and Rubber Ducky respectively

Both payloads are written in proprietary scripting languages that automate keystrokes. Injected keystrokes open a PowerShell terminal and rapidly type commands into the terminal window to carry out the attack.

[2] https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Payload---reverse-shell

[3] https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Payload---Data-Exfiltration---Backdoor

Once scenario creation concluded, system memory images were obtained by pausing the virtual machine and copying the .vmem file from the virtual machine directory. VMWare for Linux uses this file to maintain a snapshot of the emulated physical memory so that the state may be restored at a later time. It is important to note that virtual machine suspension is implemented differently on Windows systems, requiring the user to manually create a snapshot in order for such a file to be generated. This is noteworthy because this slight change will have effects on tooling attempting to automate the data collection process.

3.2. Obtaining Signatures

Once it was understood how the devices interfaced with the OS, initial memory analysis was conducted with simple string searches. System memory images were obtained and analyzed to serve as a starting point for identifying IOC signatures and potential data structures for extraction.

The Unix *strings* utility was used with the -td option to extract strings from the image and record the offset in a Volatility-compatible format. The resultant string file was searched with the *grep* utility to find known strings related to device activity. Namely, the search strings included: the IP addresses related to the reverse shell attacks, permutations of the strings “Rubber Ducky” and “Bash Bunny”, and the USB vendor and product ID codes of the devices.

These searches yielded a large number of results, which were then fed into the Volatility *strings* plugin to determine the process spaces and memory addresses at which they were located. The memory regions surrounding the search hits were manually inspected with the *volshell* plugin to determine if they were contained within an immediately apparent data structure.

The vast majority of hits were found in free space and did not belong to a process, or were preceded and followed by seemingly random bytes. These hits would be useful in detecting device utilization, but do not provide any additional information related to the time or nature of activity conducted with the device. However, some hits, particularly those within two svchost processes, appeared to be contained within a predictable structure and viable for extraction. Sections 3.3.1 and 3.3.2 detail the process by which these structures are identified, traversed, and extracted to obtain forensically relevant data related to actions taken by the devices.

3.3. Plugin Development

The strings described in the aforementioned signatures can be used as IOCs to detect Rubber Ducky or Bash Bunny usage on a given system. Part of our contribution is the presentation of these strings as YARA rules to be used to rapidly determine if a USB Rubber Ducky or Bash Bunny was recently connected to a computer suspected of being compromised. This initial signature detection may serve to inform a deeper investigation into what actions may have been carried out with the device.

Table 1: Workstation Details

System Details		Software Details	
Device	**Details**	**Software**	**Version**
Processor	Intel Core i7-8750H	VMWare Workstation Pro	15.5.1
System Type	64-bit OS, x64 processor	Volatility	2.6
Virtual Memory (VRAM)	2.00 GB	Windows 10	1903_18362
Bash Bunny	Firmware v1.6		
Rubber Ducky	Firmware v1.0		

3.3.1. usbhunt

While reviewing the string hit results discussed in Section 3.2, it was noted that several instances of the USB vendor and product ID were contained within large JSON structures. Further review of the JSON structures indicated that they were recording Microsoft telemetry and diagnostic data in an svchost process. A full list of required Microsoft diagnostic events can be found on Microsoft's website[4].

After reviewing the list telemetry events Microsoft requires to be enabled, it was determined that the the events most related to USB device activity were *Windows.Kernel.DeviceConfig.DeviceConfig* and *Windows.Inventory.Core.InventoryDevicePnpAdd*. These events are in no way specific to the Bash Bunny or Rubber Ducky. In theory, they will be generated by any USB device.

According to the Microsoft documentation, the *DeviceConfig* event "provides information about drivers for a driver installation that took place within the kernel." [29] Because this event is related to driver installation, the event should theoretically only be present in memory the first time the device is plugged in and the drivers are installed. In our testing, this was in fact the case and the events were not generated by subsequent attacks.

The second diagnostic event of note, *InventoryDevicePnpAdd* is described in the Microsoft documentation as containing: "basic metadata about a PNP device and its associated driver to help keep Windows up to date. This information is used to assess if the PNP device and driver will remain compatible when upgrading Windows." [29] This event is generated whenever a new virtual device is created by the OS. Because this event is not related to the initial installation of drivers, it was found to be in memory after rebooting the system and carrying out subsequent attacks.

Both the *DeviceConfig* and *InventoryDevicePnpAdd* event JSON structures contained timestamps related to device connectivity and other metadata of forensic relevance, the specifics of which will be discussed in Section 4.

We created *usbhunt*, a Volatility plugin for extracting these structures from Windows 10 memory dumps and presenting them to the command line. The tool works by locating the start address of potential JSON event structures using YARA scans. Once a candidate structure has been detected, they are serialized using the JSON extraction process detailed in Algorithm 1. It is not enough to simply serialize the structure with a standard JSON string processing library because the structure may be damaged or truncated if it is partially overwritten.

