Forensic Science International: Digital Investigation 44 (2023) 301505

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2023 EU - Selected papers of the Tenth Annual DFRWS Europe Conference

Module extraction and DLL hijacking detection via single or multiple memory dumps

Pedro Fernández-Álvarez, Ricardo J. Rodríguez[*]

Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain

ARTICLE INFO

Article history:

Keywords:
Digital forensics
Memory forensics
Volatility
Windows
DLL hijacking

ABSTRACT

A *memory dump* contains the current state of a system's physical memory at the time of its acquisition. Among other things, it contains the processes that were running at the time of acquisition. These processes can share certain functionalities provided by shared object files, which are internally represented by modules in Windows. However, each process only maps in its address space the functionalities it needs, and not the entire shared object file. In this way, the current tools for extracting modules from existing processes in a memory dump from a Windows system obtain the partial content of a shared object file instead of the entire file. In this paper we present two tools, dubbed `Modex` and `Intermodex`, which are built on top of the Volatility 3 framework. These tools allow a forensic analyst to extract a 64-bit module from one or more Windows memory dumps as completely as possible. To achieve this, they aggregate the contents of the same module loaded by multiple processes that were running in the same memory dump or in different dumps (we called it *intradump* and *interdump*, respectively). Additionally, we also show how our developed tools are useful to detect *dynamic-link library* (DLL) *hijacking* attacks, a widely used attack on Windows where attackers trick processes into loading a malicious DLL instead of the benign one.

1. Introduction

Memory forensics is a subfield of digital forensics (Wu et al., 2020) that consists of collecting a snapshot of the system in its current state (called *memory dump*), which is then analyzed with appropriate tools (such as Volatility (Ligh et al., 2014), Rekall (Rekall, 2014), or Helix3, to name a few). A complete survey of state-of-the-art memory acquisition techniques is given in (Latzo et al., 2019). This type of forensics is useful in scenarios where encrypted storage, encrypted network traffic, or even no activity traces are written to disk. Storage-based forensics may be impracticable or useless in these scenarios.

A memory dump contains relevant data (called *memory artifacts*) for incident analysis, comprising elements such as the running processes, open files, logged users, or open network connections at the time of memory acquisition. A memory dump can also contain many artifacts that are more likely to reside in memory rather than in disk, due to their volatile nature, such as as residual IP packets, Ethernet frames, and other associated data structures (Beverly et al., 2011).

Much modern malware performs *memory-only* attacks, where no activity traces or malware footprints are left on disk (Oosthoek and Doerr, 2019; Manna et al., 2022). Traditional storage-based forensics are useless in detecting these types of threats, which can only be analyzed using memory forensics. In this work, we focus on Windows, since today it is still the predominant operating system that is abused by programs with malicious intent (AV-TEST, 2022).

In essence, Windows malware are pieces of software that primarily rely on Windows APIs to interact with the Windows operating system. For Windows operating systems, much of the operating system functionality accessible through Windows APIs is provided by *dynamic-link libraries* (DLL) files.

A DLL is a module that contains functions and data that can be used by other modules (Microsoft, 2022a). In Windows terminology, a *module* is an executable file or DLL (Microsoft, 2021b). When a Windows program is running, much of its functionality may be provided by one or more modules. For example, some programs may contain many different modules, and each program module is contained and distributed in DLL files. Using DLLs helps promote code modularization, code reuse, and efficient memory usage, among other benefits.

Multiple processes can load the same module into memory when the functionality provided by such a module is needed. Due

[*] Corresponding author.
E-mail address: rjrodriguez@unizar.es (R.J. Rodríguez).

https://doi.org/10.1016/j.fsidi.2023.301505

to memory usage optimization techniques, not all processes will map the entire module or even the same parts of the module in their address space: the pages (fixed-length, contiguous blocks of virtual memory) mapped for a given process are those accessed by the process. Therefore, current tools for extracting modules from memory dumps (such as the Volatility 3 plugin `windows.modules.Modules` with the `--dump` parameter) are imprecise, as only the mapped pages into a single process address space are extracted.

Taking this shortcoming into account, we have developed tools to get as much content as possible from a given module. In particular, our goal is to extract a module as complete as possible from a single memory dump (we called it *intradump extraction*) or from several memory dumps (*interdump extraction*). This extraction is particularly useful when a module is loaded by more than one process on the same machine, or by multiple processes running on different machines, as it allows a forensic analyst to get the running module as complete as possible even when the content of the device is encrypted. Note that although we focus on modules from 64-bit DLL files in this paper, our tools are also valid for extracting modules from 64-bit executable files.

This scenario is particularly relevant when the forensic analyst wants to analyze whether a DLL module is malicious or not. In the case of DLLs, a common Windows mechanism that is abused by attackers is the way that some Windows applications look for DLLs. Using a *DLL hijacking* attack (MITRE ATT&CK, 2021), attackers exploit this mechanism to force an application to load malicious code (contained in a DLL with the same name as the non-malicious DLL) into the application's address space. Malware typically uses this attack to execute malicious payloads, escalate privileges, or gain persistence (Uroz and Rodríguez, 2019), among other goals. Our module extraction tools perform a variety of checks to detect this attack.

The contribution of this paper is twofold. First, we have developed two tools, `Modex` and `Intermodex`, to do intradump or interdump extraction, respectively. Both tools are written in Python 3 and their source code is freely available as open source under the GNU/GPLv3 license. In particular, `Modex` is a Volatility 3 plugin, while `Intermodex` is a standalone tool that relies on `Modex` for its operation. Second, we have integrated the ability to detect DLL hijacking attacks as a feature in both tools. We also present the experimental evaluation of the tools to validate them.

The rest of the paper is organized as follows. Section 2 goes over some background related to (virtual) memory management, Windows modules, and DLL hijacking attacks. Section 3 discusses related work, while Section 4 describes our `Modex` and `Intermodex` tools. Section 5 presents the experiments and limitations of our intradump and interdump extraction approach. The proof of concept for detecting DLL hijacking attacks is given in Section 6. Finally, Section 7 concludes the paper and sets out future work.

