Master thesis

Computer Science
Cyber Security

Radboud University

Radboud University Nijmegen
Faculty of Science

Towards process migration in Windows 10

Moving a running process from one machine to another

Author:
Wietze D. Mulder
s4557557
wietzem@gmail.com

Supervisor & First Assessor:
dr. ir. Erik Poll
erikpoll@cs.ru.nl

Second Assessor:
drs. Pol Van Aubel
radboud@polvanaubel.com

July 12, 2023

Abstract

Malware increasingly employs evasion techniques, raising the need for malware analysis tools. Most of the evasion and anti-analysis techniques happen during the startup of the malware, so we want dynamic analysis after this start-up phase. Current dynamic analysis of in-memory malware requires analysis on the infected machine, but we prefer not to alter the malware on the infected machine to keep the forensic traces intact. This thesis provides a solution for this analysis: by checkpointing the current state of the malware and restoring it on an analysis machine, we skip the startup of the process and retain the behavior of the process running in memory. We investigate the possibility of migrating a running process from one Windows 10 machine to another.

Process migration does not yet exist for Windows 10, but it does for Linux. This provides the background information required to understand our process migration. We incrementally increase the complexity of processes to migrate and provide methods for migrating them, starting from a toy example to a more real-world application. Later on, we examine whether process migration is a suitable method for malware analysis. The source code of our implementation can be found on: `https://github.com/keukentrap/process-copy`

We conclude process migration is possible, as we show in our proof-of-concept. We can migrate threads, memory, and common external resources such as files. However, some external sources such as networking require further research. Because of this, our process migration is a start in research for malware analysis.

Keywords: Windows, process migration, checkpointing, malware analysis

Acknowledgements

This endeavor would not have been possible without the support of the people that helped me write this thesis. I would especially like to thank dr. ir. Erik Poll for his thoughtful feedback and recommendations in the approach and structure of this thesis. I want to thank the multiple individuals who helped me with debugging the numerous crashes during development. I would also thank the ones who read my thesis repeatedly and gave insightful feedback. This thesis has become so much better with all your support and feedback.

Contents

Glossary

Throughout this thesis, we make use of standard terminology and some of which we have introduced ourselves. Figure 1 shows a conceptual description of process migration alongside the terminology. The definitions are listed below.

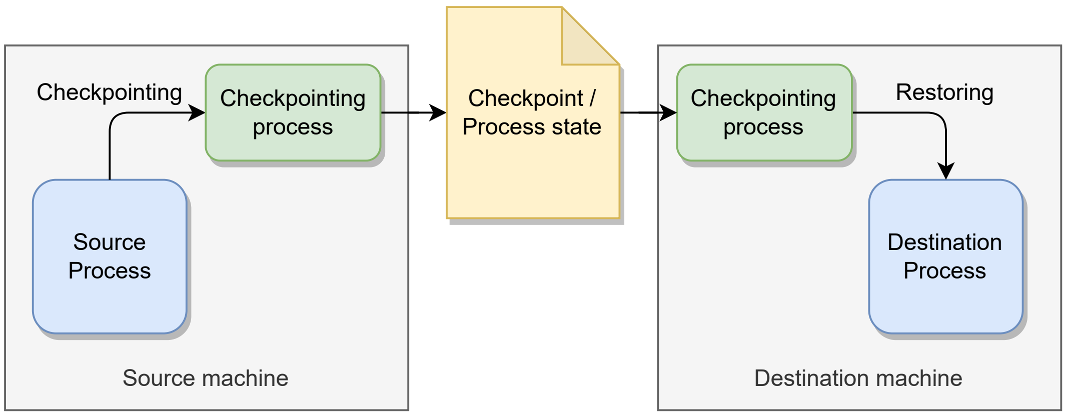


Figure 1: Conceptual visualization of process migration

Process migration	is the act of checkpointing a process from the source machine and restoring this process on the destination machine [1]. More information in Section 2.3.
Source machine	is the machine we migrate from. On this machine, we only run our checkpointing tool. We do not tamper with the machine further.
Source process	is the process running on the source machine we want to checkpoint. This process is only temporarily suspended and copied, not altered.
Destination machine	is the machine that receives the process. We can shape it to our liking to match the source machine.
Destination process	is the process we create on the destination machine in which to restore the source process.

Checkpoint	is the process state as complete as possible to restore the process from this state. A checkpoint includes all the data we collect when checkpointing the source process. Ideally, this is the complete process state.
Checkpointing	is storing the process state to restart the process from this state. A good checkpointing implementation can migrate processes. More information in Section 2.1. Checkpointing is done on the source machine.
Restoring	is the method we use to restore the destination process from a checkpoint. This is done on the destination machine.
Checkpointing process	is the process we use to checkpoint the source process and restore the destination process. This process runs our implementation of process migration.

Chapter 1

Introduction

The world becomes more and more digital and we have become more reliant on the digital domain than ever before. This brings a lot of opportunities, but also raises the risk of digital attacks: malware can do more damage than ever [2]. Malware analysis is crucial for the detection and prevention of these digital attacks. There is an ongoing cat-and-mouse game between the malware writer and the analyst for better evasion of analysis and better analysis techniques. For example, modern malware does all kinds of checks on startup to prevent dynamic analysis [3], and some malware only resides in memory. Dynamic analysis for this kind of malware can be very difficult. Dynamic analysis of in-memory malware requires analysis on the infected machine, but this is not preferred, since we want to analyze in our quarantined malware lab. We also may want to keep the malware running on the infected machine, to keep the forensic traces intact. In this thesis, we aim to find out if it is possible to migrate a running process from one Windows 10 machine to another and research if this is useful for malware analysis. We choose this operating system because 83% of malware is written for Windows [4].

To specify the goal of this thesis, our main research question is twofold and the two questions are as follows:

RQ1: Is it possible to migrate a running process from one Windows 10 machine to another machine?

RQ2: To what extent is process migration a useful method for the analysis of malware?