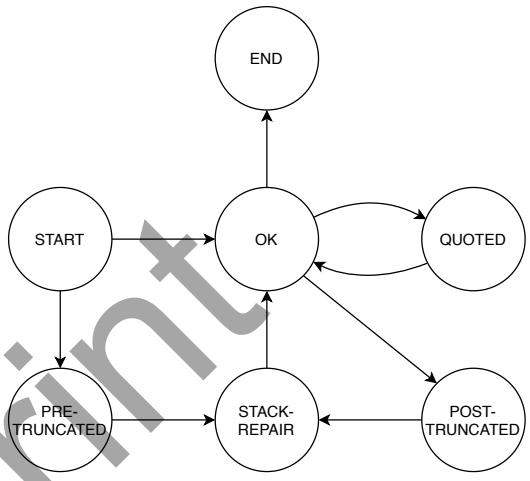

Figure 2: JSON Reconstruction State Machine Diagram

The JSON processing algorithm was implemented as a Deterministic Finite Automaton (DFA) state machine. It consists of a series of states, beginning with a starting state. The algorithm iterates over the data and conducts the operation relevant to the current state context. The state is then changed dependent upon conditional evaluations of the previous state. Once the ending state is reached, the JSON has been fully repaired and is capable of being parsed as a Python dictionary. This algorithm operates in *O(n)* time.

Due to the nature of the state machine, any valid input will produce a valid output. In the case of this work, a valid input is defined as a JSON structure which was overwritten at the beginning (pre-truncated) or at the end (post-truncated). This work does not cover the cases in which the data was intact at the beginning and end, but overwritten in the middle. Figure 2 shows the state flow for the JSON reconstruction algorithm.

[4] https://docs.microsoft.com/en-us/windows/privacy/basic-level-windows-diagnostic-events-and-fields-1903

Algorithm 1 JSON Reconstruction

```
for char ∈ data do                          ▷ Delimiter stack creation
    if isDelimeter(char) then
        stack.push(char)
    end if
end for
state ← START, index ← 0
while state ≠ END do                        ▷ Begin reconstruction
    if not invalidStart(stack) then
        state ← PRE-TRUNCATED
        rebuildHead(data, stack)
    else if prematureEnd(stack) then
        state ← POST-TRUNCATED
        rebuildTail(data, stack)
    else if end of data and isValid(stack) then
        state ← END, index ← index + 1
    else state ← OK                         ▷ Proceed through data
    end if
    process(state, data, index)
end while
```

3.3.2. dhcphunt

While searching for instances of the IP address connected to with the reverse shell payload, some search hits appeared to be highly regular and predictable strings with a structure similar to network logs. Further analysis demonstrated that the these logs were contained within an svchost process responsible for providing data to the *netsh* Windows utility.

Our second Volatility plugin, *dhcphunt*, extracts these logs and presents them in the command line. The majority of these logs are related to DHCP client activity. All log entries begin with a timestamp and a static and predictable string that precedes the variable data. Similar to the extraction process for *usbhunt*, the first step in extracting *netsh* log data is conducting YARA scans to determine the start address of the extractable string.

Once the log entry is located, it can be extracted by reading a fixed number of bytes before and after the target string. This can be used to effectively reconstruct the full timeline of events presented in the *netsh* utility. Basic string formatting operations are performed to ensure that the string is not corrupted and does not contain any unprintable characters.

3.4. Data Collection and Visualization

In order to determine how long these structures were present and whether or not they could be reliably expected to be in memory, we utilized an existing memory acquisition and visualization framework presented in [30].

Using a combination of Bash scripting and Volatility plugins, system memory images were acquired and analyzed at regular intervals after adversarial activity was simulated on the experimental virtual machine. For each memory image, a CSV file was generated that contained extracted data structures, IOCs, and their address offsets.

This process was carried out several times for the Bash Bunny using a reverse shell payload, and for the Rubber Ducky using a document exfiltration payload. Collection took place over a 24 hour period. The interval between acquisitions varied between one and two minutes, depending on how long the Volatility analysis and CSV generation took from the previous iteration.

The CSVs generated by the collection tooling was then fed into the second half of the framework in order to gain insights into data lifetime and availability. The visualization framework was used to illustrate two variables in relation to time: artifact presence, and artifact integrity. The former, shown in Figures 3 and 4, calculates how many artifacts of each type were in memory at each time interval over the course of the data collection. The latter determines how long artifacts remain in memory before they are overwritten, and calculates the degree to which they are corrupted over time. The analysis and ramifications of this data are discussed in detail in Sections 4 and 5.

4. Findings

Example output from our *dhcphunt* tool can be seen in Figure 5. The tools are invoked as standard Volatility plugins and do not accept any command line arguments. Likewise, example JSON structures extracted by *usbhunt* can be seen in Listings 1 and 2. These examples are not the complete JSON structures found in memory. Redundant and forensically irrelevant data was redacted for brevity.

4.1. Indicators of Compromise

Several byte sequences uncovered during the course of this analysis can be used as indicators that a USB Rubber Ducky or Bash Bunny was used against a system. Table 2 details these indicators.

Every USB device has a vendor and product identification code that is used by OSs to identity the manufacturer and product to know what drivers are necessary to interact with the device. In the case of the Bash Bunny, F000 is used as a vendor ID and FF03 as a product ID. There is no vendor registered with this ID meaning that any devices utilizing it can be viewed with suspicion. The string in Table 2 shows how these ID's appear in system memory. The presence of these ID's is indicative of Bash Bunny activity on a system.

