Digital Investigation 26 (2018) S3–S11

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2018 USA — Proceedings of the Eighteenth Annual DFRWS USA

Memory forensics and the Windows Subsystem for Linux

Nathan Lewis [c], Andrew Case [a], Aisha Ali-Gombe [d], Golden G. Richard III [b, c, *]

[a] *Volatility Foundation, USA*
[b] *Center for Computation and Technology, Louisiana State University, USA*
[c] *School of Electrical Engineering & Computer Science, Louisiana State University, USA*
[d] *Department of Computer and Information Sciences, Towson University, USA*

Keywords:
Memory forensics
Computer forensics
Memory analysis
Windows 10
Linux
WSL

A B S T R A C T

The Windows Subsystem for Linux (WSL) was first included in the Anniversary Update of Microsoft's Windows 10 operating system and supports execution of native Linux applications within the host operating system. This integrated support of Linux executables in a Windows environment presents challenges to existing memory forensics frameworks, such as Volatility, that are designed to only support one operating system type per analysis task (e.g., execution of a single framework plugin). WSL breaks this analysis model as Linux forensic artifacts, such as ELF executables, are active in a sample of physical memory from a system running Windows. Furthermore, WSL integrates Linux-specific data structures into existing Windows data structures, such as those used to track per-process metadata as well as userland runtime data. This integration results in existing analysis plugins producing inconsistent results when analyzing native Windows processes compared to WSL processes. Further complicating this situation is the fact that much of the WSL subsystem internals are completely undocumented. To remedy the current deficiencies related to WSL analysis, a research effort was undertaken to understand which existing Volatility plugins are affected by the introduction of WSL as well as what updates are necessary to fully support memory forensics of WSL. This paper describes these efforts, including our study of the operating systems data structures relevant to WSL as well as the development of new Volatility analysis plugins.

1. Introduction

The Windows Subsystem for Linux (WSL) (The Windows Subsystem for Linux, 2017) is a significant new feature that was introduced in the Anniversary Update of Microsoft's Windows 10 operating system. WSL provides the first truly native support for Linux applications on a Windows operating system by implementing loading and execution of ELF applications and libraries. The ability to run native ELF files brings a large and diverse set of existing Linux applications to Windows users, such as web, email, FTP, and SSH servers, as well as a full suite of end-user applications. Along with providing a simple method for transitioning existing applications from Linux to Windows, Microsoft has also pledged a long-term commitment to WSL as reflected in its documentation (MSDN, 2017) and in the large set of updates and new features that were included in the Fall Creators Update (Raj, 2017). The combined effect of these actions suggests that WSL will be present and supported for many years and that defensive security practices must account for its existence.

Unfortunately, the introduction of a new executable file format into Microsoft Windows, along with a very large number of new Linux applications, provides an immense challenge for endpoint software security vendors, such as anti-virus companies (Ionescu, 2016a). While these companies have dedicated nearly two decades of research to understanding and detecting threats from Portable Executable (PE) format files, the native Windows executable file format, the very recent introduction of ELF requires an entirely new set of detection capabilities and algorithms. As described in Section 3, not only does the file new format provide challenges, but the architecture that supports ELF files also introduces many new data structures that make traditional malware detection techniques inadequate.

This gap in traditional Windows analysis techniques affects not only runtime software security vendors, but also memory forensics frameworks, since these frameworks are very sensitive to the location and layout of data structures populated by the operating system. Specifically, the ability to correctly locate and parse these

* Corresponding author.
E-mail addresses: nplewis@lsu.edu (N. Lewis), andrew@dfir.org (A. Case), aaligombe@towson.edu (A. Ali-Gombe), golden@cct.lsu.edu (G.G. Richard).

https://doi.org/10.1016/j.diin.2018.04.018

data structures is a fundamental design component of memory forensics tools and similarly, the ability to locate all relevant memory-resident artifacts is a requirement for thorough malware and anomaly detection. The introduction of new data structures and algorithms by WSL breaks many existing algorithms implemented by current analysis frameworks. Furthermore, a class of malware known as *bashware* can programatically enable WSL and execute malicious code while taking advantage of the obfuscation provided by WSL (Elbaz and Atias, 2017).

To close the detection gaps currently available to attackers through WSL, we conducted research to document the new sources of forensics artifacts produced by WSL as well as creating new memory forensics algorithms that provide better coverage of the WSL subsystem. This paper describes this research and its outcomes, including discussion of the relevant WSL architectural components, the deficiencies in existing memory forensic algorithms, and the new algorithms we created to recover WSL-related memory artifacts. Our research was conducted through reverse engineering of the WSL userland and kernel components as well as testing and creation of Volatility (The Volatility Framework, 2017) plugins. Volatility was chosen as our target memory analysis framework because of its widespread use throughout the digital forensics community combined with its ample documentation. All of our newly created Volatility plugins, along with our patches to existing plugins, will be contributed to the upstream project upon publication of this paper.

2. Related work

2.1. WSL architecture memory analysis research

Internal components of the WSL architecture are closed source and sparsely documented by Microsoft. While Microsoft's MSDN and Windows Internals 7th Edition (Yosifovich et al., 2017) document the high-level design ideas and exported APIs, these references do not describe data structures or algorithms utilized by WSL. Microsoft also does not provide full Visual Studio debugging files (generally referred to as PDB files) for the WSL subsystem.

The only substantial existing memory analysis research for WSL was undertaken by Alex Ionescu and appeared in Blackhat 2016 (Ionescu, 2016a). Code, in the form of WinDbg scripts, related to this effort is publicly available in a Github repository (Ionescu, 2016b). A complete comparison between our research effort and his is provided in Section 4.

Concurrently with our research effort, a member of the Volatility development team, Michael Ligh, published a set of patches that enabled correct reporting of WSL process names (Ligh, 2017). Our team had performed the same research, as discussed in Section 5.

2.1.1. Cygwin for Linux on Windows

Executing Linux programs on Windows systems was possible before the release of WSL. Cygwin is a software project that allows users to execute Linux programs in Windows environments. The Cygwin terminal provides a shell environment from which users can interact with a virtual filesystem, execute supported programs, and issue POSIX system calls (Cygwin, 2017). The Cygwin design is similar to WSL in that both bring lightweight virtualization of Linux environments to Windows systems. However, the ways in which this functionality is provided are significantly different. Cygwin compiles Linux source code into standard PE-formatted executables, which are then linked against a library that provides POSIX compatibility by translating between Unix and Windows system calls. Notably, Cygwin does not introduce ELF files into Windows and operates entirely in userspace, without kernel components. In contrast, WSL is more tightly integrated, introduces support for executing ELF files, and has both userland and kernel space components.