In Windows, process migration is not a trivial task to achieve. Running a process from the start is trivial since this would only require us to load the correct program. But resuming a checkpointed process from where it left off is complex. It requires one to checkpoint the complete state of the process and restore a process in such a way it is almost the same as the original process, yet on a different machine. To achieve this goal, this thesis investigates the Windows operating system. We explain how processes are created, how memory is managed, and how handles work.

We incrementally work towards a solution. We begin by migrating a toy example consisting of two assembly instructions in Chapter 3. This lays the foundation of our process migration. We extend our method by migrating all the memory of a process, or so-called virtual address space, in Chapter 4. Then we extend our method by migrating the handles (e.g. file handles) of a process in Chapter 5. Each step increases the complexity of our implementation. In every chapter, we demonstrate how to collect the process state to create a checkpoint and how to restore a process from a checkpoint.

'Migrating a Windows process' is a very broad research goal. To narrow this down, the scope we have is:

1. Windows 10 only (version 22H2).
2. x86-64-bit architecture.
3. No GUI applications.

Process migration is already complex enough without differences in Windows version and CPU architecture. Our method is Windows version dependent due to the use of internal structures and functions in Windows. Because of this, we further narrow our scope to a single version of Windows 10 and a single architecture. Because GUI applications also increase complexity, they are considered out of scope.

In our literature research (in Chapter 2) we find *checkpointing* was indeed possible in 1999 [5] for Windows NT on the same machine. However, the Windows OS has become a lot more complex over time, making this solution unusable for Windows 10 processes. In Linux, a few solutions exist already to migrate a process. However, current solutions for Windows are outdated or incomplete and in this research, we show process migration is indeed possible (up to a certain extent, more on this in Chapter 8).

1.1 Reading guide

In Chapter 2, we describe related work in the academic literature and existing software solutions. We explain the related notions *checkpointing*, *process migration* and *forking* and discuss how they relate to each other and our work. We describe existing solutions for Windows and Linux. Afterward, we describe related work in exploit development and fuzzing to describe their relation with process migration.

In Chapter 3 (toy example), Chapter 4 (memory), and Chapter 5 (handles), we describe our method to migrate different types of processes. Each with increased difficulty. Each chapter has the same structure. We describe how our migration method works and what we need for a proper checkpoint to store it all in a checkpoint file. Later on, we describe how to restore a process from a checkpoint file and its workings. Finally, we verify our method.

In Chapter 6 we verify whether we can migrate real-world malware and discuss possible showstoppers. Chapter 7 provides suggestions for future work. Finally, we conclude and reflect on our work in Chapter 8.

Chapter 2

Related work

The academic literature is widely available regarding process migration with other motivations [6][7][8][9]. For example, process migration is found in computing clusters to balance the computing load. Most of the research we find regarding checkpointing and process migration is with this goal in mind. The important difference between process migration in computing clusters and process migration in forensics is that computing clusters are designed to checkpoint and migrate processes whereas our environment and malware are not. Because of this difference, most process migration tools can record the state outside of the process and follow the process from startup. We are not able to accomplish this, since we miss the start of the source process.

We observe most of the research is done for the Linux kernel [7][10] and Virtual Machines as a whole [11][1]. Windows is quite different from Linux, but we can learn from the methodology and design of a solution. Also, the limitations of a certain approach are interesting to note.

In the 1990s and early 2000s, there was more focus on computations in a cluster of machines. A lot of research has been done on checkpointing and process migration [8][12]. Nowadays, the research for checkpointing for process migration on the process level subsided, but we observe a rise in practical open-source implementations [7][13].

Forking is a restricted form of checkpointing. *Forking* can be described as checkpointing a process and immediately starting a new process with this checkpoint, creating a duplicate of the same process. We discuss *forking* in Windows in more detail in Section 2.2.

The next sections demonstrate what we can learn from the academic literature and how related applications work and relate to our research. We describe *checkpointing* in Linux in Section 2.1. Section 2.2 describes *checkpointing* and

[1] https://learn.microsoft.com/en-us/azure-stack/hci/manage/vm-load-balancing

forking in Windows. Related work around *Process migration* can be found in Section 2.3. The section also describes the usage process migration done in computer clusters, fuzzing tools, and exploit writing.

With our gained knowledge we demonstrate our method to migrate processes in the next chapters. We start with a toy example in Chapter 3.

2.1 Checkpointing in Linux

Process *checkpointing* is the method of creating a snapshot of the process' state, to roll back or restart from this state in case a crash or failure is imminent. This use case overlaps with multiple other domains, for example, forensics, multiple node computing, and fault mitigation. Because of this, there are different terminologies for the same phenomena. *Checkpointing* can also be called *checkpoint/restore*, *checkpoint/restart* and *rollback recovery*. In this thesis, we will use the term *checkpointing*. We distinguish two types of *checkpointing*:

1. A checkpoint is created by the process itself to store the data required for the current state. This is called a "save state". In this thesis, we are not interested in this type of *checkpointing*.
2. A checkpoint is created externally by a service that snapshots the process, such as a hypervisor snapshotting a VM[2]. This method is called transparent checkpointing since the process does not know that a checkpoint was made.

On Linux systems, a wide collection of solutions for *checkpointing* is available. Solutions include: *DMTCP* [7], *CRIU* [13], *BLCR* [10], *Linux-CR* [14] and others [15][16]. This is mostly due to the wide usage of Linux in the academic world and Linux is fully open-source. In the next subsections, we will describe some checkpointing implementations mentioned above in more detail. The implementations chosen all have a slightly different approach.

2.1.1 DMTCP

Distributed Multi-Threaded CheckPointing (*DMTCP*) [7] runs on Linux and transparently checkpoints a single-host or distributed computation in user space, with no modifications to user code or the OS. As new processes are created, the `LD_PRELOAD` environment variable is used to preload the *DMTCP* library (`dmtcphijack.so`). This library runs before `main()` and does the following:

1. It launches a checkpoint management thread in every user process.
2. It adds a wrapper around a small number of *libc* functions to record information about open sockets, etc. at creation time.