Conversely, the Rubber Ducky attempts to obfuscate its USB ID by using the vendor and product ID's of 05AC and 0220 respectively. This ID combination is registered by Apple as an HID keyboard. While the Rubber Ducky does not use a suspicious USB ID, the device instance ID and hardware ID it provides to Windows contains the conspicuous sub-string "Ducky_Storage". While the Apple HID USB ID is not itself suspicious, these two indicators taken together can be used to infer Rubber Ducky usage.

Likewise, the PowerShell payloads executed by both USB devices were found in clear text in memory after the

Device	Type	Example
Bash Bunny	Reverse Shell Payload	*Q STRING "powershell -W Hidden \"Remove-ItemProperty...*
Rubber Ducky	USB ID	VID_05AC&PID_0220
Bash Bunny	USB ID	VID_F000&PID_FF03
Rubber Ducky	Device Instance Id	USBSTOR\\DISK&Ven_ATMEL&Prod_Ducky_Storage
Rubber Ducky	Hardware Identifier	USBSTOR\\DiskATMEL___Ducky_Storage___1.00

Table 2: Indicators of Compromise

attacks. We provide a complete list of these IOCs in our publicly shared Github repository[5] in the form of YARA rules.

4.2. In-memory DHCP Logs

The DHCP logs introduced in Section 3.3.2 and extracted with our *dhcphunt* plugin contain timestamps and information related to the local DHCP client instance. This information is relevant because some devices spoof RNDIS Ethernet gadgets and obtain IP addresses. This can be used to detect such activity.

A full list of the log messages our plugin extracts are enumerated in Table 3. This is by no means a complete list of all possible log messages in the svchost, but only the ones that contained IP addresses and were deemed to be most relevant to the current application. Compiling a complete list would prove difficult as Microsoft does not provide documentation.

In the case of both the USB Rubber Ducky and the Bash Bunny, multiple instances of these logs were present in memory throughout the duration of the experiment. This is indicative that the logs are not deallocated while the system process is running. From a forensic perspective, the artifacts being contained in system processes and not being tied to peripheral hardware or running third-party software is an ideal scenario. However, it was noted that as new logs were generated, existing logs were overwritten. While DHCP client log data can expected to be in memory at any given time, the amount of time that individual log events are retained in memory varies from several minutes to many hours.

The *DeviceConfig* and *InventoryDevicePnpAdd* diagnostic telemetry events are generated by driver installation and device connectivity respectively. The events are stored in an svchost process in a JSON-formatted UTF-8 encoded string. The JSON structures contain large amounts of meta-data related to the the device including time stamps and driver versions. See Table 4 for a full itemization of the forensically relevant data contained within these structures.

[5] https://github.com/unhcfreg/DuckHunt

Listing 1 Example InventoryDevicePnpAdd


```
{
 "name": "Microsoft.Windows.Inventory.Core.InventoryDevicePnpAdd",
 "time": "2020-07-31T16:48:58.7911891Z",
 "ext": {
  "protocol": {
   "devMake": "VMware, Inc.",
   "devModel": "VMware7,1"
  },
  "user": {
   "localId": "w:4D621E97-299E-7C02-BD51-54C81554ED6D"
  }
 },
 "data": {
  "ContainerId": "{70c9a979-a0cd-56e2-afec-26b5c53a8ce3}",
  "ParentId": "usbid_05ac&pid_0220\\123123123123",
  "Description": "USB Mass Storage Device",
  "BusReportedDescription": "HID Keyboard and MSC",
  "ClassGuid": "{36fc9e60-c465-11cf-8056-444553540000}",
  "Manufacturer": "Compatible USB storage device",
  "Model": "USB Mass Storage Device",
  "Inf": "usbstor.inf",
  "DriverName": "usbstor.sys",
  "DriverVerVersion": "10.0.18362.1",
  "DriverVerDate": "06-21-2006",
  "Provider": "Microsoft",
  "DriverPackageStrongName": "usbstor.inf_amd64_c04619397269cc4c",
  "DriverId": "0000d0f4d0ac3c2b29dbc671b22034e209cb410dd9b2",
  "Service": "usbstor",
  "baseType": "Ms.Device.DeviceInventoryChange"
 }
}
```


DHCP Log Messages
Recieving (sic) a DHCP message on ...
ACK of ... from ...
DhcpSetIpRoute: ADD: Dest= ...
Adding the address of ...
Successfully Plumbed the address: ...

Table 3: DHCP Client Logs Extracted by *dhcphunt*

4.3. Windows Diagnostic Events

Some notable data points include: the time the event was generated, driver installation timestamp and version, the Globally Unique Identifier (GUID) of the user account that the event was generated by, USB vendor and product ID, and the GUID of the "container" the virtual USB device belongs to.