3. WSL background

Microsoft's Drawbridge project team focused its research efforts on application sandboxing, a method for lightweight virtualization. The project's goal was to introduce a library operating system model into a commercial version of Windows that relocated operating system dependencies of sandboxed applications into their process' address spaces (Baumann et al., 2016). Drawbridge first produced a prototype version of Windows 7 using a library OS architecture in 2011 (Porter et al., 2011).

Drawbridge proposed two new process types - *minimal* and *pico* - while retaining support for Microsoft's traditional NT processes. Unlike NT processes, minimal processes lack key Window components that tie NT processes directly to the kernel. Fig. 1 depicts these components. Minimal processes have empty userland memory and are unmanaged by the kernel in many respects. Pico processes are minimal processes that are also associated with a corresponding kernel driver. A pico process' kernel driver is responsible for managing the process' userland memory, threads, scheduling, file handles, and sockets (Hammons, 2016a; Hron, 2017). This driver is commonly referred to as the *pico provider*.

WSL, the most prominent application of pico processes in Windows, was released in 2017 with the 64-bit version of the Windows 10 Fall Creators Update after more than one year of beta testing (Turner, 2017). It enables users to directly execute userland Linux programs in Windows 10 by associating each executing Linux application with a pico process. This allows users to execute ELF binaries without the need for a virtual machine, source code modification, or an intermediate application. Furthermore, users can download an app for each of the five currently supported Linux distributions from the Microsoft Store (Cooley et al., 2017): Ubuntu, Debian GNU/Linux, openSUSE Leap 42, SUSE Linux Enterprise Server 12, and Kali Linux. The following processes are components of WSL's implementation and are illustrated in Fig. 2:

- *wsl.exe* or *bash.exe*: A userland command line process through which users interact with WSL. This program can be instantiated more than once.
- *LxssManager*: A Windows service that facilitates communication between *wsl.exe*/*bash.exe* processes and the WSL pico provider.
- *lxss*: A Windows system service that serves as the WSL pico provider.
- */init*: A Linux pico process that facilitates communication between Windows processes and its descendants. *lxss* creates one */init* process per instantiated Linux distribution.
- */bin/bash*: A Linux pico process that supports the WSL shell program. Each *wsl.exe* and *bash.exe* process is paired with a matching */bin/bash* process.

To start WSL, a user executes the <*distro*>*.exe* program corresponding to a desired Linux distribution, which creates a *wsl.exe* process. A user can also access the system's default distribution by executing *bash.exe* or *wsl.exe* directly. Each execution is isolated by Windows in its own *Linux instance*. The WSL NT services and an*/init* pico process will be created for the user's Linux instance if they don't already exist. The *lxss* service registers itself as the pico provider with the Windows kernel through the `PsRegisterPicoProvider` system call. This instructs the kernel to allow lxss to manage system calls, exceptions, and resources on behalf of WSL pico processes (Hammons, 2016a). A Linux shell GUI will be created if wsl.exe is executed either from within cmd.exe or from the Windows GUI.

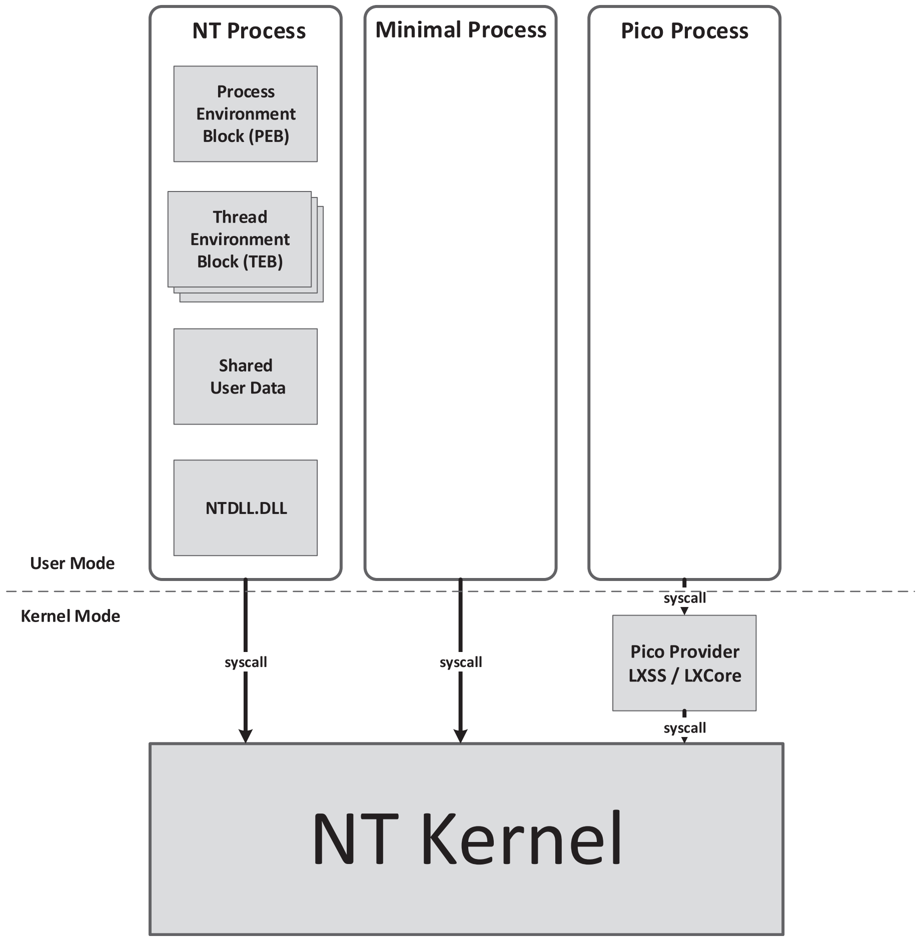


Fig. 1. A comparison of Drawbridge's process types. Each of the components associated with NT processes are left out of minimal and pico processes (Hammons, 2016a).

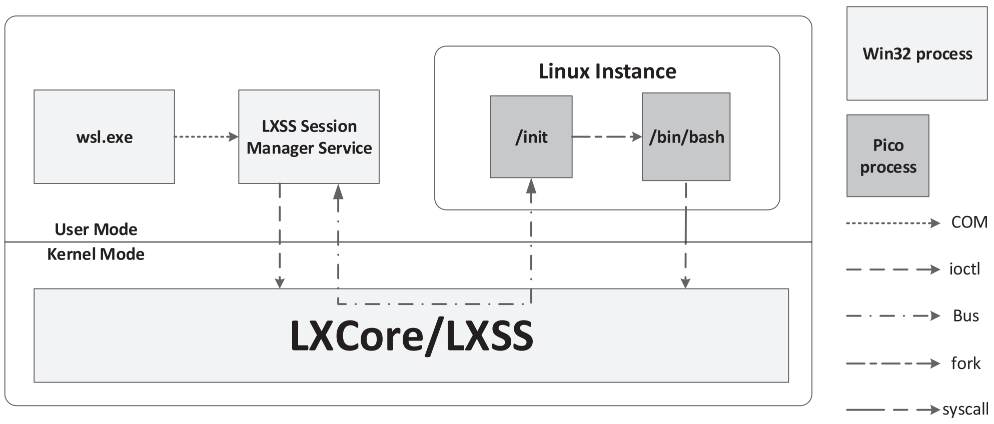


Fig. 2. Communication between components of WSL (Hammons, 2016b).