[2]https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.vm_admin.doc/GUID-9720B104-9875-4C2C-A878-F1C351A4F3D8.html

DMTCP does a shadow bookkeeping of open sockets and file handles to be able to “replay” data streams after a recovery. *DMTCP* also uses the `/proc` file system to probe the kernel state. Their first publication is from 2006 and the last release is from 2019. However, their GitHub repository is still active in 2023.[3]

DMTCP’s approach requires preloading their library during the start of a process. This means one has to deliberately start the process with *DMTCP* to be able to checkpoint it in the future. Since the goal of our research is to avoid startup, this approach is not usable for our use case.

Alex Sardan [17] started a *DMTCP* clone for Windows. Although it compiles, restoring from a checkpoint does not work and is not complete in the current state. It seems to be a discontinued small hobby project. The code is available online and may serve as inspiration for others.

2.1.2 BLCR

Berkeley Lab Checkpoint/Restart for Linux (*BLCR*) is a checkpointing tool that uses a kernel driver to checkpoint processes [10]. Instead of injecting threads in user space as *DMTCP* does (described in Section 2.1.1), it uses a kernel module to follow the open sockets and used memory. They derive their code from a kernel module called `VMADump` [18]. `VMADump` is a kernel module that saves or restores a process’ memory space to or from a stream. *BLCR* does not support checkpointing important resources such as TCP/UDP sockets [10]. Their first release was in 2003 [19]. Kernel modules are quickly deprecated and outdated in newer kernel versions and thus need regular maintenance (as they discuss in their paper [10]). Because of the higher maintenance, the project had its last update in 2013 and is no longer maintained. Kernel programming is more time-consuming, more challenging, and also more difficult to maintain than programming in user space. Because of that, we prefer to abstain from *BLCR’s* method. If our goal is not possible in user space, *BLCR’s* method may be suitable.

2.1.3 Linux-CR

Linux-CR [14] takes the approach of designing kernel drivers and syscalls to the checkpoint and restarting a process hierarchy, i.e. multiple processes in a hierarchy. The paper was written as a proposal for standardized checkpointing in the Linux kernel. Their method is to *freeze* a tree of target processes and checkpoint all global data and states of all processes. Resources such as open files are also saved. Afterward, the processes are killed. Their checkpoint is stored in a file for it to be migrated or stored as a backup. In the references section of this report, we find many examples of application checkpointing [10],

[3] `https://github.com/dmtcp/dmtcp`

[16], [20], [21]. As described in the previous section, Since *Linux-CR* makes use of a kernel module, we wish to abstain from *BLCR's* method

2.1.4 CRIU

Checkpoint/Restore In Userspace, or *CRIU* (pronounced kree-oo) [13] can *freeze* or checkpoint running Linux containers or standalone processes and save them to disk. The checkpoint procedure relies heavily on the `/proc` file system and the `ptrace` system call [13]. The `/proc` file system acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain parameters at runtime.[4] With the `ptrace()` system call one process can observe and control the execution of another process, and examine and change the process' memory and registers. It is primarily used to implement breakpoint debugging and system call tracing [22]. The resources are obtained via:

1. Virtual memory areas are parsed from `/proc/$pid/smaps` and mapped files are read from `/proc/$pid/map_files` links.
2. File descriptor numbers are read via `/proc/$pid/fd`.
3. Core parameters of a task (such as registers and friends) are being dumped via `ptrace` interface and parsing the `/proc/$pid/stat` entry.

Then *CRIU* injects a parasite code into a process via the `ptrace` interface. This is done in three steps:

1. At first we inject only a few bytes for an *mmap* syscall at the instruction pointer when the task is at the moment of seizing.
2. Then `ptrace` allows us to run the injected syscall and we allocate enough memory for a parasite code chunk we need for dumping.
3. The parasite code is copied into a new place inside the dumper address space and the instruction pointer is set respectively to point to the parasite code.

From this context *CRIU* reads more information such as credentials and contents of memory. *CRIU* is currently used and is integrated into many container runtimes such as Docker[5], Podman[6] [23], Kubernetes[7] and LXC[8]. *CRIU* can also be used for process migration.

CRIU seems to be the modern solution for checkpointing in Linux. *CRIU* was initially released in 2012 [13]. The latest release was in 2022, which shows us

[4] `https://www.kernel.org/doc/html/latest/filesystems/proc.html`
[5] `https://docs.docker.com/engine/reference/commandline/checkpoint/`
[6] `https://podman.io/docs/checkpoint`
[7] `https://fosdem.org/2023/schedule/event/container_kubernetes_criu/`
[8] `https://linuxcontainers.org/lxc/manpages/man1/lxc-checkpoint.1.html`

it is actively being developed. We can learn what *CRIU* stores in modern operating systems to create a proper checkpoint. One might think *CRIU* would be very useful for this research. However, since *CRIU* heavily relies on the `/proc` file system, which is not all available on Windows, we cannot use the same strategy. Another downside is that handles in Linux are very different compared to Windows, so we cannot use CRIU's approach with handles. On the plus side, we can use the parasite code strategy to collect information not available from outside of the process.

2.2 Checkpointing in Windows

In 1999, Chung, *et al.* demonstrated a complete transparent checkpointing and recovery implementation on Windows NT [6][5]. They demonstrate *Nt-SwiFT*: a collection of tools to build fault-tolerant and highly available applications on Windows NT. As the authors mention, the *SwiFT* components were originally designed on UNIX, and are ported to Windows-NT. Their method of checkpointing a process is very similar to our method. The authors mention the usages of Windows API calls such as: `VirtualQueryEx` and `ReadProcessMemory`. *NT-SwiFT* intercepts all Windows API calls which create file handles, process handles, thread handles, socket handles, and Windows handles. It also intercepts Winsock communication endpoints and logs messages. It even records mouse inputs and keystrokes. During recovery, *NT-SwiFT* recreates the communication channels and replays all logged messages, keystrokes, etc. They showed that even opened files and network traffic were restored and most processes could be restored flawlessly. Their website went offline years ago, but the Web Archive still has their website [24]. Unfortunately, the code and binaries are unavailable. Nevertheless was *Nt-SwiFT* of great use as inspiration for our method.