2. Background

This section explains concepts about Windows memory management and hijacking attacks that are important to a better understanding of this paper.

2.1. Paging and the virtual memory manager

Each Windows process has its own private virtual address space (Ligh et al., 2014), which is a linear memory space (i.e., with contiguous addresses) divided into blocks of the same length, called pages. A *page* of a process's virtual address space can be in different states (Microsoft, 2021c) (free, reserved, or committed) and its size can be small (4 KiB, on x86 architectures) or large (ranging from 2 to 4 MiB, on x64 and ARM architectures) (Yosifovich et al., 2017).

Windows maintains the relationship between virtual memory and physical memory through *Page Table Entries* (PTEs), which map a process virtual memory page to a physical memory page. With PTEs, the virtual memory manager (a Windows kernel process) keeps track of the virtual addresses of reserved or committed pages through the *Virtual Address Descriptor* (VAD) tree (Yosifovich et al., 2017; Dolan-Gavitt, 2007). In addition, it ensures that when a thread (in the context of a process) reads or writes to addresses in its virtual memory space, refers to the correct physical addresses (Microsoft, 2021a).

A *shared page* is a page accessed by multiple processes, while a *private page* for a given process is a page accessed only by that process. Unlike private pages, shared pages are stored only once in physical memory, and all processes that share a given page access the same physical address when reading information from that page through their own virtual addresses. To make this possible, Windows uses *prototype PTEs* (Yosifovich et al., 2017), a special type of PTE that enables shared memory support in Windows and that are not stored in process page tables (called *real* or *process PTEs*).

When a process wants to modify content present in a shared page, a new private page is created for that process to reflect the modification, leaving the shared page unchanged. This mechanism, called *copy-on-write*, prevents modifications to a shared page from being visible to all processes sharing that page. In this way, this modification only affects the process that performs it.

2.2. Page frame number database

The *page frame number database* (PFN DB) is a Windows kernel data structure that describes each page stored in physical memory (Yosifovich et al., 2017). This database is, in effect, an array of elements (called *PFN DB entries*) that describe the state of a single physical page.

Among other fields, a PFN DB entry has a field called `PteAddress` that contains the virtual address of the PTE that points to the page represented by that PFN DB entry. This `PteAddress` points to a prototype PTE or a real PTE depending on whether the page represented by a PFN DB entry is shared or private. In addition, each PFN DB entry has a flag named `PrototypePTE`, which is used to indicate whether the PTE referenced by that PFN DB entry is a prototype PTE or not. This indicator is useful to differentiate between shared pages and private pages. Therefore, it is necessary to parse the PFN DB to distinguish between PTE types.

2.3. Windows modules

On Windows, an *image* is any executable, shared dynamic library, or driver file loaded as part of the kernel or a user-mode process, while an *image file* is the file as in disk. Internally, an image and a process are represented by a module (Microsoft, 2021b). In what follows, we adhere to this terminology.

A Windows image file follows the *Portable Executable* (PE) format (Microsoft, 2022c), which encapsulates the information necessary for the Windows PE loader to manage the executable code. When an image file runs, the Windows PE loader creates a virtual address space for the process and maps the image file from disk to the process address space. It tries to load the image at its preferred base address (defined in a PE field) and maps the PE sections into memory. During this mapping, multiple pages (typically small pages) are allocated to accommodate the content of the image file. In addition to the memory required to hold the PE sections, more memory is allocated to hold the process stack and heap. In addition, external dynamic shared libraries on which the program depends are also loaded into the same memory space in a similar way, allocating memory appropriately.

When a module's preferred base address is already occupied, the Windows PE loader relocates the module appropriately, setting references to certain memory addresses in the module's code if necessary. Apart from this *relocation process*, the address space of the stack, the heap, and external dynamic shared libraries are randomized (on every Windows boot) thanks to *Address Space Layout Randomization* (ASLR), a software security mechanism to prevent certain memory corruption vulnerabilities.

2.4. Hijacking execution flow attacks

Hijacking execution flow attacks can be used for the purposes of persistence (Uroz and Rodríguez, 2019), escalating privileges, or hiding malicious actions behind a legitimate process. According to MITRE (MITRE ATT&CK, 2021), there are two techniques that adversaries can use in Windows to execute their own malicious payloads by hijacking the way operating systems search and load DLLs.

One of these techniques is *DLL search order hijacking* (MITRE ATT&CK, 2022), where adversaries take advantage of the Windows DLL search order (Microsoft, 2022b) to make a particular program load a malicious DLL that has the same filename as the legitimate DLL, but different content. A similar technique is *DLL side-loading*, where adversaries run the legitimate program themselves to force the loading of a malicious DLL.

A program can import one or more functions from a DLL, which implement the functionality required by the application. Therefore, the malicious DLL must not only have the same filename as the legitimate one, but also the same exported function names. In addition, they must also provide the expected implementation of these exported functions so that the program can work as usual. *DLL proxying*, which consists of simply calling the functions of the legitimate DLL instead of implementing the exported functions, allows an attacker to do this. As a result, the malicious DLL acts as a proxy between the program and the legitimate DLL. A well-known malware that uses this technique is *Stuxnet* (Langner, 2011).

3. Related work

Memory forensics has been considered as the basis for malware analysis in many works, although this approach is not without its problems (Martín-Pérez and Rodríguez, 2021). Cohen (2017) discusses the effective use of YARA (a very popular tool for identifying and classifying malware samples) to scan for malware in memory. Other approaches are based on machine learning. For instance, Aghaeikheirabady et al. (2014) use information from various sources such as VADs (Dolan-Gavitt, 2007), registry hives, and other internal process structures to detect malware. Also, the work in (Mosli et al., 2016) uses features (registry keys, DLLs, and the called operating system functions, among others) extracted from the reports of Cuckoo Sandbox. Bozkir et al. (2021) use a different approach, combining machine learning and image processing for malware classification of Windows processes extracted from memory.