Alternatively, users can execute wsl.exe with the `-c <command>` argument to execute an ELF binary and immediately return to the calling process without spawning a */bin/bash* GUI (Cooley, 2017).

4. Deficiencies in WSL memory analysis

4.1. Identifying deficiencies

Our research effort began by testing existing memory analysis algorithms, through the use of Volatility plugins, to determine which were affected by the data structure and algorithm changes introduced by WSL. Through this testing, many deficiencies were noted.

First, the name of a pico process is not stored in the traditional *ImageFileName* member of the *_EPROCESS* kernel structure. This causes `pslist`, as well as the numerous other Volatility plugins that print the names of processes, to incorrectly report an empty string as the name of each WSL pico process.

The parent/child relationship between processes is also broken, which affects the `pstree` plugin. With the exception of/init, the usual _EPROCESS structure member for a process' parent is not populated. Furthermore, there is a unique set of process IDs (PIDs) used by the Linux subsystem versus the normal Windows PIDs. This makes it impossible to match process identifiers from Volatility's process listing plugins with those found in WSL log files, such as/var/log/syslog or/var/log/messages, within the Linux filesystem.

As discussed in Ionescu's Black Hat presentation as well as Windows Internals 7th Edition, pico processes do not have an associated process environment block (PEB). For native NT processes, this data structure tracks a number of crucial userland memory artifacts, all of which are missing from WSL pico processes. These missing artifacts and the corresponding Volatility analysis plugins that rely on them are:

Affected Plugin	Missing Artifact
dlllist	List of loaded DLLs
ldrmodules	List of loaded DLLs
cmdline	Command line arguments
envars	Environment variables
procdump	Application base address
dlldump	Base addresses of loaded DLLs
impscan	Location of exported APIs

Along with a missing PEB, WSL pico processes also do not have a traditional handles table. This breaks Volatility's `handles` plugin as it is unable to track which resources, such as files, that a process is utilizing. Tracking threads of execution is also broken since some of the traditional _ETHREAD fields are not populated for threads of WSL pico processes. This affects the `thrdscan` and `threads` plugins.

For non-pico consoles, such as *cmd.exe* and *powershell.exe*, the `cmdscan` and `consoles` plugins enumerate all remnant input and output generated on the consoles. These plugins operate by focusing on data structures inside of the server components of these client consoles (Stevens and Casey, 2010), which stay active even after a particular console exits. Unfortunately, wsl.exe does not leverage the same subsystem and therefore does not populate the data structures targeted by existing memory forensics algorithms.

4.2. Deficiencies targeted by existing research

To ensure that our research did not overlap with existing work, we compared the deficiencies found through our analysis with the code and works published by others.

As mentioned in section 2 the two main previous research efforts against WSL are the work done by Alex Ionescu and Michael Ligh. Combined, these covered the following deficiencies:

- The missing process names of WSL pico processes
- Recovery of command line arguments
- Locating the handle table of WSL pico processes, but not parsing the related file descriptors or referenced file paths and resources
- Enumeration of threads for WSL pico processes

The remaining deficiencies became the focus of our research effort.

5. Analyzing WSL memory artifacts

The primary focus of this section is the presentation of algorithms to recover forensic artifacts created by WSL application activity. The goal is to provide automated recovery, through the implementation of Volatility plugins, of userland and kernel space data structures utilized by WSL components.

For analysis, we collected memory samples from the Windows 10 x64 Version 1703 operating system with developers mode enabled and the Ubuntu WSL distribution installed. Volatility 2.6 was used for both initial memory analysis and plugin development. The `Win1064x_15063` Volatility profile already existed in Volatility 2.6 and matched the system version used for testing and research. Memory samples generated included instantiations of common Linux programs such as *top*, *man*, *ifconfig*, *iperf*, *python*, and */bin/bash* that were either currently running or that had terminated before collection.

We disabled developers mode and upgraded our system to the Fall Creators Update after it was released, then performed similar analysis on each Linux distribution using the `Win1064x_16299` Volatility profile. Our results are similar between versions except where noted in later subsections. The Linux distributions share a common pico provider, allowing our plugins to be distribution-agnostic.

5.1. Memory artifacts of a pico process

To determine if a process is a full NT process or a pico process, several members of the process structure (*_EPROCESS*) can be utilized. The following type information, derived from Volatility's `volshell` plugin, illustrates the relevant members:

```
>>> dt("_EPROCESS")
 '_EPROCESS' (2104 bytes)
...
0x300 : PicoCreated ['BitField',
        {'end_bit': 1, 'start_bit': 0,
         'native_type': 'unsigned long'}]
...
0x6cc : Minimal     ['BitField',
        {'end_bit': 1, 'start_bit': 0,
         'native_type': 'unsigned long'}]
...
0x710 : PicoContext ['pointer64',['void']]
...
```

For each pico process, the *Minimal* flag will be set as all pico processes are also minimal processes by definition. The *PicoCreated* flag will also be set for all pico processes and cleared for all NT processes.

The *PicoContext* pointer is cast as void because the structure of its corresponding object is defined by its process' pico provider, allowing the data structure to support the specific needs of each pico provider. Microsoft has not published any information regarding the structure of WSL *PicoContext* objects. Therefore, this structure must be reverse engineered in order to determine its layout before useful information can be extracted.

5.2. Initial binary analysis

The pico provider is the process that manipulates the *PicoContext* object. Therefore, reverse engineering the pico provider's executable can yield additional insight into the *PicoContext*'s structure. Our effort began with static analysis using IDA Pro (Hex-Rays.). First, we analyzed *lxcore.sys*, which is the executable that provides most of *lxss*'s functionality, to locate references to and within *PicoContext* objects. The *lxcore.sys* executable contains 2355 subroutines, of which about 3% are listed in *lxcore.sys*'s export table. The remaining subroutines are internal functions, which are unnamed.