The container GUID is used to identify the physical device that the virtual USB devices belong to. This is relevant because some USB attack platforms, namely the Bash Bunny, are able to spoof multiple logical USB devices. In

Listing 2 Example DeviceConfig


```
{
    "name": "Microsoft.Windows.Kernel.DeviceConfig.DeviceConfig",
    "time": "2020-07-31T16:45:56.5111482Z",
    "ext": {
        "device": {
            "localId": "s:B85550C1-8574-4594-92F0-16BD531BF5FD",
            "deviceClass": "Windows.Desktop"
        },
        "protocol": {
            "devMake": "VMware, Inc.",
            "devModel": "VMware7,1"
        },
        "user": {
            "localId": "w:4D621E97-299E-7C02-BD51-54C81554ED6D"
        }
    },
    "data": {
        "DeviceInstanceId": "USB\\VID_F000&PID_FF03&MI_00",
        "FirstHardwareId": "USB\\VID_F000&PID_FF03&REV_0333&MI_00",
        "LastCompatibleId": "USB\\Class_ef",
        "ClassGuid": "{4d36e972-e325-11ce-bfc1-08002be10318}",
        "DriverInfName": "rndiscmp.inf",
        "DriverProvider": "Microsoft",
        "DriverDate": "06/21/2006",
        "DriverVersion": "10.0.18362.1",
        "InstallDate": "2020-09-21T15:44:40.6689571Z"
    }
}
```


the case of the reverse shell Bash Bunny payload carried out during scenario creation, the device spoofs an RNDIS Ethernet gadget and an HID keyboard simultaneously to send keystrokes and create a TCP/IP connection to the device. The container ID can be used by an investigator to link these two logical devices. Further discussion on the relevance of these fields is explored Section 5.

Figures 3 and 4 provide a visual representation of how long these events persist in memory in an extractable form. An overview of artifact memory persistence is discussed in Section 5

Table 4: Relevant Windows Diagnostic Event fields

Field	Description
DeviceConfig	
— time	Timestamp of event creation
— ext.user.localId	GUID of user
— data.DriverInfName	Device driver
— data.InstallDate	Driver installation date
— data.DeviceInstanceId	Device vendor and product ID
InventoryDevicePnpAdd	
— time	Timestamp of event creation
— device.user.localId	GUID of user
— data.HWID.Value	Device vendor and product ID
— data.containerId	Parent device container ID
— data.Description	Vendor supplied description
— data.Manufacturer	Vendor supplied manufacturer
— data.Model	Vendor supplied model information
— data.Inf	Device driver
— data.Provider	Driver provider

4.4. Evidence of Payloads

The Rubber Ducky's PowerShell payload was found in memory in its entirety as part of a DOS command. The string found in memory was part of the Ducky Script text, rather than logged PowerShell. This is supported by the presence of a Ducky Script variable, in this case "{BACKSLASH}."

Similarly, the reverse shell PowerShell payload executed by the Bash Bunny was also retrieved from memory in its entirety. In this case, two versions were found: both the line of code written in the Bash Bunny's scripting language which wrapped the PowerShell, and the standalone PowerShell which was executed on the Windows machine. The former is recognizable by the scripting command "Q STRING" followed by the quoted PowerShell script block. Notably, in memory dumps taken immediately after the attack was carried out, the payloads were present in their entirety.

Our investigation did not focus on artifacts generated by specific payloads, as their behavior varies dramatically. However, Hak5 makes many payloads publicly available and the scripts are present in memory after an attack. Therefore, by compiling a list of known payloads, our YARA rules can be used to detect common payload execution on a system.

5. Discussion

The diagnostic events extracted by *usbhunt* are agnostic to OS structures and memory context. In other words, it does not matter if the Virtual Address Descriptor (VAD) that originally contained the string still belongs to the process. If the memory regions that contained the strings have not been overwritten by another process, the diagnostic structures are still extractable. This is especially useful given that anti-forensic countermeasures can be taken by an attacker to clear the event viewer and Windows registry. Even if an attacker attempts to cover their tracks in this way, the diagnostic events may still be present in memory.

The *DeviceConfig* and *InventoryDevicePnpAdd* diagnostic events share common fields; however, *InventoryDevicePnpAdd* contains significantly more device metadata. Both events contain the vendor and product ID, GUID of the user account that was responsible for the event, and time of event creation. With these pieces of metadata a forensic investigator is capable of answering three critical questions: (1) What device was plugged in? (2) Who plugged it in? (3) When was it plugged in? Additionally, the name of the device driver is also included in both events, which can be used to make inferences on the capabilities and nature of the connected device.

Several fields are unique to *InventoryDevicePnpAdd*. Namely, the container ID, description, manufacturer, model, and driver provider. The container ID is useful in associating virtual devices belonging to a common physical device. The description, manufacturer, and model are all provided by the device and can therefore not be reliably used to detect a potentially malicious device as they can be altered. However, the Rubber Ducky makes no attempt