2.2.1 Forking in Windows

Forking is an operation whereby a process creates a duplicate of the same process and that is executed afterward. This allows for concurrent programs. *Forking* can be seen as a restricted form of checkpointing. It can be described as checkpointing a process and starting a new process with this checkpoint, creating a duplicate of the same process. The difference is a checkpoint can be stored on disk and a *fork* immediately starts a new thread. Windows does not facilitate forking as a UNIX system does. We looked into forking implementations in Windows [25][26][27], but found them of little use.

2.3 Process migration

As shown in the previous checkpointing methods (Section 2.1 and Section 2.2), a well-designed checkpointing service can allow checkpointed processes to be restarted not only on their original machine but also on other machines that are supported. *Process migration* is the method of moving a running process from one machine to the other [1]. *Transparent* process migration means the process being copied does not know or notice it is being copied. Process migration is preemptive: the process is already running and the state is saved and migrated to another machine. Non-preemptive process "migration" is called process placement [28].

Khidhir *et al.* [9] described the design and implementation of process migration in Windows 7. Their paper describes technical terms which were useful for this thesis. Currently, this paper from 2012 is the most recent research for Windows on the subject we have found. Their method is similar to *Nt-SwiFT* [6], by making use of the same Windows API calls. An interesting difference is they use the PE header in the binary to restore certain parts of the memory. Their method is rather limited since they do not migrate any handles or log any messages. They only migrate memory, thread context, and binary. Nevertheless, their description of terms and their method was useful for the development of our implementation.

2.3.1 Process migration in computing clusters

Process migration is mainly used for distributed computing clusters to balance the computing load over the network and to prevent imminent crashes by migrating the process currently running on failing hardware [1]. At the end of the previous century, the academic field was actively developing operating systems that support the migration of processes. The operating systems were designed for computer clusters to distribute the load between machines with process migration. Some examples of these operating systems are: *MOSIX* [8] and *Sprite* [12].

MOSIX [29] works as a software layer that allows processes to run in remote computers as if they run locally. The current version of *MOSIX* [29] works on an unpatched Linux kernel. *MOSIX* supports (preemptive) process migration among nodes in a cluster. Process migration consists of checkpointing the memory image and setting the run-time environment. When a process is migrated, *MOSIX* intercepts all system calls and forwards them to the source machine. For every process, a mailbox is created which can receive and send messages to other processes in the cluster. These mailboxes work independently of the location of the process and allow for migratable sockets. In comparison to our scenario, forwarding all system calls to the source machine would not be desirable. This would defeat the whole purpose of the analysis.

2.3.2 Process migration in fuzzing

Fuzzing is a form of automated testing and is the method of attempting many inputs on a program to find errors in the code. To improve performance, many fuzzing tools checkpoints just before the target function to save initialization time every round. The popular fuzzing tool called *AFL* [30] runs on Linux and uses a *forking* server to *fork* at the moment the process waits for input. *AFL* is also ported to Windows and is called *WinAFL* [31]. Unfortunately *WinAFL* has a very rough approach of "checkpointing" by only moving the instruction pointer back to the start of the function that we want to fuzz. We conclude the fuzzing research is of little use to us since we require a more robust method.

2.3.3 Process migration in exploit development

In exploit development, the malware creates processes to their liking in Windows. Process injection in exploits uses many Windows API calls that are also useful for our implementation. Most of the time when process injection occurs, the shellcode is injected into a process. One of the properties of shellcode is that it works independently of its location in memory. More information about shellcode can be found in Section 3.2. *Meterpreter* [32], a popular shellcode from *Metasploit* [33], for example, has a command to inject itself inside another process. It copies its shellcode into memory and launches a thread to run the shellcode. Although this is related to process migration, inserting a process into another process is very different.

Chapter 3

Migrating a toy example

In this chapter, we describe our method to migrate a process. We demonstrate this with the easiest process we could think of a toy program consisting of two assembly instructions: we increment a register and jump back to the start of the loop. Our method of process migration takes place in three steps:

1. Collecting the data of the state of the sourcing process to create a checkpoint (Section 3.3).
2. Storing the collected data in a file called checkpoint.dat and moving it from the source machine to the destination machine (Section 3.4).
3. Restoring the destination process from this checkpoint (Section 3.5).

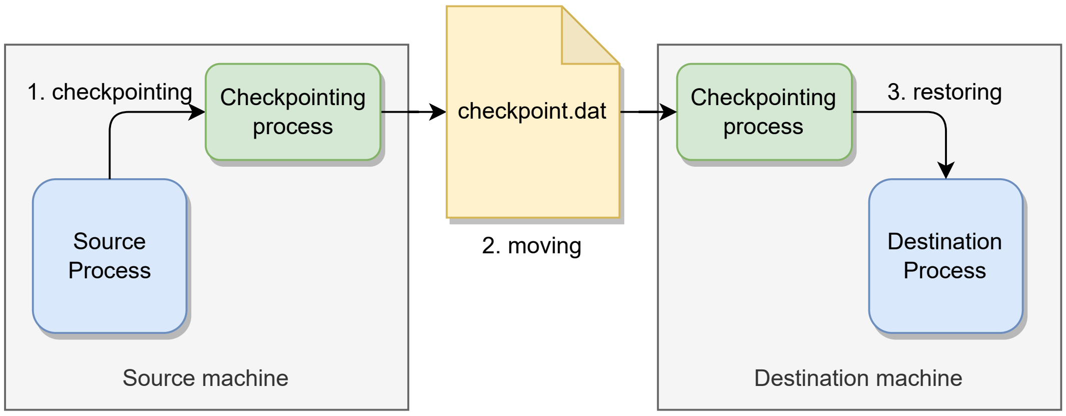


Figure 3.1: Flow of process migration

Figure 3.1 shows a visual representation of process migration. Step 2, moving the data, is a requirement that is trivial to perform. Steps 1 and 3 are the complex parts.