Although only a small percentage of functions are purposely exported, we determined that the names of many internal subroutines can be extracted from the executable through cross-referencing the exported `LxpTraceLoggingBreakPoint` function, which is called when an exception occurs within the WSL pico provider. Among other tasks, it displays a name and a status string with details about the breakpoint. The status string often contains the name and return code of a recently-returned subroutine whose failure triggers the exception. Over 600 subroutines can be named by relying on this method. Fig. 3 shows an example of such a subroutine.

The names of subroutines often provide enough context to understand the general tasks the subroutines perform. Many relate to the virtual filesystem, networking stack, and threading support responsibilities of the pico provider. One design choice indicated by the naming scheme is the ownership of threads by the WSL pico provider, which is responsible for requesting resources from the kernel on behalf of the threads it supports. Threads that belong to a common pico process are organized into a *thread group* as indicated by the `LxpThreadGroup` prefix.

5.3. Structured enumeration of WSL pico processes

In his WinDbg scripts, Ionescu enumerates processes by following data structures linked from the global list of Linux Subsystem (LXSS) sessions. In our work, to preserve the existing work flow of Volatility plugins and avoid reliance on global data structures, we instead chose to enumerate pico processes by walking the well-known active process list.

Our `picolist` plugin, derived from the existing `pslist` Volatility plugin, enumerates active processes and outputs only the processes that match the following conditions:

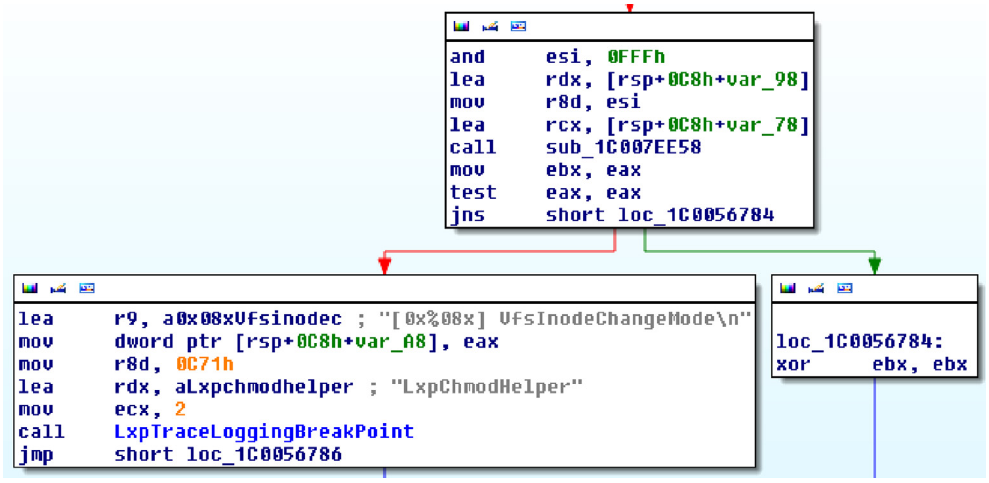
- *_EPROCESS.Minimal* $==$ *1*
- *_EPROCESS.PicoCreated* $==$ *1*
- *_EPROCESS.PicoContext* $\neq$ *null*

This filtering criteria ensures that our plugin 1) provides analysts with a quick method to determine if pico processes are present and 2) allows developers to inherit from our plugin to write analysis plugins targeted specifically at pico processes. Fig. 4 illustrates the output of the `picolist` plugin against a memory sample from our testbed. Each process is listed along with its full path inside of the Linux filesystem, its Windows and Linux PID, and its creation and termination time.

5.4. Scanning for WSL pico processes

WSL pico processes can also be identified via pool tag scanning (Schuster, 2008). The WSL pico provider allocates pool memory using the tag 0x4c782020 (“Lx” followed by two spaces) for many of the various types of objects it creates. The pool memory dedicated to *PicoContext* objects has a fixed size and is allocated in an unnamed internal function. Our new `picoscan` plugin uses the information related to this allocation, including the tag, size, and type, to locate PicoContext instances within the objects found through the existing `bigpagepools` Volatility plugin. Thus `picoscan` provides an alternate method of enumerating WSL pico processes without relying on `pslist`. Removing the reliance on `pslist` allows the potential discovery of processes that are hidden by malware.

Unfortunately, we have not developed a reliable method for finding metadata of terminated processes, like the existing Volatility plugin `psscan` does for NT processes. `psscan` successfully recovers terminated process metadata because the information it reports,

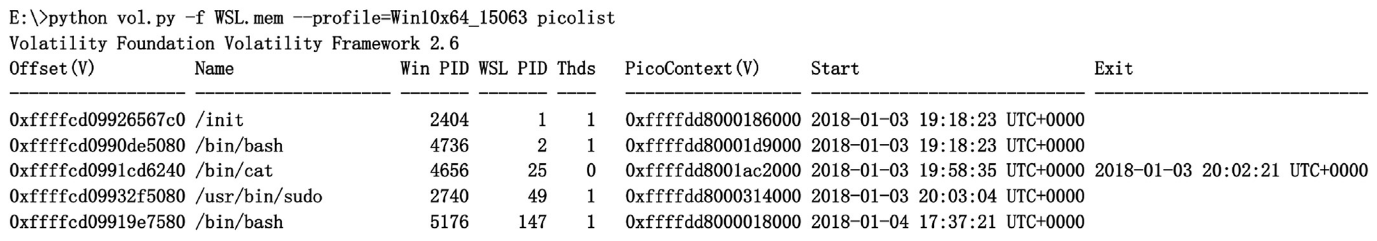


```
and     esi, 0FFFh
lea     rdx, [rsp+0C8h+var_98]
mov     r8d, esi
lea     rcx, [rsp+0C8h+var_78]
call    sub_1C007EE58
mov     ebx, eax
test    eax, eax
jns     short loc_1C0056784
```

```
lea     r9, a0x08xVfsinodec ; "[0x%08x] VfsInodeChangeMode\n"
mov     dword ptr [rsp+0C8h+var_A8], eax
mov     r8d, 0C71h
lea     rdx, aLxpchmodhelper ; "LxpChmodHelper"
mov     ecx, 2
call    LxpTraceLoggingBreakPoint
jmp     short loc_1C0056786
```

```
loc_1C0056784:
xor     ebx, ebx
```