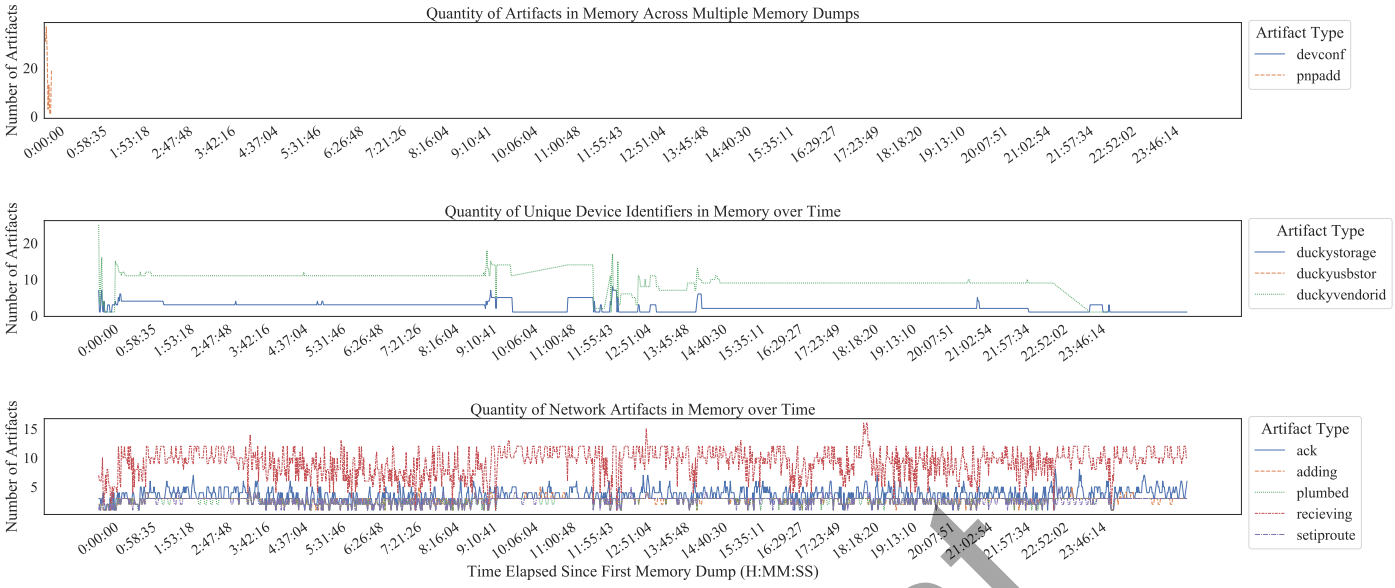

Figure 3: Rubber Ducky - Artifacts in Memory over Time

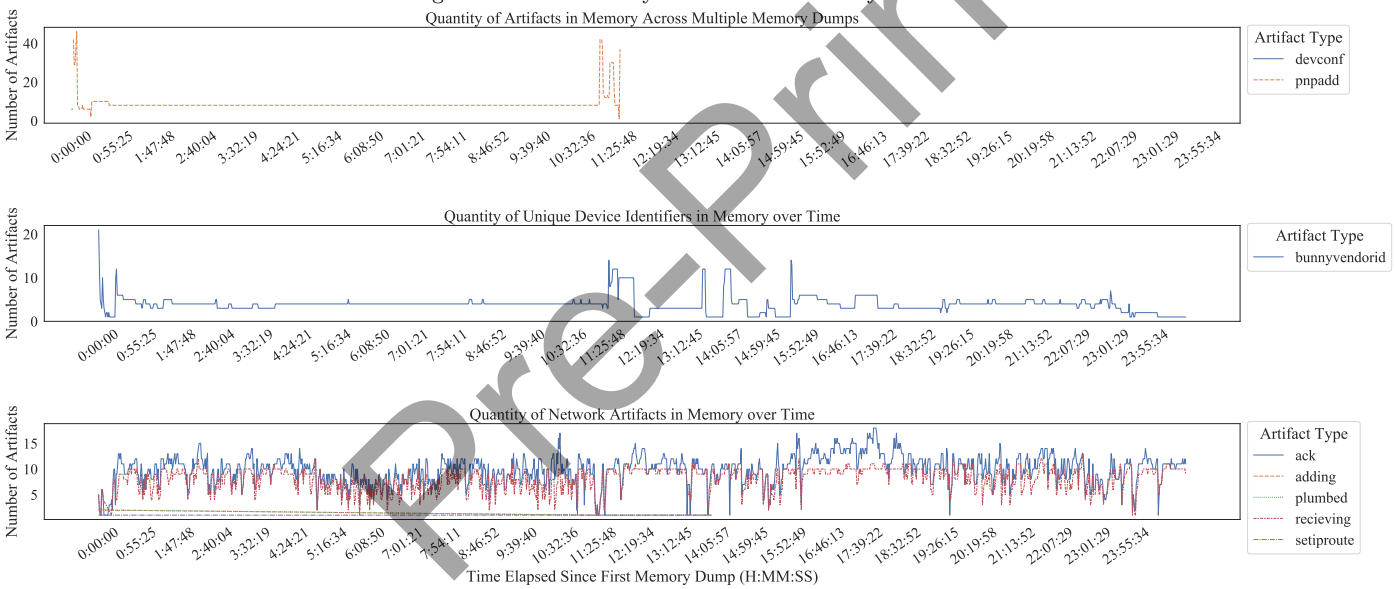

Figure 4: Bash Bunny - Artifacts in Memory over Time

Volatility Foundation Volatility Framework 2.6.1
Address Log Message
---------- ----------