The prevalence of certain DLLs in processes contained in a memory dump is used as a characteristic of the malicious behavior in Duan et al. (2015). The detection of DLL hijacking attacks that we perform in this paper is similar to the work in (Case et al., 2020), where a Volatility plugin called `hooktracer_messagehooks` is introduced that helps analyze hooks in a Windows memory dump. This plugin helps determine if hooks are associated with malicious keylogger or benign software. Our tools are complementary to this plugin and can be combined to provide better memory dump analysis.

With regard to malware focused on hiding its presence, Block and Dewald (2019) developed an approach to detect different code injection techniques despite the use of stealthy techniques. Instead of VADs, this approach relies on information stored in the PFN DB to avoid certain (advanced) stealthy techniques. Likewise, Balzarotti et al. (2015) present different techniques malware can adopt to hide its presence using GPU memory. The analysis of memory other than the physical memory, though, is beyond the scope of this paper.

4. `Modex` and `Intermodex`

In this section we describe the tools developed in this work, `Modex` and `Intermodex`. Both tools are publicly and freely available under the GNU/GPLv3 license (Fernández-Álvarez and Rodríguez, 2022) to foster research in the field of memory forensics. Fig. 1 shows a high-level diagram of both tools and their relationships to each other and to Volatility 3.

4.1. `Modex`

`Modex` is a Volatility 3 plugin implemented in Python 3 and designed to extract a 64-bit Windows module from a memory dump as completely as possible (i.e., it makes an intradump extraction). We chose Volatility because it is the most widely used framework for extracting digital artifacts from volatile memory (Volatility Foundation, 2022).

This plugin takes the memory dump path and the name of a module as arguments and outputs a directory containing several files: a file with the extension *.dmp* (which corresponds to the extracted module with as many pages as possible), a JSON file containing metadata about the extracted module, and a log file with information about the execution of `Modex`. Currently, our tool only extracts 64-bit modules. The JSON file contains, among other data, all the processes where the extracted module was loaded, and for each page extracted it includes its offset and whether it was a shared or private page.

The extraction of a module is as follows. `Modex` walks through all the processes in the memory dump and checks which one loaded the given module as an argument. When loaded into a process, the module is dumped, getting as many (*intermediate*) *.dmp* files as there are processes where the module was loaded.

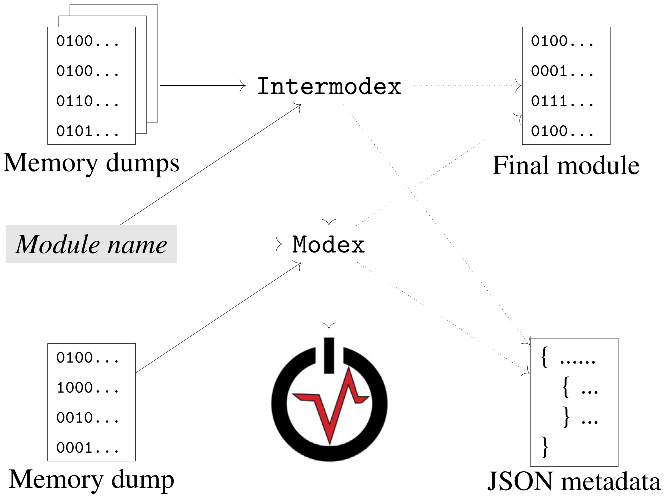


Fig. 1. High-level diagram of `Modex` and `Intermodex`. Solid arrows indicate the tool inputs, while dashed and dotted arrows indicate tool dependencies and outputs, respectively.

Finally, it combines all those intermediate dumped modules into a single *.dmp* file (we called it the *final module*). Below, we explain each step in detail.

We use the DllList plugin to dump a module that is loaded in a given process.[1] This plugin only dumps the module pages that are assigned in that process, creating what we call an intermediate *.dmp* file. To combine these *.dmp* files, we need to know which pages corresponding to the module were allocated in each process in which the module was loaded, and whether each of those pages is shared or private. To do this, we rely on its *PrototypePTE* flag (see Section 2.2). To help us get this information, we use the plugin SimplePteEnumerator (Block, 2021), which provides the PTE and the entries in the PFN DB of pages in a process that fall within a certain range of virtual addresses. After collecting this information, we can aggregate all dumped modules to get just one. In the current version of Modex, modules must have the same path, the same size, and must all be loaded at the same base address in all processes to be combined.

We can retrieve zero, one, or multiple pages at each module offset. When no pages are retrieved, the final module will have that page filled with zeroes. When only one page is retrieved, that page is the one that is put into the final module. However, when multiple pages are found at the same offset, only one of them can be placed in the final module. Therefore, we have to decide which page to choose among the candidates. At a given offset, we distinguish three different cases:

All pages are shared. In this case, we choose one of them at random, since they all have *theoretically* the same content. In our experiments we have found that this is not always true and that sometimes (rarely) there are shared pages with small differences in their content. We discuss this in more detail in Section 5.2.

Some pages are private and some are shared. Here, we discard the private pages and consider only the shared pages, choosing one of them at random as in the previous case. We prioritize the shared pages over the private ones since we want to retrieve the contents that are most similar to those stored on the disk.

All pages are private. In this case, we choose the page that most closely resembles the shared page corresponding to that offset (i.e., the one with the fewest modifications). To do this, we create a similarity matrix of the pages as a square matrix of dimension the number of pages to compare. To populate it, we calculate a similarity score between every two pages to reflect how similar their content is. In particular, we use TLSH (Oliver, 2021) as the similarity digest algorithm (Breitinger et al., 2014). In TLSH, the more similar two pages are, the closer the similarity score is to zero (i.e., its score trend is descending (Martín-Pérez et al., 2021b)). We then add the values in each row and choose the page corresponding to the row with the lowest value as the page to include in the final module.