Fig. 3. A sample call to `LxpTraceLoggingBreakPoint` from the Graph Overview display in IDA Pro for an unnamed subroutine within *lxcore.sys*. The *rdx* register points to the name of the calling subroutine. The *r9* register points to a template that allows the status code and name of a failed subroutine call to be printed.



```
E:\>python vol.py -f WSL.mem --profile=Win10x64_15063 picolist
Volatility Foundation Volatility Framework 2.6
Offset(V)          Name                  Win PID WSL PID Thds   PicoContext(V)     Start                          Exit
------------------ --------------------- ------- ------- ----   ------------------ ------------------------------ ------------------------------

0xffffcd09926567c0 /init                    2404       1    1   0xffffdd8000186000 2018-01-03 19:18:23 UTC+0000
0xffffcd0990de5080 /bin/bash                4736       2    1   0xffffdd80001d9000 2018-01-03 19:18:23 UTC+0000
0xffffcd0991cd6240 /bin/cat                 4656      25    0   0xffffdd8001ac2000 2018-01-03 19:58:35 UTC+0000 2018-01-03 20:02:21 UTC+0000
0xffffcd09932f5080 /usr/bin/sudo            2740      49    1   0xffffdd8000314000 2018-01-03 20:03:04 UTC+0000
0xffffcd09919e7580 /bin/bash                5176     147    1   0xffffdd8000018000 2018-01-04 17:37:21 UTC+0000
```

Fig. 4. Output of `picolist` plugin.

including the process' name, PID, parent PID, and starting and exit time are all stored directly within the *_EPROCESS* structure of the terminated process. Recovery of such information for WSL pico processes is generally much more difficult because instead of storing data directly within the process structure, much of the information is referenced through pointers. Once these pointers are freed, the memory regions they point to can be recycled by the memory manager. In the case of PicoContext objects, both the process' name and its corresponding *_EPROCESS* instance are stored outside of the structure. In rare cases, such as when analyzing a system that is very lightly used or when a sample is taken immediately after a process has terminated, these pointers may remain valid, but this is less likely to occur in real incident response scenarios.

5.5. *Enumerating threads from WSL pico processes*

Threads have also been impacted by Microsoft's implementation of pico processes. Each *_ETHREAD* contains a *PicoContext* pointer, which is set if the thread is owned by a pico process and null otherwise. The presence of a non-null value in this field indicates whether or not the thread is a *pico thread*. The *PicoContext* maintained by a pico thread is not the same as the *PicoContext* maintained by its owning process, but both contexts include attributes that identify their relationships with one another. A WSL pico process' *PicoContext* maintains both a list and a counter describing the *PicoContext* objects of its pico threads (Ionescu, 2016b). Each WSL pico thread's *PicoContext* contains a pointer to the *PicoContext* of its owning process as well as its corresponding *_ETHREAD*.

The values of non-pico fields are also affected. First, as shown in Fig. 1, pico threads do not have Thread Environment Blocks (TEBs). More importantly, some WSL pico threads maintain null *_ETHREAD.StartAddress* pointer values. Volatility's *_ETHREAD* class assumes that valid potential thread structures are invalid if these pointers are null. The `thrdscan` plugin only reports threads it considers valid, so many WSL pico threads are omitted from its output. At this time we do not know why some WSL pico threads do not have this field populated, but we suspect that it is instead stored somewhere in the thread's *PicoContext*.

Our new `picothreads` plugin enumerates the threads owned by WSL pico processes. This plugin could have referenced the thread list at *_EPROCESS.ThreadListHead*, but we chose to use the *PicoContext* list described above as it exposes an alternative method for discovering WSL pico threads. We also report per-process thread counters read from WSL pico process' *PicoContext* objects instead of their *_EPROCESS.ActiveThreads* values for similar reasons. The plugin's output is similar to `thrdscan` while also reporting each thread's *PicoContext*.

5.6. *Scanning for WSL pico threads*

Scanning for WSL pico threads is similar to scanning for WSL pico processes. *lxcore.sys's* `LxpThreadCreate` function uses a fixed-size pool memory allocation with a tag value of 0x4c782020 to create each thread's *PicoContext*. The function then populates the new context object with values and updates the process' *PicoContext* accordingly. We created a `picothrdscan` plugin to search for these objects and produce the same data that is output by `picothreads`. The plugin filters the output of `bigpools` using the known size and tag to produce a list of potential PicoContext objects belonging to WSL pico threads. The objects are validated if their corresponding *_ETHREAD* objects can be fetched and the *_ETHREAD.PicoContext* pointer matches the address of the scanned object.

Similar to WSL pico process' *PicoContext* objects, WSL pico threads' *PicoContext* memory allocations are susceptible to being overwritten shortly after being freed, compared to objects with flat structures. `picothrdscan` relies on the presence of several pointers to objects associated with the contexts to validate its findings. If any of these pointers are overwritten, then the thread cannot be found with this plugin. In these cases, the *_ETHREAD* class' validation function could be loosened to allow a null *StartAddress* pointer when the *PicoProcess* pointer is set, to increase the likelihood that terminated WSL pico threads are detected.

5.7. *Recovering process names*

As stated earlier, the *ImageFileName* field within *_EPROCESS* objects is not populated for WSL pico processes. The same is true for the *SeAuditProcessCreationInfo.ImageFileName* field. Given that these are the fields used to extract the names of NT processes, an alternative method of discovering WSL pico process names is required.

Analysis of the `LxpThreadGroupSetExecutable` function reveals that the WSL pico provider creates a Unicode string for each process that includes the full path to the process's executable. The path is relative to the user's *AppData\Local\Packages\<distro>\LocalState\rootfs* directory where the WSL files are located within the Windows filesystem. A pointer to this string is then stored in the process's *PicoContext* at a fixed offset. The correct process name is included in the `picolist` plugin's output and is provided to other WSL-related plugins.

The missing *ImageFileName* values likely result from a Windows kernel bug. A recent system patch to the Fall Creators Update enables the kernel to correctly populate these values (Ionescu, 2018). Our plugins continue to report WSL pico process names based on the values managed by the process' *PicoContext* objects in case further discrepancies between these values arise.

5.8. *Recovering process IDs*

A pico process' PID, as reported by Linux programs, is different from the PID tracked by Windows programs, such as *Task Manager*. This causes discrepancies when analyzing log files and other runtime data from Linux programs and cross-referencing PIDs reported by Volatility's `pslist` plugin. The same issue would arise if a live response tool was used to generate a list of running processes from a Windows program.

Accessing the Linux PID of a WSL pico process first requires dereferencing a pointer within the process' *PicoContext* to an undocumented object. A *uint32_t* field at a fixed offset within this object stores the Linux PID. All of our developed plugins report both the Windows PID, stored at *_EPROCESS.UniqueProcessId* as well as the Linux PID. A full analysis of the undocumented object remains the subject of future work.

5.9. Building parent/child process mappings

Based on our analysis, the Linux parent PID of a WSL pico process does not appear to be stored anywhere within the *PicoContext* object. However, the parent/child relationship can still be reconstructed as each *PicoContext* contains a pointer to its parent process' *PicoContext* object. The lone exception to this rule is/*init*, which instead has a correct *_EPROCESS.InheritedFromUniqueProcessId* value. Our new `picotree` plugin is a modified version of Volatility's `pstree` plugin that uses PicoContext addresses to correctly create the inheritance tree of WSL pico processes.