In this chapter, we only migrate a very simple process. As is also mentioned in

Chapter 1, we work towards a solution incrementally. By consecutively adding components to migrate, we increase the complexity of each chapter step by step. Every milestone has its chapter, and the structure of these chapters remains the same. We extend our processes with the following milestones:

1. Thread context, i.e. CPU registers (this chapter).
2. All process memory: memory pages, e.g. stack, heap, PEB, TEB, image file, and DLLs (see Chapter 4).
3. Handles, e.g. opened files, stdout, stdin, registry keys, and (network) sockets (see Chapter 5).

The source code of the complete implementation can be found on: `https://github.com/keukentrap/process-copy`.

We start by describing the background information required to understand what processes are in Section 3.1. Section 3.2 describes our toy example. Section 3.3 discusses how all relevant data is collected on the source machine to create our checkpoint. Section 3.4 describes the method of how to create a checkpoint file to send to the destination machine. Section 3.5 then discusses how to migrate our toy program.

3.1 Background: Processes

In this section, we provide background information about processes and treads in Windows and provide information about the Windows API and its documentation.

3.1.1 Difference between a program and a process

To prevent further confusion, the terms program and process may seem to be interchangeable, but there is a difference.

Program When we execute a program that is compiled, the OS creates a process to execute the program. A program is a passive entity that resides on a disk. One program can produce several processes.

Process The term process refers to program code that is loaded into memory so it can be executed by the CPU. A process can be described as an instance of a program running on a computer. A program becomes a process when loaded into memory and is called an active entity.

As we see in Figure 3.2, a program file contains the information and content to initialize a process. More information about how a process starts from a program can be found in Section 4.1.

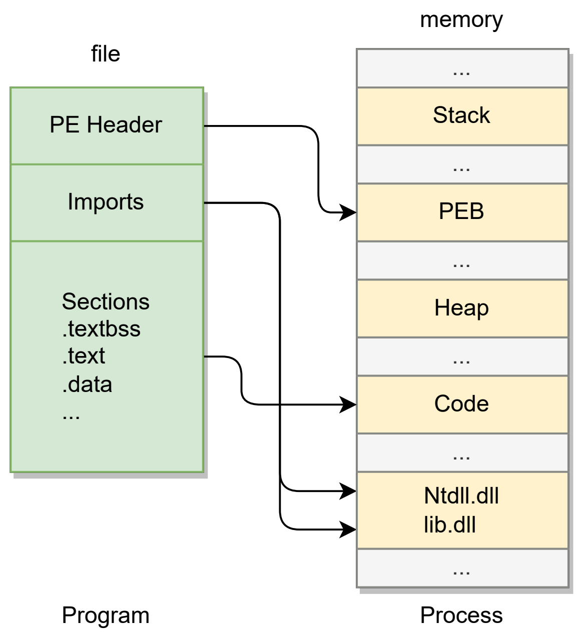


Figure 3.2: Difference between a program and a process

3.1.2 Threads and thread context

Each process has one or more threads. Each thread has a Thread Environment Block (TEB) which contains thread-specific data. More on the TEB can be found in Section 4.4.6.

To create the illusion of running more threads simultaneously than CPU cores available, the CPU constantly switches between all running threads. During this multitasking, the thread context is constantly being loaded and stored. The thread context contains all the values of the registers of the CPU when the thread was suspended. To restore a process, all threads and their context also have to be restored. The thread context contains essential information such as the instruction pointer. Without the thread context, resuming the migrated process would be impossible. To fully restore a process, all its threads have to be restored. In this chapter, we only consider single-threaded processes. In Chapter 5, we add multi-threaded processes.

The x86-64 architecture has 16 general purpose 64-bit registers.[1] Some important registers to note are shown in Table 3.1.

[1] `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture`

Register	C/C++ compiler calling convention
rip	is the instruction pointer and points to the next instruction that will be executed.
rsp	is the stack pointer and points to the top of the stack.
rcx, rdx, r8, r9	are used to pass integer and pointer arguments to functions. Any additional arguments are pushed on the stack.
rax	contains the return value of a function.

Table 3.1: Common registers and their function in x86-64

3.1.3 Windows API

The Windows API was formerly called Win32 API, but this was confusing as it roots in 16-bit Windows and also supports 64-bit Windows.[2] The Windows API contains a vast amount of functionality that runs on all versions of Windows. Our implementation makes use of many Windows API functions to collect data and restore a process.

A lot of useful information can be found in the official Microsoft documentation [34]. In particular, the books called *Windows Internals* [35][36] are of great help in learning the internals of Windows. When diving into the official documentation, one is likely to discover that some data structures (and functions) are missing. These are called "opaque" data structures. They are not officially documented, since Microsoft wants to have the ability to alter them in the future and does not document them for better backward compatibility guarantees. In this research, we make a lot of use of these opaque data structures and not officially documented functions. So we risk our migration method may not work on future versions of Windows. Luckily, these opaque data structures are not unknown.

Software such as *ReactOS* [37] and *Wine* [38] can run Windows programs that use these data structures and have documented them. Multiple websites [39][40] have collected these undocumented functions and data structures. The official debugger called *WinDbg* [41] is also an excellent tool for obtaining information about opaque data structures. Last, write-ups for Windows exploits are useful for learning the technical details of the OS.[3][4]

3.1.4 Sysinternals tools

The Sysinternals tools are an official set of debugging tools for Windows. These tools are very useful to collect information about processes. For each process, it provides a list of used resources. *Process Explorer* [42] is part of the Sysinternals

[2] https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list

[3] https://www.ired.team/offensive-security/defense-evasion/finding-all-rwx-protected-memory-regions

[4] https://captmeelo.com/redteam/maldev/2022/05/10/ntcreateuserprocess.html

suite and is a greatly improved task manager that displays a lot of information per process. Very similar tools, namely *Process Hacker* [43], and the recent successor *System Informer* [44], also exist and show even more information.