4.2. Intermodex

Intermodex is a tool written in Python 3 also with the goal of extracting a module as complete as possible, but using multiple memory dumps (i.e., it makes an interdump extraction). This type of extraction can be useful for detecting a malicious DLL deployed on some workstations in a corporate environment that are centrally managed by the organization.

It accepts as arguments the path to a directory where various memory dumps are located and the name of the module to extract. Since we currently cannot handle multiple dumps at once from Volatility, we have implemented this tool as a separate tool but using Modex underneath. To perform extraction across multiple memory dumps, Intermodex first uses Modex to extract the module given as an argument from each memory dump and then combines all these extracted modules appropriately. The combination of extracted modules follows the same rules as Modex, relying on the metadata generated for each extracted module.

As before, the output of Intermodex consists of three files: a *.dmp* file that represents the combined module, metadata in JSON format, and an execution log file. Intermodex also offers the option of performing a derelocation process on the extracted module, to make it more similar to the file on disk. Since Modex and Intermodex share functionality and source code, they are both in the same software repository (Fernández-Álvarez and Rodríguez, 2022).

At the moment, Intermodex only combines the extracted modules if, considering all memory dumps together, all processes where the module of interest was loaded had that module loaded at the same base address, with the same path, and the same size. This limitation is discussed in more detail in Section 5.4.

Note that the memory dumps provided to Intermodex can be from the same machine or from different machines. However, when they are from different machines, the extraction will make sense if those machines have a similar configuration (so that the same versions of the modules are combined). Otherwise, combining different versions of modules can be problematic if there are substantial differences between them.

5. Experiments

In this section we describe the experiments performed to evaluate the Modex and Intermodex tools. First, we describe the methodology followed to carry out the experiments. Then, we present and discuss the results obtained. Finally, we detail some interesting findings and the limitations of our approach.

5.1. Methodology

For the evaluation, we have used a virtual machine Windows 10 64-bit (Pro edition, version 21H2) with 8 GiB of RAM on top of the VirtualBox virtualization software. We use Windows 10 because it is the most popular version of Windows today (StatCounter, 2022). We installed four applications on that machine: a web browser (Google Chrome), a word processor (Microsoft Word), a PDF reader (Adobe Acrobat Reader DC), and a spreadsheet processor (Microsoft Excel). We have selected these types of programs because they are the most used and these particular applications because they are the most popular in their respective categories.

In this virtual machine we simulate the usual behavior of users, performing a memory dump and shutting down the machine after each simulation. This simulation is done manually, as we did not find any tool that fit our purposes without investing a considerable amount of time. To simulate user activity, we perform the following steps: (1) power on the machine; (2) open Google Chrome and use its search engine to find three popular news websites, visiting them, scrolling through them, and visiting some news articles; (3) open Microsoft Word and create a document that includes text and images; (4) view and navigate multiple PDF files with Adobe Acrobat Reader DC; and (5) open Microsoft Excel and insert some data into a spreadsheet. Each application is used for 5 min. These steps are repeated twice, defining two experimental scenarios. The first time we do not close the applications after using them, while the second time we explicitly close them.

As objects for the measurements of the experiments, we select a subset of DLLs that are loaded by all the applications we chose for

[1] Unlike Volatility 2 (which has a dedicated plugin), Volatility 3's DllList plugin has an optional flag to dump modules.

the experiments (in particular and ordered in decreasing order of size, `ntdll.dll`, `user32.dll`, `ole32.dll`, `kernel32.dll`, `advapi32.dll`, and `gdi32.dll`). These DLLs provide important functionality for Windows applications and are also used by many processes, in addition to the applications selected for experimentation.

We collect a memory dump after each step in both scenarios (for a total of 10 memory dumps). We have obtained the memory dumps through the `VirtualBox` dump function. For each DLL and scenario, we first execute `Modex` on the first memory dump (obtained after the first step). Next, we run `Intermodex` on the first and second memory dumps, then `Intermodex` on the first, second, and third, and so on until we consider all five memory dumps for each scenario.

5.2. Results and discussion

We first explore whether the DLLs selected for the experiments are used in multiple processes in both scenarios. Fig. 2 shows a box plot of the percentage of processes by scenario where each DLL is loaded. All of them (except `ole32.dll`) are loaded by a large number of processes. This is in line with what we expected, as the chosen DLLs provide important functionality, and will help the validation of our tools as there will be many modules to combine.

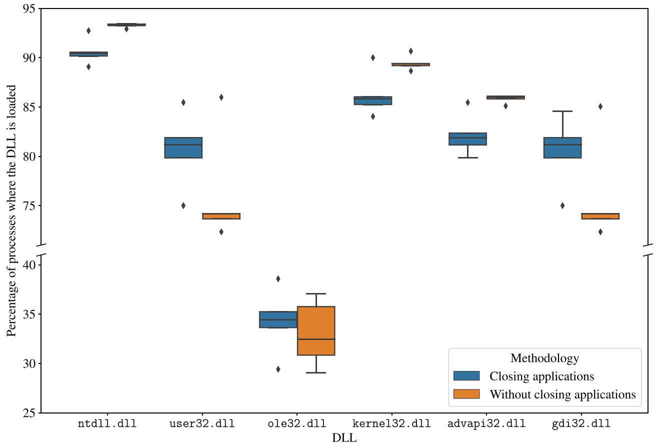


Fig. 2. Number of pages retrieved for each DLL.