5.10. Enumerating environment variables

Whereas NT processes store environment variables in a block of memory pointed to by their PEBs' *ProcessParameters.Environment* field, WSL pico processes do not have PEBs and instead store environment variables at addresses tracked by a field at a fixed offset within their *PicoContext* objects. The environment variable names and values are stored in a contiguous set of character strings whose total length is specified by a *size_t* field at an adjacent *PicoContext* offset. The WSL pico provider writes this information in one of two internal functions — `LxpThreadGroupSetupUser` or `LxpThreadGroupCreate`. Automated recovery of WSL pico process environment variables is included in the output of our `picoenvars` plugin.

5.11. Locating the process executable

For an NT process, the base address of the application executable can be found by referencing the *ImageBaseAddress* member of the process' PEB. The ability to determine where an application is loaded into process memory enables several key memory forensic capabilities, such as:

- Extraction of the running executable of a process, as implemented in Volatility's `procdump` plugin
- Extraction of in-memory, unpacked malicious code (Ligh et al., 2014)
- Detecting process hollowing techniques (Monnappa, 2016)
- Automating Yara and other signature-based scans across running processes (Case, 2016)
- Reconstructing API usage to aid reverse engineering (Reverse Engineering Rootkits, 2014)

The inability to access this information for pico processes breaks all of the capabilities listed above, among others. Fortunately, the load address can be recovered through analysis of the *PicoContext* object. Specifically, by following two undocumented pointers that are written to the context object in the WSL pico provider's `LxpThreadGroupSetupUser` internal function, a pointer to the application's ELF program headers can be obtained. This metadata can then be used to determine the initial load address, which is accessible via our plugin API. Access to the executable load address can be used to restore all of the previously listed capabilities except for those related to extraction, which is covered in Section 5.13.

5.12. Locating shared libraries

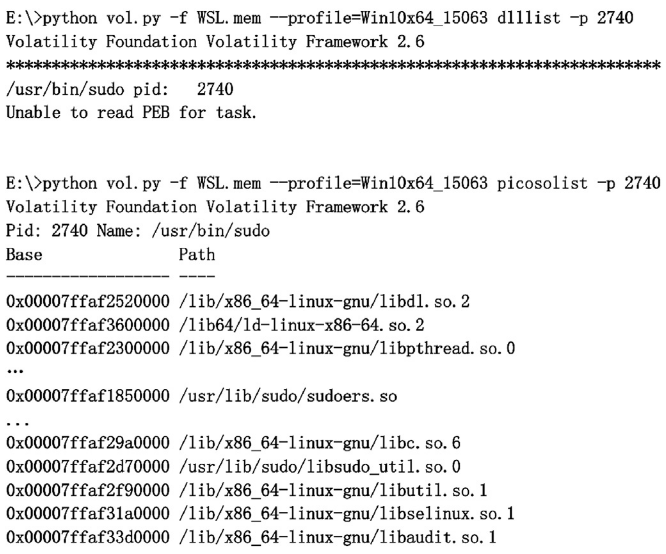
The lack of a PEB also prevents Volatility's `dlllist` and `ldrmodules` plugins from enumerating shared libraries associated with a WSL pico process. This presents many of the same issues as the inability to locate where the application executable resides in memory.

To recover this information, Volatility's existing algorithm for enumerating shared libraries from Linux's runtime loader, as implemented in the existing `linux_library_list` plugin, was ported to target WSL pico processes. Fortunately, it appears that Microsoft did not substantially change the algorithm used by the runtime loader as the existing algorithm was able to successfully recover all shared libraries and their metadata. Fig. 5 contrasts the output of `dlllist`, which attempts to use a process' PEB, with the output of our new `picosolist` plugin. By leveraging our new `picosolist` Volatility plugin, analysts can now determine which shared libraries are loaded by WSL Linux processes.

5.13. Extracting Linux executables

Once a process' executable and shared libraries are located in process memory, analysts may then want to extract them from memory. For NT processes, Volatility provides the `procdump` and `dlldump` plugins for this purpose. Unfortunately, neither of these plugins work properly for WSL pico processes, because they handle only Windows PE format executables and not Linux's ELF format.

To remedy this issue, we created two new Volatility plugins, `picoelflist` and `picoelfdump`. The `picoelflist` plugin is similar to the existing `linux_elfs plugin`, which enumerates all ELF files mapped into processes on Linux systems. To gather the list of ELF files in a WSL pico process, `picoelflist` walks the process' VAD tree (Dolan-Gavitt, 2007) and focuses on VAD nodes with `PAGE_EXECUTE_WRITECOPY` protection and an associated FileObject pointer. It then verifies that each matching memory region begins with a valid ELF header. By default, our `picoelfdump` plugin relies on `picoelflist` to find and properly extract all loaded



```
E:\>python vol.py -f WSL.mem --profile=Win10x64_15063 dlllist -p 2740
Volatility Foundation Volatility Framework 2.6
************************************************************************
/usr/bin/sudo pid:   2740
Unable to read PEB for task.


E:\>python vol.py -f WSL.mem --profile=Win10x64_15063 picosolist -p 2740
Volatility Foundation Volatility Framework 2.6
Pid: 2740 Name: /usr/bin/sudo
Base               Path
------------------ ----
0x00007ffaf2520000 /lib/x86_64-linux-gnu/libdl.so.2
0x00007ffaf3600000 /lib64/ld-linux-x86-64.so.2
0x00007ffaf2300000 /lib/x86_64-linux-gnu/libpthread.so.0
...
0x00007ffaf1850000 /usr/lib/sudo/sudoers.so
...
0x00007ffaf29a0000 /lib/x86_64-linux-gnu/libc.so.6
0x00007ffaf2d70000 /usr/lib/sudo/libsudo_util.so.0
0x00007ffaf2f90000 /lib/x86_64-linux-gnu/libutil.so.1
0x00007ffaf31a0000 /lib/x86_64-linux-gnu/libselinux.so.1
0x00007ffaf33d0000 /lib/x86_64-linux-gnu/libaudit.so.1
```

Fig. 5. Output of `dlllist` compared to output of the new `picosolist` plugin.

ELF executables. Extraction is performed through Volatility's existing ELF parsing and extraction API.

To replicate the functionality of Volatility's existing procdump and dlldump plugins, we also created two new plugins, picoprocdump and picosodump. These new plugins leverage our ability to find the load address of a process as well as its associated libraries to find executables to extract. Furthermore, we combined the logic of picoelflist, picoscan, and picosolist to implement a new picoldrmodules plugin. This new plugin replicates the malware-finding algorithm of the existing ldrmodules plugin, so WSL pico processes can be scrutinized in the same way.