3.2 The wrapper around our toy program

We execute our toy program the same way one may execute shellcode. A shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called “shellcode” because it typically starts a command shell from which the attacker can control the compromised machine, but any piece of code that performs a similar task can be called shellcode.[5] In our case, the shellcode is benign and is not launched via a vulnerability.

This toy program only consists of only two assembly instructions. The program loops indefinitely and increments a counter. The program only uses two registers (`eax` and `rip`) and a self-allocated memory page that is easy to checkpoint. Using this toy program has multiple benefits. This loop does not require any other memory pages such as the stack, heap, or other DLLs. The loop also does not need external handles, such as files or network streams. And the memory page does not require an absolute position in the Virtual address space and thus can be easily copied. In order the launch our toy program, we have written a wrapper to launch this program, as shown in Listing 3.1.

```
 1 #include<Windows.h>
 2
 3 // start:
 4 //   inc eax    ; FF C0
 5 //   jmp start  ; EB FC
 6 const char toy_program[] = "\xFF\xC0\xEB\xFC";
 7
 8 int main()
 9 {
10     void *buf = VirtualAlloc(NULL, sizeof(toy_program), MEM_COMMIT |
       MEM_RESERVE, PAGE_EXECUTE_READWRITE);
11     if (!buf) { return 1;}
12     memcpy(buf, toyprogram, sizeof(toy_program));
13     // Cast the pointer to a function pointer and run the toy program.
14     (*(void (*)()) buf)();
15 }
```

Listing 3.1: The wrapper for our shellcode

Our toy program is currently located in the .data section in the binary. Because of a security measure called *Data Execution Prevention (DEP)*, we cannot execute our program in the .data section. It has to be moved to a memory region where it can be executed. So we allocate a page and mark it as executable. `VirtualAlloc` does this on line 11 in Listing 3.1. On line 15, we run our toy program by first casting the buffer to a function.

[5] `https://en.wikipedia.org/wiki/Shellcode`

3.3 Collecting state to create a checkpoint

The goal of our checkpointing method is to collect all the data of the process state we need to migrate the source process. In the following subsections, we show our checkpointing method for the toy program. We describe how every item in this list can be collected and copied. But first, we show the outline of our procedure to checkpoint the memory and thread context of our source process:

1. Find the source process.
2. Suspend threads of the source process.
3. Checkpoint the relevant memory page (in this case only the toy program).
4. Checkpoint thread context of the main thread.
5. Resume threads.

In order to create a checkpoint for our current example, we only have to checkpoint the memory page with our toy program and the thread context. By looking for the protection `PAGE_EXECUTE_READWRITE` on the copied memory pages (as described in Section 3.3.1), we find our needed memory page. Since our toy program only uses two registers (the `eax` and `rip` registers) we only have to checkpoint these as the thread context (as described in Section 3.3.2).

3.3.1 Collecting the toy program

We first have to find the source process. We can iterate over all processes with `CreateToolhelp32Snapshot`, `Process32First` and `Process32Next`. Note that the numbers 32 in the function names do not relate to 32-bit. With every iteration, we get information about a process, such as a process handle, its name, and the number of threads it has. This way we can find the process we want to migrate.

After we find our process, we run the code shown in Listing 3.2. We loop over the virtual memory with `VirtualQueryEx`, storing the `MEMORY_BASIC_INFORMATION` (more information in Section 4.4.4) and checkpointing the contents of the page regions with `ReadProcessMemory`. As the name `ReadProcessMemory` implies, the function reads the virtual memory of a process starting from a given address until a given size. To support multiple memory pages, we save the memory pages in a list. For our current example, we filter for the protection `PAGE_EXECUTE_READWRITE` to get our toy program. This may include false positives, but it does not hurt to migrate those memory pages since they will not be used.

```
MEMORY_BASIC_INFORMATION mbi;
LPVOID offset = 0;
while (VirtualQueryEx(process, offset, &mbi, sizeof(mbi)))
{
    if (mbi.Protect == PAGE_EXECUTE_READWRITE
        && mbi.State == MEM_COMMIT)
    {
        curmem = new Memory;
        curmem->mbi = mbi;
        curmem->buf = new BYTE[mbi.RegionSize];
        memory_list->push_back(curmem);
        ReadProcessMemory(process, mbi.BaseAddress, curmem->buf, mbi.
    RegionSize, NULL);
    }
    offset = (LPVOID)((UINT64)mbi.BaseAddress + mbi.RegionSize);
}
```

Listing 3.2: Reading all memory pages from a process

3.3.2 Collecting thread context

In Windows, the data structure called `CONTEXT`[6] contains the thread context in 64-bit processes. The functions `GetThreadContext` and `SetThreadContext` can be used to get and set the thread context of a specific thread.

With calls to the functions `CreateToolhelp32Snapshot`, `Thread32First` and `Thread32Next` in the Windows API we can find the threads corresponding to our source process. A single process may contain multiple threads that actually perform the execution. And again, the number 32 in the function names does not relate to 32-bit.

We can find the main thread by looking for the longest-living thread. We checkpoint the thread context of the main thread. In the checkpointing methods described in the next chapters, we also checkpoint the corresponding PEB and TEB addresses. More about the TEB is found in Section 4.4.6. In Chapter 5, we checkpoint all threads.

3.4 Moving our toy example

In the previous section (Section 3.3), we showed how to collect the process state. In this section, we describe how we store the collected data and we describe how we send it to the destination machine. First, we prepare the process for transmission by storing the complete state in a single file in Section 3.4.1. Second, we send this file to our destination machine.

We started with migrating a toy program from and to the same machine. This lowered the complexity and improved development time. Later on, when we developed our checkpoint file, we found that migrating to another machine did not add more complexity.

[6] `https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-context`

As described in the introduction, we are allowed to modify the destination machine to our wishes. A lot of the environment can be different when the process is migrated to another machine. Since our toy program is system-independent (only architecture dependent), we do not have to change anything on the destination machine.

3.4.1 Storing a process' state in a single file

We have defined a format to store the information collected in the previous two sections in a single file called: `checkpoint.dat`. The complete format can be found in Appendix A. For now, we add the following items to our checkpoint file.