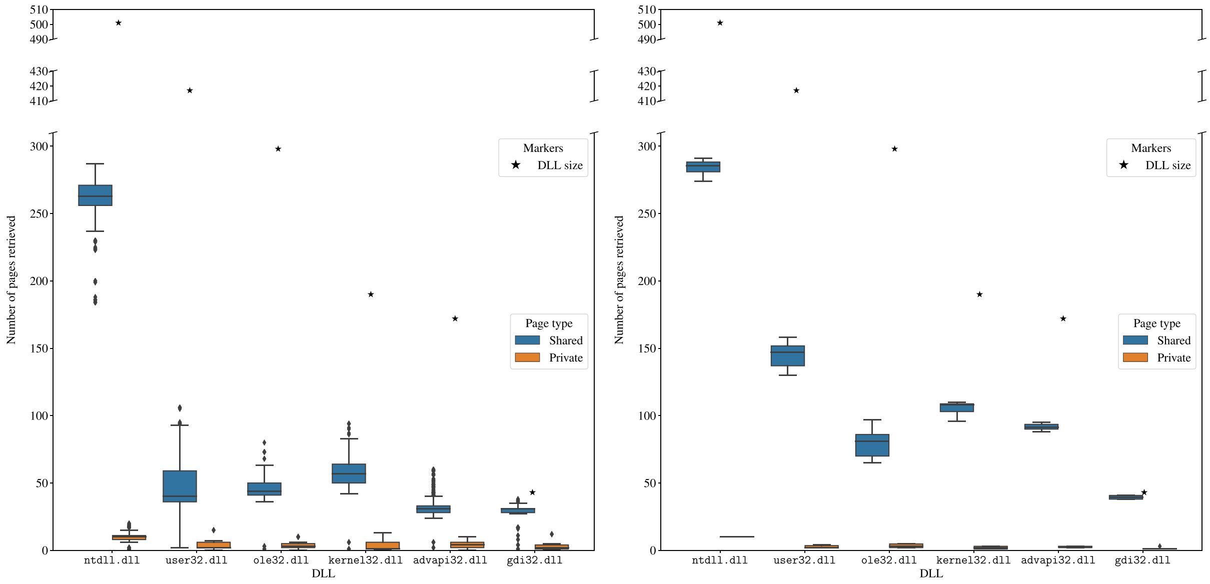
5.2.1. Intradump extraction

Next, we evaluate the performance of our `Modex` tool. To do this, we first calculate the number of pages retrieved for the DLLs in the processes where they were loaded, without combining any pages. The results of these intermediate dumped modules are plotted on Fig. 3a. We explicitly marked in this graph the size of each module (in small pages). We only found small pages (i.e., 4 KiB) in all experiments we ran. These results reflect the fact that, for any given process, only the mapped pages on its address space can be retrieved. Additionally, the number of shared pages is greater than the number of private pages for all modules, which is an expected result since DLLs are designed to be shared between different processes.

Fig. 3b shows the number of pages retrieved, but after combining the intermediate dumped modules with the `Modex` tool. As before, we explicitly marked in this graph the size of each module (in small pages). As shown, the combined module contains more pages in all cases. Also, the number of private pages decreases when modules are combined, since shared pages take precedence over private pages. As a result, the resulting final modules have fewer private pages. As a conclusion, our findings show that the intradump extraction allows to obtain more complete modules than if they are extracted from individual processes.

5.2.2. Interdump extraction

Next, we evaluate the performance of our `Intermodex` tool. The evolution of the recovered pages in both scenarios when more memory dumps are taken into account is shown in Fig. 4. Regardless of the scenario, the number of retrieved pages increases when considering more memory dumps. Recall that in our experiments, each new dump contains a new running application. Therefore, when a new application is started it is likely to use some functions of the DLL that others do not, and thus load these pages that were not previously in memory.



(a) Without combining pages.

(b) Combining pages.

Fig. 3. Percentage of processes by scenario where each DLL is loaded.

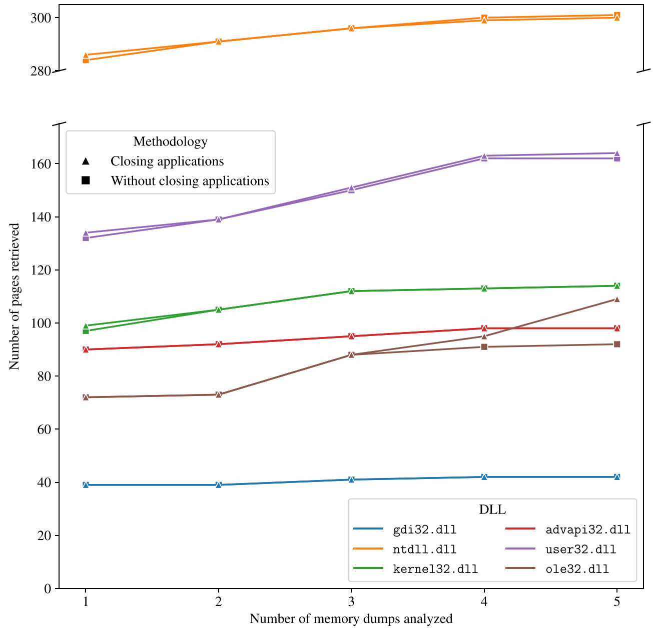


Fig. 4. Evolution of retrieved pages as more memory dumps are considered.

In general, the results in both scenarios are very similar, with slightly variations in the number of pages. The `ole32.dll` DLL exhibits a different behavior between scenarios when considering the last two memory dumps. This difference can be caused by many factors, such as the presence of a system process that suddenly started running after the third memory dump and loaded new pages from `ole32.dll`.

Finally, we compare the number of pages retrieved in both scenarios when considering only the last memory dump or when considering all memory dumps. The results are plotted on Fig. 5. In the first scenario (see Fig. 5a), there is practically no difference as the same content was in memory in both situations because the applications were not closed. These results may vary on systems with restricted RAM and a large number of running applications, as not all program content will fit in RAM when more programs are started and the operating system will swap.

In the second scenario, considering more memory dumps is clearly beneficial. As shown in Fig. 5b, the number of pages retrieved is higher considering all memory dumps than just the last memory dump, particularly for the `ntdll.dll`, `user32.dll`, and `ole32.dll` DLLs. Since we expected these differences to be larger, we performed a manual inspection of the memory dumps and discovered that some processes (in particular, the `Google Chrome` and `Adobe Acrobat Reader DC` application processes) continued to run in the background even though the user had closed the application windows.