0x9a9327b 9-21-2020 11:35:55:447Recieving a DHCP message on {d0ed74cd-51ce-4597-91bf-599fb6a2d0ce}. Error code is 0
0x9a935fb 9-21-2020 11:35:55:447Recieving a DHCP message on {d0ed74cd-51ce-4597-91bf-599fb6a2d0ce}. Error code is 0
0x9a93f9b 9-21-2020 11:44:15:426Recieving a DHCP message on {d0ed74cd-51ce-4597-91bf-599fb6a2d0ce}. Error code is 0
0x9a9397b 9-21-2020 11:35:55:463Adding the address of 192.168.232.128 on {d0ed74cd-51ce-4597-91bf-599fb6a2d0ce}. Error code is 0.
0x9a9389b 9-21-2020 11:35:55:463Successfully Plumbed the address: 192.168.232.128
0x9a936dc 9-21-2020 11:35:55:447ACK of 192.168.232.128 from 192.168.232.254 is accepted on {d0ed74cd-51ce-4597-91bf-599fb6a2d0ce}.
0x48d9403b 9-21-2020 11:44:46:946Recieving a DHCP message on {93a38501-7a81-44c9-ad0a-4430f5e690f9}. Error code is 0
0x48d943bb 9-21-2020 11:44:46:962Recieving a DHCP message on {93a38501-7a81-44c9-ad0a-4430f5e690f9}. Error code is 0
0x48d9457b 9-21-2020 11:44:46:962DhcpSetIpRoute: ADD: Dest=0.0.0.0, DestMask=0.0.0.0, NextHop=172.16.64.1, Metric=0, Address= 172.16.64.10
0x48d9481b 9-21-2020 11:44:46:993Adding the address of 172.16.64.10 on {93a38501-7a81-44c9-ad0a-4430f5e690f9}. Error code is 0.
0x48d9473b 9-21-2020 11:44:46:993Successfully Plumbed the address: 172.16.64.10
0x48d9449c 9-21-2020 11:44:46:962ACK of 172.16.64.10 from 172.16.64.1 is accepted on {93a38501-7a81-44c9-ad0a-4430f5e690f9}.
0x4d072d9b 9-21-2020 11:44:45:946Recieving a DHCP message on {93a38501-7a81-44c9-ad0a-4430f5e690f9}. Error code is 121
0x4d07207c 9-21-2020 11:44:15:426ACK of 192.168.232.128 from 192.168.232.254 is accepted on {d0ed74cd-51ce-4597-91bf-599fb6a2d0ce}.
0x71576b7b 9-21-2020 11:35:50:649Recieving a DHCP message on {d0ed74cd-51ce-4597-91bf-599fb6a2d0ce}. Error code is 1223

Figure 5: *dhcphunt* Example Output

at hiding its identity by spoofing these fields by presenting itself as "AMTEL Ducky Storage USB Device".

The only field that is unique to the *DeviceConfig* event is the driver installation timestamp. This is potentially useful for an investigator if they are trying to determine the time and date the device was plugged in for the first time.

Analysis of the data collected in our experiment revealed that these diagnostic fields are present for a significant amount of time. While the *InventoryDevicePnpAdd* events remained in memory for just under an hour in the case of the USB Rubber Ducky (Figure 3), the same events generated by the Bash Bunny were present for over 11 hours (Figure 4). Therefore, the information discussed above, including identifiers and forensically-relevant timestamps, could feasibly be extracted from memory if the host is accessed within a reasonable time frame.

We posit that the presence of events in memory for the Rubber Ducky dropped quicker than that of the Bash Bunny due to implementation differences between the devices. The Rubber Ducky only utilizes a single logical HID, while the Bash Bunny created two logical devices. This resulted in significantly more events being generated and therefore remained in memory for a longer period.

Furthermore, our time series data has shown that the unique identifiers for both devices discussed in Section 4.1 remain in memory in some capacity for at least twenty-four hours (second plot of Figures 3 and 4). Therefore, in a forensic context, should investigators gain access to a device compromised by a USB attack before it is power cycled, it is highly likely that these identifiers could be extracted. It is worth noting that the actual duration they remain in memory could be significantly longer, but our work was limited to the duration of the experiment. Moreover, we posit that while the exact identifiers tracked in our work pertain to the USB Rubber Ducky and Bash Bunny, the lifetime of any USB device's hardware identifier and device instance ID should be similar.

We also measured the degree to which these artifacts are overwritten in memory over time. In the case of the indicators of compromise (Device Instance ID, Hardware Identifier, USB IDs) and the Windows diagnostic events, they remained perfectly intact without any corruption for as long as they remained in memory. As such, when extracted, they would be in their unmodified original form and would be forensically valid. This is likely do to their storage in OS processes which remain active until a power cycle occurs. On the other hand, DHCP client logs were highly volatile. As evidenced by Figures 3 and 4, new log events are consistently being generated throughout the course of data collection as normal DHCP client activity resumes.

It is also worth noting that the pattern of DHCP log events generated after disconnection differ significantly between the devices. The Bash Bunny trial resulted in substantially more "ACK" messages being present in memory. Rerunning the tests several times for both devices resulted in the pattern being replicated. The potential implications of this are discussed in Section 6.

6. Future Work

Since our Volatility plugins are not specific to the devices investigated in this work, they can be used in the memory forensic analysis of other devices or applications. The plugin leveraging device driver diagnostics, *usbhunt*, can detect the use of any USB peripheral and extract metadata related to its usage, even if anti-forensic countermeasures are taken to prevent traditional forensic analysis using the registry and event viewer. Because of this unique capability, the tool may be useful in the forensic analysis of other USB devices. Likewise, the DHCP client log extraction plugin, *dhcphunt*, can be useful in the analysis of network traffic regardless of the applications or devices generating it.