5.14. Enumerating file system handles

Since the handles tables for WSL pico processes are not maintained, the existing Volatility handles plugin is unable to enumerate which system resources a WSL pico process is currently utilizing. To address this issue, we created the picolsof plugin to provide capabilities similar to those of the existing Volatility linux_lsof plugin. Creation of this plugin required careful reverse engineering of the *lxcore.sys* driver to answer several key questions, including how:

- a process' file descriptor table is linked to its *PicoContext*
- file descriptors are stored within the table
- to recursively recover the full path of files associated with descriptors

The most useful functions to analyze for recovering this information include LxpThreadCleanup for linking a file descriptor table to its PicoContext, LxpFileReferenceByDescriptor for descriptor enumeration, and VfsDirectoryEntryGetPathHelper for mapping opened file paths. After a thorough analysis of these functions, plus a few related helper functions, we developed the picolsof plugin, which reports the opened file descriptors associated with WSL pico processes. Sample output from this plugin appears in Fig. 6.

5.15. Command history recovery - LXSS

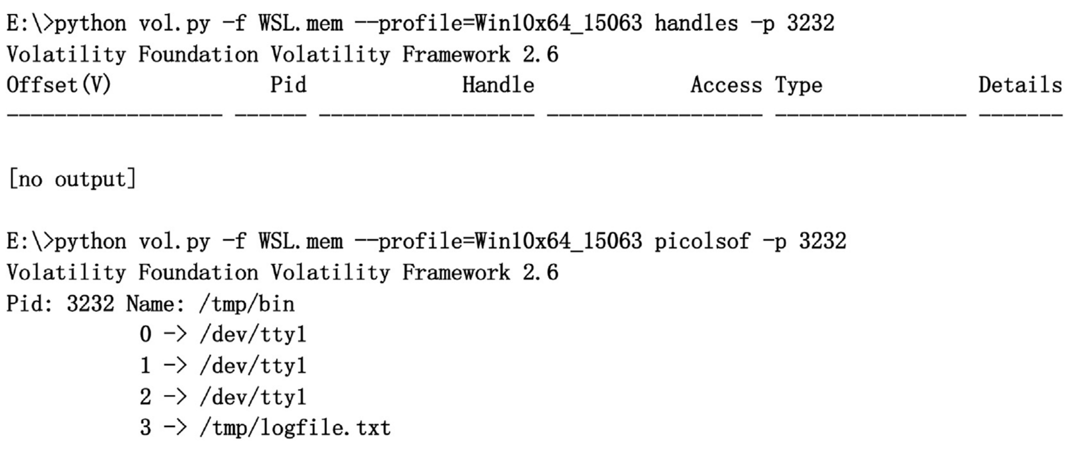
Since WSL is largely driven by command line activity through *wsl.exe*, a natural artifact of interest is the list of commands entered into the consoles as well as any resulting console output. For this reason, we investigated why Volatility's cmdscan and consoles plugins were unable to produce console input and output related to WSL *wsl.exe* sessions. These plugins operate by scanning instances of the *Client/Server Runtime System* (Client/Server Runtime Subsystem, 2017a; Windows 7/Windows Server, 2008 R2, 2017b) for input and output generated by console programs such as *cmd.exe* and *powershell.exe*.

Our inspection of the parent/child relationship of processes related to WSL showed that each *wsl.exe* instance spawned an associated *conhost.exe*, which is the server component of the client/server runtime system. When testing the plugins against various memory samples, we noticed that the plugins not only missed activity related to *wsl.exe*, but also the traditional command shell and PowerShell consoles. At this point, we contacted a Volatility developer, who informed us that these plugins do not currently support Windows 8 and Windows 10.

Analysis of *conhost.exe* from our Windows 10 test system showed that it was using a new background implementation contained within *conhostV2.dll*. This DLL and its implementation do not appear before Windows 8. Binary analysis of this DLL showed that several of the structure members and algorithms expected by the Volatility plugins were drastically changed. We then updated the cmdscan plugin to support Windows 10. After our updates, the plugins successfully recovered commands executed in the traditional consoles, but were still unable to produce results related to *wsl.exe* activity. Next, we performed binary analysis of *wsl.exe* and *cmd.exe*, to better understand their interactions with *conhost.exe*. This research was not conclusive, but our current assessment is that wsl.exe does not leverage the same APIs as the traditional consoles, which prevents the data structures from being created and populated on the server side of the activity. This research is still ongoing.

5.16. Command history recovery - bash

Beyond the interaction between *wsl.exe* and *conhost.exe*, we also investigated recovery of command activity directly from */bin/bash*. This yielded much better results. To investigate this theory, we created a version of the existing linux_bash Volatility plugin designed to target WSL */bin/bash* processes. Fortunately, Microsoft seems to have leveraged the same code, or at least the same data structures, as the familiar Linux bash console. This allows use of the existing bash history recovery algorithm for WSL processes and this is implemented by our new picobash plugin. As illustrated in Fig. 7, the plugin successfully recovers executed commands as well as the execution times for commands entered in the current sessions. Commands from previous sessions that were saved to disk will have starting times matching the time the shell process was created.



```
E:\>python vol.py -f WSL.mem --profile=Win10x64_15063 handles -p 3232
Volatility Foundation Volatility Framework 2.6
Offset(V)           Pid        Handle              Access Type              Details
------------------ ------ ------------------ ------------------ ---------------- -------

[no output]

E:\>python vol.py -f WSL.mem --profile=Win10x64_15063 picolsof -p 3232
Volatility Foundation Volatility Framework 2.6
Pid: 3232 Name: /tmp/bin
          0 -> /dev/tty1
          1 -> /dev/tty1
          2 -> /dev/tty1
          3 -> /tmp/logfile.txt
```

Fig. 6. Output of the new picolsof plugin, enumerating file handles associated with a pico process.

```
E:\>python volatility\vol.py -f samples\WSL.vmem --profile=Win10x64_15063 picobash
Volatility Foundation Volatility Framework 2.6
Win PID WSL PID Command Time                    Command
------- ------- ------------------------------ -------
   5472       2 2018-01-05 22:40:00 UTC+0000   uname -a
   5472       2 2018-01-05 22:40:02 UTC+0000   ls -ltr
   5472       2 2018-01-05 22:40:11 UTC+0000   ifconfig
   5472       2 2018-01-05 22:40:21 UTC+0000   echo $PATH
   5472       2 2018-01-05 22:40:27 UTC+0000   which iperf
   5472       2 2018-01-05 22:40:32 UTC+0000   top
    716      17 2018-01-05 22:40:56 UTC+0000   iperf -s
   3160      27 2018-01-05 22:41:01 UTC+0000   iperf -c 127.0.0.1
   3160      27 2018-01-05 22:41:17 UTC+0000   wget https://bootstrap.pypa.io/get-pip.py
   3160      27 2018-01-05 22:41:27 UTC+0000   sudo -H python get-pip.py
```

Fig. 7. Output of the new `picobash` plugin. This output indicates that three */bin/bash* pico processes were instantiated and displays the commands executed within them.