5.3. Observations from our results

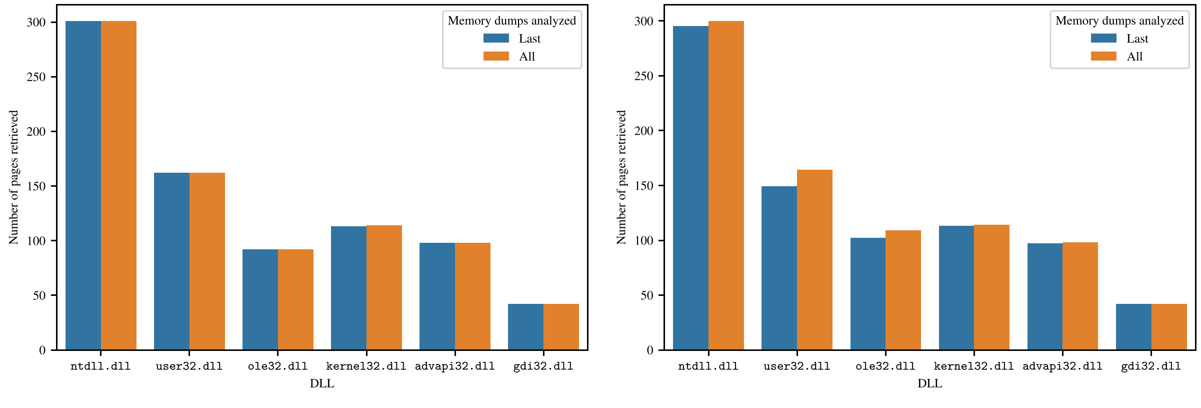
In our experiments, we found that `Modex` sometimes detected pages marked as shared and owned by a DLL loaded at the same base address in multiple processes with different content. This problem was found in DLLs other than the ones used for experimentation. The number of times we have run into this is very low, and each time we manually verified that the contents of all the pages of that particular DLL were the same except for a few pages. Since this happens very rarely, we treat it as an anomaly and implement functionality to analyze it in `Modex`.

After careful analysis, we discovered that the differences in these pages correspond to memory addresses stored within those pages. We need to investigate further to understand the reason behind these anomalies. As a consequence, we have extended the algorithm of `Modex` to, in case of detecting this problem, choose the most repeated shared page to insert it in the final module.

5.4. Limitations

ASLR is a software security mechanism that helps prevent certain memory corruption vulnerabilities by changing the base addresses of modules on every Windows startup. The base address of a module is the lowest virtual address associated with its image. Today, most DLLs have this mechanism enabled.

With the current versions of `Modex` and `Intermodex`, the base addresses of the modules must be the same to be combined. This can be a limitation when mixing memory dumps from different machines. A realistic scenario where there are multiple memory dumps is a corporate environment where multiple computers are centrally managed so that they all have the same configuration (and few differences between them), and some have been infected by a malicious DLL. To solve this issue, a page-granularity level derelocation process is required to normalize the page content (Martín-Pérez et al., 2021a) before combining the dumped modules. We plan to add this feature to our tools as future work.



(a) Without closing applications.

(b) Closing applications.

Fig. 5. Number of pages retrieved in both scenarios considering only the last memory dump or considering all memory dumps.

6. DLL hijacking detection

In this section, we first describe how the feature incorporated in `Modex` and `Intermodex` works to detect DLL hijacking attacks. Then we validate it through a proof of concept.

6.1. Description of the detection method

The DLL hijacking techniques described in Section 2.4 can be detected using `Modex` and `Intermodex` with the `--detect` flag. When this argument is given, our tools try to detect whether the module given as argument has been hijacked and no modules are extracted, as the goal is to detect DLL hijacking techniques. In this case, the JSON file provided as output contains information about the detection of DLL hijacking techniques. When a hijacking attack is detected, `Modex` indicates the affected processes, while `Intermodex` also indicates the affected memory dumps.

Detection of DLL hijacking techniques is as follows. We first list all processes that have the module given as an argument loaded. We then check the module path and size in each of these processes. We consider the actual path and size of the module to be those that are most common for all the modules identified above. This allows us to detect both the loading of modules with the same name but different path, and modules on the same path but different size. Consequently, we define a potential case of DLL hijacking when at least one path is different from the most common path or at least one size is different from the most common size. A disadvantage of this approach is that it will not detect this attack when the paths and sizes of the malicious DLL and the legitimate DLL match.

Note that our detection approach assumes the actual path and size of the module of most processes that have the module loaded. Therefore, we assume that the processes targeted by DLL hijacking techniques are a minority. We believe this assumption is valid, as attackers generally want to be as stealthy as possible to avoid detection.

Finally, note that the limitations mentioned in Section 5.4 do not apply in this case, as we do not use any base addresses to perform the detection. As a result, `Intermodex` can detect DLL hijacking considering multiple memory dumps from different machines at the same time, as long as they all have similar settings (i.e., same version of Windows, same Windows updates installed, etc.).

6.2. Proof of concept

To test this new functionality, we have created a DLL hijacking *Proof of Concept* (PoC). In particular, we have used the DLL proxying technique (see Section 2.4), following the details published in (Barile, 2021) and (Labro, 2019). The chosen victim application is `VLC media player`, which is a popular free and open source media player. The DLL we are targeting is `cryptbase.dll`, which is loaded by `VLC media player`, as well as other processes, and is not on the list of known DLLs. Known DLLs, stored in the `KnownDLLs` Windows object (Yosifovich et al., 2017), are not valid targets because if the DLL is in this list (such as `kernel32.dll` or `user32.dll`, for instance), Windows guarantees that the system copy of the DLL is used instead of searching and loading it (Microsoft, 2022b).