While the scope of our experiment was limited to twenty-four hours and did not involve constant user activity, an expanded trial would provide insight into how USB artifacts persist in memory. A test conducted over multiple days and employing simulated user actions, such as automated web browsing, might better demonstrate the extractability of this data in a real-world scenario. Moreover, because our plugins were tested on VMEM files, it may be worthwhile to run an experiment with memory dumps obtained via alternative acquisition methods, such as *DumpIt* or *NotMyFault*.

While *DeviceConfig* and *InventoryDevicePnpAdd* are the only diagnostic events relevant to the analysis of USB devices, there may be promise in using diagnostic events to extract metadata related to other devices and system activity. The Microsoft documentation for the required diagnostic events is extensive and can be explored as a potential starting point for future research. Additionally, Microsoft only provides documentation for events that cannot be disabled. There are an unknown number of optional diagnostic events that are undocumented. If fully enumerated, these events may potentially provide a wealth of forensically relevant data. It is well understood that Microsoft records a large amount of information related to user and system activity. Researchers and investigators may be able to leverage this to collect evidence from system memory images that may not be accessible via traditional forensic techniques.

A discrepancy exists between the USB Rubber Ducky and Bash Bunny in their ratios of network log artifacts. The Rubber Ducky, lacking an RNDIS Ethernet gadget, generated significantly more "Recieving" (sic) events than "ACK" events. Conversely, the Bash Bunny, which contains an Ethernet gadget, produced slightly more "ACK" events than "recieving" events. We speculate that patterns in these network event logs could be used to identify an unknown device. A sound method of fingerprinting devices based on these patterns may help with forensic analysis if an unknown device has spoofed its unique identifiers.

7. Conclusion

By leveraging Windows telemetry diagnostic events, it is possible to collect large amounts of metadata related to USB device usage up to 11 hours after the disconnection of the device. Carving these events from images of physical memory allows for their recovery, even in the event of anti-forensic countermeasures attempting to prevent traditional methods of USB detection.

Moreover, specific malicious devices, such as the Bash Bunny and Rubber Ducky, can be fingerprinted by analyzing the latent artifacts present in memory after their use. These can then be used to scan system memory images for IOCs. In our testing, the artifacts generated by both devices stayed in memory for over 24 hours, the duration of the testing period, and may potentially be present for as long as the system remains powered on.

Additionally, scripts carried out by the aforementioned devices can be found in plaintext in their entirety in memory after the attacks were performed and the USB device was disconnected. In conjunction with the indicators of compromise and diagnostic events, these artifacts can be used to paint a complete picture of the nature of the USB device.

References

[1] Cinthya Grajeda, Laura Sanchez, Ibrahim Baggili, Devon Clark, and Frank Breitinger. Experience constructing the artifact genome project (agp): Managing the domain's knowledge one artifact at a time. *Digital Investigation*, 26:S47 – S58, 2018. ISSN 1742-2876. URL https://doi.org/10.1016/j.diin.2018.04.021.

[2] Nir Nissim, Ran Yahalom, and Yuval Elovici. Usb-based attacks. *Computers & Security*, 70:675 – 688, 2017. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2017.08.002. URL http://www.sciencedirect.com/science/article/pii/S0167404817301578.

[3] Jing Tian, Nolen Scaife, Deepak Kumar, Michael Bailey, Adam Bates, and Kevin Butler. Sok:" plug & pray" today–understanding usb insecurity in versions 1 through c. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 1032–1047. IEEE, 2018.

[4] Ferdous A. Barbhuiya, Tonmoy Saikia, and Sukumar Nandi. An anomaly based approach for hid attack detection using keystroke dynamics. In Yang Xiang, Javier Lopez, C.-C. Jay Kuo, and Wanlei Zhou, editors, *Cyberspace Safety and Security*, pages 139–152, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-35362-8.

[5] Sebastian Neuner, Artemios G Voyiatzis, Spiros Fotopoulos, Collin Mulliner, and Edgar R Weippl. Usblock: Blocking usb-based keypress injection attacks. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 278–295. Springer, 2018.

[6] Francesco Bergadano, Daniele Gunetti, and Claudia Picardi. User authentication through keystroke dynamics. *ACM Transactions on Information and System Security (TISSEC)*, 5(4): 367–397, 2002.

[7] Fabian Monrose and Aviel Rubin. Authentication via keystroke dynamics. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 48–56, 1997.

[8] Fabian Monrose and Aviel D Rubin. Keystroke dynamics as a biometric for authentication. *Future Generation computer systems*, 16(4):351–359, 2000.

[9] Amin Kharraz, Brandon L. Daley, Graham Z. Baker, William Robertson, and Engin Kirda. USBESAFE: An end-point solution to protect against usb-based attacks. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 89–103, Chaoyang District, Beijing, September 2019. USENIX Association. ISBN 978-1-939133-07-6. URL https://www.usenix.org/conference/raid2019/presentation/kharraz.