6. Conclusions and future work

In this paper we have detailed many new memory forensic algorithms and Volatility plugins that enable deep analysis of Microsoft's Windows Subsystem for Linux. Combined, these plugins fix the many deficiencies we discovered when existing memory forensic algorithms are applied to WSL pico processes. By using our new plugins, analysts will now have the same capabilities for analyzing WSL pico processes as they do for traditional Windows processes.

While the results of this research led to many new analysis capabilities for WSL, much remains to be done. For example, our `picohandles` plugin can enumerate the WSL-specific file system cache to match a particular file descriptor, but no other plugin separately inspects the entire cached file system or fully enumerates it. Furthermore, while the existing Volatility `netscan` plugin can recover networking artifacts from WSL application network activity, our time in reversing *lxcore.sys* revealed that there is a substantial networking stack layer implemented inside of lxss. So far, none of the artifacts from this network layer have been examined in detail. Finally, there are entire capabilities of *lxss*, such as IPC between WSL's pico and NT processes, that remain unexplored.

Beyond simply enumerating artifacts from these undocumented subsystems, the size of the *lxss* executables, along with the substantial functionality they implement, suggests that there are likely many new ways for malware to interfere with and hide from inspection. Discovering weaknesses in these subsystems that can be abused by malware will require a larger research effort to uncover and document. Furthermore, existing Volatility plugins that hunt for Linux userland malware, such as `linux_apihooks` and `linux_plthook` have yet to be ported to WSL. We are currently working with the Volatility developers to accomplish this goal in an efficient manner.

Finally, our current research effort for WSL has focused on two versions of Windows 10. As shown in Ligh's work on porting the recovery of WSL process names to all versions of Windows 10, the offsets inside of key data structures can and likely will change. We experienced such changes when upgrading from Build 1703 to the Fall Creators Update. To make porting our many new plugins to new versions of Windows 10 easier, we have documented the function(s) inside of the *lxss* executables that reference needed offsets. This allows quickly determining which offsets are appropriate when the offsets change.

References

Baumann, Z., Zill, B., Galen, H., Lorch, J., Olinsky, R., 2016. Drawbridge. https://www.microsoft.com/en-us/research/project/drawbridge/.

Case, A., 2016. Automating Detection of Known Malware through Memory Forensics. https://volatility-labs.blogspot.com/2016/08/automating-detection-of-known-malware.html.

Client/Server Runtime Subsystem, 2017a. https://en.wikipedia.org/wiki/Client/Server_Runtime_Subsystem.

Cooley, S., 2017. Command Reference. Windows Subsystem for Linux. https://msdn.microsoft.com/en-us/commandline/wsl/reference/.

Cooley, S., Hanselman, S., McBee, A., Kottmann, R., Nikoli, A., 2017. Windows 10 Installation Guide. https://docs.microsoft.com/en-us/windows/wsl/install-win10/.

Cygwin, F.A.Q., 2017. https://cygwin.com.

Dolan-Gavitt, B., 2007. The VAD tree: a process-eye view of physical memory. In: Proceedings of the 2007 Digital Forensic Research Workshop.

Elbaz, G., Atias, D., 2017. Beware of the Bashware: a New Method for Any Malware to Bypass Security Solutions. https://research.checkpoint.com/beware-bashware-new-method-malware-bypass-security-solutions/.

Hammons, J., 2016a. Pico Process Overview. Windows Subsystem for Linux. In: https://blogs.msdn.microsoft.com/wsl/2016/05/23/pico-process-overview/.

Hammons, J., 2016b. Windows Subsystem for Linux Overview. https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-overview/.

Hex-Rays IDA Pro Disassembler. http://www.hex-rays.com/products/ida/index.shtml.

Hron, M., 2017. PICO Processes Toolbox, a Playground for PICO Processes Research. https://github.com/thinkcz/pico-toolbox/.

Ionescu, A., 2016a. The Linux Kernel Hidden inside Windows 10. Blackhat.

Ionescu, A., 2016b. Fun with the Windows Subsystem for Linux (WSL/LXSS). https://github.com/ionescu007/lxss/.

Ionescu, A., 2018. https://twitter.com/aionescu/status/971510784672092161.

Ligh, M., 2017. Patches to Volatility to Correctly Parse Pico Process Names. https://github.com/volatilityfoundation/volatility/commit/6d24b05f86ecb854ef0994933d4baa4d68171b24Ω.

Ligh, M., Case, A., Levy, J., Walters, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. Wiley, New York.

Monnappa, K.A., 2016. Detecting Deceptive Hollowing Techniques. https://cysinfo.com/detecting-deceptive-hollowing-techniques/.

MSDN, 2017. Windows Command Line Tools for Developers. https://blogs.msdn.microsoft.com/commandline/.

Porter, D.E., Boyd-Wickizer, S., Howell, J., Olinsky, R., Hunt, G.C., 2011. Rethinking the library OS from the top down. ACM SIGPLAN Notices, 46 (3), 291–304.

Raj, T., 2017. Windows 10 Fall Creators Update. https://blogs.msdn.microsoft.com/commandline/2017/10/11/whats-new-in-wsl-in-windows-10-fall-creators-update/.

Reverse Engineering Rootkits, 2014. https://www.youtube.com/watch?v=LVJ5mpZZdY4.

Schuster, A., 2008. The impact of microsoft windows pool allocation strategies on memory forensics. In: Proceedings of the 2008 Digital Forensic Research Workshop.

Stevens, R.M., Casey, E., 2010. Extracting windows command line details from physical memory. Digit. Invest. 7, S57–S63.

The Volatility Framework, 2017. Volatile Memory Artifact Extraction Utility Framework. https://github.com/volatilityfoundation/volatility.

The Windows Subsystem for Linux, 2017. https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux.

Turner, R., 2017. Windows Subsystem for Linux Out of Beta! Windows Command Line Tools for Developers. https://blogs.msdn.microsoft.com/commandline/2017/07/28/windows-subsystem-for-linux-out-of-beta/.

Windows 7/Windows Server 2008 R2, 2017b. Console Host. https://blogs.technet.microsoft.com/askperf/2009/10/05/windows-7-windows-server-2008-r2-console-host/.

Yosifovich, P., Ionescu, A., Russinovich, M., Solomon, D., 2017. Windows Internals, Part 1, seventh ed. Microsoft Press.