Our malicious DLL is placed in the same directory as `VLC media player` to make it easier to load when the application is run. When running, it simply opens a command shell window when the DLL is loaded, and all the legitimate functions of `cryptbase.dll` are appropriately proxied. We run this PoC in the same scenarios to those described in Section 5.1, but on a virtual machine running Windows 10 Education edition with 4 GiB of RAM due to hardware limitations to carry out the experiment. In either case, the version of the Windows OS is the same, albeit with a different edition. After running `VLC media player` and verifying that our malicious DLL is loaded, a memory dump of the machine is acquired (INFECTED DUMP). We also acquired a memory dump when the application was running without our PoC (CLEAN DUMP).

Running `Modex` on INFECTED DUMP successfully detected our PoC, giving the process identifier (PID) of the `VLC media player` process as suspicious. Results are displayed in Listing 1. Similarly, running `Intermodex` on INFECTED DUMP and CLEAN DUMP also successfully detected our PoC, marking the same PID as suspicious and INFECTED DUMP as the dump where that process was running (see Listing 2). The detection is successful because neither the malicious module's path nor its size match the original module's values.

```
{
    "memory_dump_location": "file:///tmp/
        MemoryDumps/InfectedDump.elf",
    "mapped_modules": [
        ...
    ],
    "dll_hijacking_detection_result": true,
    "suspicious_processes": [
        3208
    ]
}
```

Listing 1: DLL hijacking detection of our PoC with `Modex`.

```
{
    "dll_hijacking_detection_result": true,
    "suspicious_processes": {
        "file:///tmp/MemoryDumps/
            InfectedDump.elf": [
            3208
        ]
    }
}
```

Listing 2: DLL hijacking detection of our PoC with `Intermodex`.

Using a technique such as DLL proxying like the one we used in our PoC, threat actors can hide their malicious code behind a popular and legitimate process and give the impression that it is a common library. Also, the library used in this PoC (`cryptbase.dll`) is signed by Microsoft, so it does not look suspicious at a quick glance. Therefore, we believe that being able to detect DLL hijacking techniques can be valuable in a forensic investigation.

In short, the output of our tools can help the forensic analyst to know which DLLs may be malicious, indicating which of them (and even the processes where they are located) must be analyzed later in more detail. Also, if the infections occurred on one or more machines and the current limitations described in Section 5.4 are irrelevant in that particular situation, the `Intermodex` tool can extract the malicious module as completely as possible for further detailed analysis.

6.3. Limitations

A current drawback of our detection approach is that our tools need a DLL name. To work around this issue, for now, `Modex` and `Intermodex` can be built into an analysis pipeline to iterate through all modules found in a memory dump. As future work, we will refactor our code to enable this feature directly in the tools.

In addition, our detection approach can be circumvented by an attacker hijacking DLLs on 32-bit processes, since we focus exclusively on 64-bit processes. Similarly, if the hijacked DLL is loaded only in a single process, there would be no other processes to compare against, and therefore our approach will not be able to detect it.

7. Conclusions and future work

Multiple processes can load the same module to use its functionality. However, each process only allocates to its memory address space the parts (at page granularity) strictly necessary to function. Current tools for extracting modules from memory dumps focus on specific processes, making it difficult to retrieve a complete module. In this paper, we have developed a Volatility 3 plugin, dubbed `Modex`, which extracts a 64-bit module from a Windows memory dump as complete as possible. To do this, it combines the pages of the same module that are mapped in different processes. We have called this intradump extraction. With the same goal, but taking into account multiple memory dumps (interdump extraction), we have created the `Intermodex` tool. Both tools are available under the GNU/GPLv3 license.

To validate our tools, we have simulated user behavior using widely used software on a virtual machine. Our results show that for a given module, the more processes that are taken into account, the more pages we can retrieve. Similarly, the more memory dumps we consider, the more pages we can retrieve. We have also investigated hijacking execution flow attacks and implemented functionality in `Modex` and `Intermodex` to detect a particular type of these attacks (specifically, DLL hijacking) in a single memory dump and multiple memory dumps, respectively.

Our approach, however, has certain limitations when using dumps from different machines, such as combining the same modules with different base addresses. We aim to address this limitation as future work. Additionally, we would like to extend our tools to detect DLL injection techniques. Finally, we would also like to apply our approach to packed malware. This type of malware decrypts itself in memory before running, so extracting and combining modules from multiple memory dumps can help obtain partially decrypted malware, making it easier to analyze.

Acknowledgments

The research of Ricardo J. Rodríguez was supported by the grant TED2021-131115A-I00 funded by MCIN/AEI/10.13039/501100011033 and by European Union NextGenerationEU/PRTR, and by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-20R).

References

Aghaeikheirabady, M., Farshchi, S.M.R., Shirazi, H., 2014. A new approach to malware detection by comparative analysis of data structures in a memory image. In: 2014 International Congress on Technology, Communication and Knowledge. ICTCK), pp. 1–4.

AV-TEST, 2022. AV-ATLAS [Online;. https://portal.av-atlas.org/malware/statistics. (Accessed 7 October 2022).

Balzarotti, D., Di Pietro, R., Villani, A., 2015. The impact of GPU-assisted malware on memory forensics: a case study. Digital Investigation 14, S16 – S24. In: The Proceedings of the Fifteenth Annual DFRWS Conference.

Barile, L., 2021. DLL Hijacking using DLL Proxying technique [Online;. https://lucabarile.github.io/Blog/dll_hijacking_and_proxying/index.html. (Accessed 21 September 2022).

Beverly, R., Garfinkel, S., Cardwell, G., 2011. Forensic carving of network packets and associated data structures. Digital Investigation 8, S78–S89. In: The Proceedings of the Eleventh Annual DFRWS Conference.

Block, F., 2021. Release of PTE analysis plugins for Volatility 3 [Online;. https://insinuator.net/2021/12/release-of-pte-analysis-plugins-for-volatility-3/. (Accessed 8 September 2022).