[10] Nitzan Farhi, Nir Nissim, and Yuval Elovici. Malboard: A novel user keystroke impersonation attack and trusted detection framework based on side-channel analysis. *Computers & Security*, 85:240–269, 2019.

[11] Dave (Jing) Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making USB great again with USBFILTER. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 415–430, Austin, TX, August 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tian.

[12] Ryad Benadjila, Arnauld Michelizza, Mathieu Renard, Philippe Thierry, and Philippe Trebuchet. Wookey: Designing a trusted and efficient usb device. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 673–686, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376280. doi: 10.1145/3359789.3359802. URL https://doi.org/10.1145/3359789.3359802.

[13] Chia-Yu Huang, Hahn-Ming Lee, Jiunn-Chin Wang, and Ching-Hao Mao. Identifying hid-based attacks through process event graph using guilt-by-association analysis. In *Proceedings of the 3rd International Conference on Cryptography, Security and Privacy*, ICCSP '19, page 273–278, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366182. doi: 10.1145/3309074.3309080. URL https://doi.org/10.1145/3309074.3309080.

[14] DFRWS. Digital forensics research workshop: Forensics challenge 2005, 2005. http://old.dfrws.org/2005/challenge/.

[15] Andrew Case, Ryan D Maggio, Modhuparna Manna, and Golden G Richard III. Memory analysis of macos page queues. *Forensic Science International: Digital Investigation*, 33:301004, 2020.

[16] Brendan Dolan-Gavitt. Forensic analysis of the windows registry in memory. *Digital Investigation*, 5:S26 – S32, 2008. ISSN 1742-2876. doi: https://doi.org/10.1016/j.diin.2008.05.003. The Proceedings of the Eighth Annual DFRWS Conference.

[17] Andreas Schuster. The impact of microsoft windows pool allocation strategies on memory forensics. *Digital Investigation*, 5:S58 – S64, 2008. ISSN 1742-2876. URL https://doi.org/10.1016/j.diin.2008.05.007. The Proceedings of the Eighth Annual DFRWS Conference.

[18] Joe T. Sylve, Vico Marziale, and Golden G. Richard. Pool tag quick scanning for windows memory analysis. *Digital Investigation*, 16:S25 – S32, 2016. ISSN 1742-2876. URL https://doi.org/10.1016/j.diin.2016.01.005. DFRWS 2016 Europe.

[19] Kyoungho Lee, Hyunuk Hwang, Kibom Kim, and BongNam Noh. Robust bootstrapping memory analysis against anti-forensics. *Digital Investigation*, 18:S23 – S32, 2016. ISSN 1742-2876. URL https://doi.org/10.1016/j.diin.2016.04.009.

[20] Johannes Stüttgen and Michael Cohen. Anti-forensic resilient memory acquisition. *Digital Investigation*, 10:S105 – S115, 2013. ISSN 1742-2876. doi: https://doi.org/10.1016/j.diin.2013.06.012. URL http://www.sciencedirect.com/science/article/pii/S1742287613000583. The Proceedings of the Thirteenth Annual DFRWS Conference.

[21] Stefan Vömel and Felix C. Freiling. A survey of main memory acquisition and analysis techniques for the windows operating system. *Digital Investigation*, 8(1):3 – 22, 2011. ISSN 1742-2876. URL https://doi.org/10.1016/j.diin.2011.06.002.

[22] Andrew Case and Golden G. Richard. Memory forensics: The path forward. *Digital Investigation*, 20:23 – 33, 2017. ISSN 1742-2876. doi: https://doi.org/10.1016/j.diin.2016.12.004.

Special Issue on Volatile Memory Analysis.

[23] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, pages 321–336, 2004.

[24] Peter Casey, Rebecca Lindsay-Decusati, Ibrahim Baggili, and Frank Breitinger. Inception: Virtual space in memory space in real space. 2019.

[25] Adam Pridgen, Simson Garfinkel, and Dan S. Wallach. Picking up the trash: Exploiting generational gc for memory analysis. *Digital Investigation*, 20:S20 – S28, 2017. ISSN 1742-2876. URL https://doi.org/10.1016/j.diin.2017.01.002. DFRWS 2017 Europe.

[26] Aisha Ali-Gombe, Sneha Sudhakaran, Andrew Case, and Golden G. Richard III. Droidscraper: A tool for android in-memory object recovery and reconstruction. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 547–559, Chaoyang District, Beijing, September 2019. USENIX Association. ISBN 978-1-939133-07-6. URL https://www.usenix.org/conference/raid2019/presentation/ali-gombe.

[27] Hak5. Payload library for the bash bunny by hak5, 2020. URL https://github.com/hak5/bashbunny-payloads.

[28] Hak5. Usb rubber ducky, 2020. URL https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Payloads.

[29] Windows 10, version 1909 and windows 10, version 1903 required windows diagnostic events and fields. URL https://docs.microsoft.com/en-us/windows/privacy/basic-level-windows-diagnostic-events-and-fields-1903. Accessed: 2021-02-6.

[30] Tyler Thomas, Mathew Piscitelli, Ilya Shavrov, and Ibrahim Baggili. Memory foreshadow: Memory forensics of hardware cryptocurrency wallets–a tool and visualization framework. *Forensic Science International: Digital Investigation*, 33:301002, 2020.