1. Thread context of main thread (`CONTEXT`[7])
2. Self-defined memory struct that contains the toy program:
   (a) `MEMORY_BASIC_INFORMATION`[8]
   (b) Content of memory page in bytes

With `fwrite` we write the content to a new file for every element in our file format. Afterward, we send this file from the source machine to the destination machine.

3.5 Restoring the checkpoint

With our checkpoint file created in the previous section, we start to restore our destination process. With `fread` we read the contents of the process in the checkpoint file. We start "notepad.exe" as a new process with `CreateProcessA` (can be any standard program). We do not want the process to start yet, so we start the process with its thread suspended (`CREATE_SUSPENDED`).

The current procedure to restore the process is as follows:

1. Create a suspended process with `CreateProcessA`
2. Allocate the page with `VirtualAllocEx` and write the contents with `WriteProcessMemory`.
3. Set the thread context with `SetThreadContext`.
4. Resume the main thread with `ResumeThread`.

[7] https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-context
[8] https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-memory_basic_information

3.6 Verification

In this section, we verify whether our migration succeeded by demonstrating the toy program is indeed running on the destination machine. By attaching a debugger [45] and suspending the destination process, we can confirm the instruction pointer is pointing to our toy program. Here we can observe the `eax` register is incremented every cycle. We can also confirm in the Task Manager our process now utilizes 100% of a single core, as expected. From these observations we can conclude we are able to migrate toy programs.

Chapter 4

Migrating the virtual address space of a process

In the previous chapter, we designed a source process without stack, without heap and without needing an executable image it was launched from. However, most processes are started from an executable file and use a stack and heap. To migrate these more complex processes we describe in this chapter how we migrate all the memory pages in the virtual address space from the source process to the same position in the destination process. To summarize, this includes the following items stored in memory:

1. Stack
2. Heaps
3. Program image
4. Loaded DLLs
5. Process Environment Block (PEB) and related pages
6. Thread Environment Block (TEB) and related pages
7. All other pages that are allocated

In this chapter, we briefly describe how all the components mentioned above work and describe their function. Furthermore, we demonstrate how these items can be collected and can be restored.

Section 4.1 describes the background information about memory management and program startup in Windows. Section 4.2 describes how to collect all memory pages in the virtual address space. Section 4.3 describes what we append to the checkpoint file. Section 4.4 explains how to restore all memory pages from the checkpoint file. Finally in Section 4.5 we verify whether a verification process can be successfully migrated.

4.1 Background: Memory management

In order to understand why and how our migration of memory works, we need background information about how Windows implements virtual memory. We describe the basic services provided by the memory manager and key concepts such as virtual memory and security measures such as *ASLR* and *DEP*. More detailed information can be found in the books "Windows Internals" [35] and "What Makes it Page?" [46].

4.1.1 Memory Manager

As the name suggests, the memory manager controls the memory in Windows. The memory manager is located in kernel space and has two primary tasks:

1. Translating, or mapping, a process's virtual address space into physical memory. When a thread running in the context of that process reads or writes to the virtual address space, the correct physical address is referenced.
2. Paging least used contents to disk if physical memory becomes sparse, also called swapping. Paging is irrelevant for us since the physical location of the data is not important.

In this thesis, we focus on the former task, since we are not interested in where memory pages are stored in hardware. Only their virtual address and content are important.

Virtual and physical memory There is a difference between virtual memory and physical memory. As the term physical memory describes, physical memory has a physical address space, meaning an address points to a certain location in the RAM of a computer. Every process has its memory contained in a virtual address space. Virtual memory holds memory pages that the memory manager translates to a physical page stored in physical memory. An advantage of virtual memory is that processes can be isolated and are not able to read other processes' memory by default. Memory pages can be shared with other processes, as is the case with shared libraries for example.

Figure 4.1 shows how all memory management APIs relate to each other. The Virtual API is the most low-level memory API we can use in user mode to allocate memory pages. In the next paragraphs, we describe these various APIs. The dotted box shows a typical C/C++ runtime implementation of memory management using functions, such as `malloc`, `free`, `realloc`, using the Heap API.

The Local/Global API are old and deprecated memory manager functions. Starting with 32-bit Windows (Windows 95), `GlobalAlloc` and `LocalAlloc`

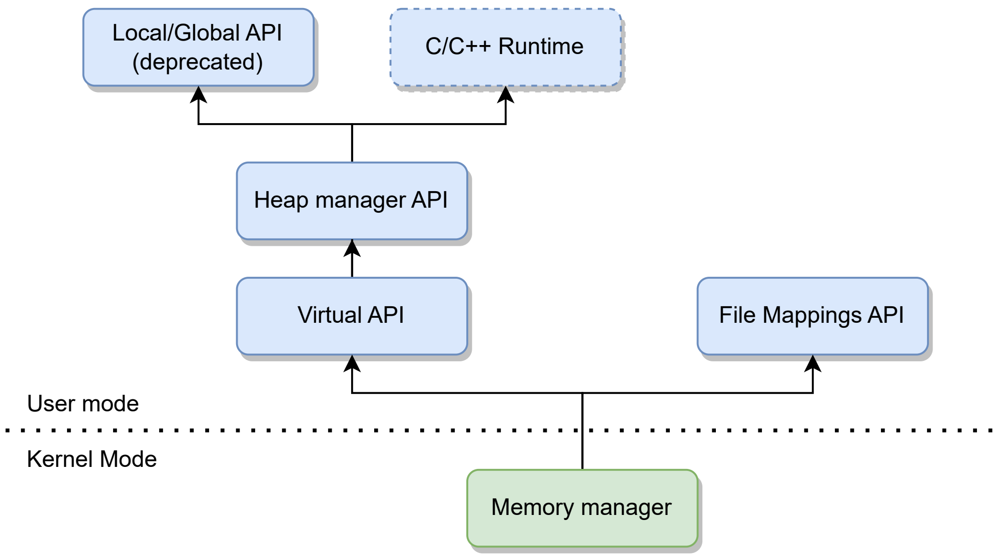


Figure 4.1: Memory API groups in user mode.

are implemented as wrapper functions that call `HeapAlloc` using a handle to the process' default heap.[1]

The Stack manager The stack manager allocates the stacks for threads. The TEB contains information about the stack such as initial size and maximum size. The stack manager is fairly simple: on thread creation, the stack is initialized and memory is allocated. The stack grows from high to low. At a certain moment when the stack keeps growing in size, the stack exceeds its allocated size, and a page fault is hit. Since a stack is located on the memory page above it, the memory manager allocates an extra memory region for the stack. The stack never shrinks in size.