Block, F., Dewald, A., 2019. Windows memory forensics: detecting (Un)Intentionally hidden injected code by examining page table entries. Digit. Invest. 29. S3–S12.

Bozkir, A.S., Tahillioglu, E., Aydos, M., Kara, I., 2021. Catch them alive: a malware detection approach through memory forensics, manifold learning and computer vision. Comput. Secur. 103, 102166.

Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., White, D., 2014. Approximate matching: definition and terminology. In: Techreport NIST Special Publication 800-168. National Institute of Standards and Technology.

Case, A., Maggio, R.D., Firoz-Ul-Amin, M., Jalalzai, M.M., Ali-Gombe, A., Sun, M., Richard, G.G., 2020. Hooktracer: automatic detection and analysis of keystroke loggers using memory forensics. Comput. Secur. 96, 101872.

Cohen, M., 2017. Scanning memory with yara. Digit. Invest. Special Issue on Volatile Memory Analysis 20, 34–43.

Dolan-Gavitt, B., 2007. The VAD tree: a process-eye view of physical memory. Digit. Invest. 4, 62–64.

Duan, Y., Fu, X., Luo, B., Wang, Z., Shi, J., Du, X., 2015. Detective: Automatically Identify and Analyze Malware Processes in Forensic Scenarios via DLLs. In: 2015 IEEE International Conference on Communications (ICC), pp. 5691–5696.

Fernández-Álvarez, P., Rodríguez, R.J., 2022. Modex v1.0 [Online;. https://github.com/reverseame/modex. (Accessed 7 December 2022).

Labro, C., 2019. Windows privilege escalation - DLL proxying [Online;. https://itm4n.github.io/dll-proxying/. (Accessed 21 September 2022).

Langner, R., 2011. Stuxnet: dissecting a cyberwarfare weapon. IEEE Secur. Priv. 9, 49–51.

Latzo, T., Palutke, R., Freiling, F., 2019. A universal taxonomy and survey of forensic memory acquisition techniques. Digit. Invest. 28, 56–69.

Ligh, M.H., Case, A., Levy, J., Walter, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. John Wiley & Sons, Inc.

Manna, M., Case, A., Ali-Gombe, A., Richard, G.G., 2022. Memory analysis of .NET and .net core applications. Forensic Sci. Int.: Digit. Invest. 42, 301404. Proceedings of the Twenty-Second Annual DFRWS USA 42, 301404. https://doi.org/10.1016/j.fsidi.2022.301404.

Martín-Pérez, M., Rodríguez, R.J., 2021. Quantifying paging on recoverable data from windows user-space modules. In: Proceedings of the 12th EAI International Conference on Digital Forensics & Cyber Crime. Springer, p. 19.

Martín-Pérez, M., Rodríguez, R.J., Balzarotti, D., 2021a. Pre-processing memory dumps to improve similarity score of windows modules. Comput. Secur. 101, 102119.

Martín-Pérez, M., Rodríguez, R.J., Breitinger, F., 2021b. Bringing order to approximate matching: classification and attacks on similarity digest algorithms. Forensic Sci. Int.: Digit. Invest. 36, 301120.

Microsoft, 2021a. Memory Management [Online;. https://learn.microsoft.com/en-us/windows/win32/memory/memory-management. (Accessed 7 October 2022).

Microsoft, 2021b. Modules [Online;. https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/modules. (Accessed 7 October 2022).

Microsoft, 2021c. Page state [Online;. https://learn.microsoft.com/en-us/windows/win32/memory/page-state. (Accessed 7 October 2022).

Microsoft, 2022a. Dynamic-link libraries [Online;. https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries. (Accessed 9 September 2022).

Microsoft, 2022b. Dynamic-link library search order [Online;. https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order. (Accessed 21 September 2022).

Microsoft, 2022c. PE format [Online;. https://learn.microsoft.com/en-us/windows/win32/debug/pe-format. (Accessed 7 October 2022).

MITRE ATT&CK, 2021. DLL search order hijacking [Online;. https://attack.mitre.org/techniques/T1574/001/. (Accessed 21 September 2022).

MITRE ATT&CK, 2022. Hijacking execution flow [Online;. https://attack.mitre.org/techniques/T1574/. (Accessed 7 October 2022).

Mosli, R., Li, R., Yuan, B., Pan, Y., 2016. Automated malware detection using artifacts in forensic memory images. In: 2016 IEEE Symposium on Technologies for Homeland Security. HST), pp. 1–6.

Oliver, J., 2021. Tlsh - technical overview [Online;. https://tlsh.org/papers.html]. (Accessed 19 September 2022).

Oosthoek, K., Doerr, C., 2019. SoK: ATT&CK techniques and trends in windows malware. In: Chen, S., Choo, K.K.R., Fu, X., Lou, W., Mohaisen, A. (Eds.), Security and Privacy in Communication Networks. Springer International Publishing, Cham, pp. 406–425.

Rekall, 2014. The Rekall memory forensic framework [Online;. http://www.rekall-forensic.com/. (Accessed 15 April 2021).

StatCounter, 2022. Desktop windows version market share worldwide [Online;. https://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide. (Accessed 14 September 2022).

Uroz, D., Rodríguez, R.J., 2019. Characteristics and detectability of windows autostart extensibility points in memory forensics. Digit. Invest. 28, S95–S104.

Volatility Foundation, 2022. Volatility 3 [Online;. https://github.com/volatilityfoundation/volatility3/. (Accessed 19 September 2022).

Wu, T., Breitinger, F., O'Shaughnessy, S., 2020. Digital forensic tools: recent advances and enhancing the status quo. Forensic Sci. Int.: Digit. Invest. 34, 300999.

Yosifovich, P., Ionescu, A., Russinovich, M.E., Solomon, D.A., 2017. Windows Internals, Part 1: System Architecture, Processes,threads, Memory Management,and More, seventh ed. Microsoft Press, Redmond, WA, USA.