The File Mapping API projects the content of a file in the virtual address space. This allows for files to be used by file pointers pointing to virtual memory. This also allows files to be opened at once by multiple processes and for files to be lazily loaded. This is done by loading the contents from the disk when a page hit occurs. This paper will not be using this API. We use the Virtual API to checkpoint file mappings.

The Heap manager manages the heaps for a process. The heap manager is made to allocate small areas with a granularity of 8 bytes (depending on the OS version) and large areas (>508 KB). The heap manager has been designed to optimize memory usage and performance in the case of these smaller

[1] `https://learn.microsoft.com/en-us/windows/win32/memory/comparing-memory-allocation-methods`

allocations [35, p.332]. How the heap manager works is a complicated subject. We will not go into too much detail for brevity. What is important to know is that there are different kinds of heaps: NT heap, low-fragmentation heap, and segment heap. Also, security features are implemented to detect simple corruptions to mitigate potential heap-based exploits [46, p.402]. Metadata is stored alongside heap allocations or at the start of the memory page, depending on the heap type.

The heap manager uses the Virtual API (`VirtualAlloc`, etc.) under the hood to make its allocations. This means we can now utilize the Virtual API to checkpoint all heaps. When we checkpoint the pages, the global metadata and the metadata for every heap are also copied. We adjust the fields in the PEB that relate to the heap, to let the heap manager know where our heaps are. More information about what is migrated can be found in Section 4.4.5.

4.1.2 The Virtual API

is the most low-level API to manage memory [35, p.309]. With this API, we can query, allocate, and free memory pages. A single memory page is 4KB. We use this to allocate our memory pages. Some functions related to the Virtual API have `Ex` at the end. `Ex` is abbreviated from `Extension`. The extension for the Virtual API functions means that a process can query and allocate memory in other processes, not just its process memory.

With `VirtualQueryEx`[2] we query memory information about a specified address range of pages from the source process. This information contains for example whether the virtual memory is mapped to real memory and what its permissions are: read, write, and/or execute.[3] `VirtualQuery(Ex)` queries a region of pages if all these statements hold:

1. The state of all the pages is the same. (`MEM_PRIVATE`, `MEM_IMAGE`, `MEM_MAPPED`, `MEM_RESERVED`, `MEM_COMMIT`)
2. The pages are allocated by the same `VirtualAlloc`, `MapViewOfFile` or their extended versions.
3. The protection on all pages is the same. (`PAGE_READ`, `PAGE_READWRITE`, `PAGE_EXECUTE_READWRITE`, etc.)

With `VirtualAlloc(Ex)`, we can allocate pages. We can reserve free pages, commit reserved pages, and reserve and commit free pages. When allocating memory, we can distinct pages in three modes:

[2] https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex

[3] https://www.ired.team/offensive-security/defense-evasion/finding-all-rwx-protected-memory-regions

`MEM_RESERVED`	The pages are reserved without any physical storage being allocated. The pages do not contain any data and protection is undefined.
`MEM_COMMIT`	The pages are committed to physical memory, either in memory or in the paging file on disk.
`MEM_FREE`	The pages are not used and are free to be allocated.

4.1.3 When a process starts

Multiple process-creation functions exist, but eventually result in calling the Windows API function `CreateProcessInternalW`. Later on, we will call this function `CreateProcess*`. The flow of `CreateProcess*` consists of seven stages [35, p. 130]:

1. Converting and validating parameters and flags.
2. Opening the image to be executed.
3. Creating the Windows executive process object.
4. Creating the initial thread and its stack and context.
5. Performing Windows subsystem-specific initialization (out of scope).
6. Starting execution of the initial thread.
7. Performing process initialization in the context of the new process.

To summarize these stages: Stage 1 sets process attributes such as a possible debug mode. Stage 2 loads the binary file into memory and detects the type of program: a batch script, a legacy Windows application, or a regular executable. In this thesis, we only consider the last option: a regular executable. At stage 3 the process is created: a Windows executive process object is created in kernel space. The `EPROCESS` struct is created in kernel space and the PEB is created in user mode. In stage 4 the initial thread and its stack and context are created. The TEB is allocated. We ignore stage 5 since Windows subsystem processes are out of scope.

Since we create a process with the `CREATE_SUSPENDED` flag, our method pauses at stage 6. Stage 7 will mainly load the `Ntdll.dll` library and starts its initialization routine called `LdrInitializeThunk`. `LdrInitializeThunk` is responsible for initializing the image loader, the heap manager, and more structures. It then loads any required DLLs. Afterward, more structures are initialized but are omitted for brevity. A more detailed description of these stages can be found in “Windows Internals Part 1” [35].

4.1.4 Structure of a process

A process is an instance of a program that is being executed by one or more threads. In Windows, every process has its own virtual address space (memory). The details about a process are mainly stored in two structs, one resides in kernel mode and the other in user mode. A process allocates in kernel mode an `EPROCESS` struct. `EPROCESS` is an opaque structure, meaning Windows does not officially document it. Since the `EPROCESS` is in kernel mode, a user in user mode is not able to edit the values inside the struct. The struct contains information such as Creation time, PID, and PEB address. It also contains various kernel-related information for a process.[4] A process also has a similar data structure in user mode, called the PEB. More info about the PEB can be found in Section 4.4.5.

4.1.5 Import Address and Lookup table

The Import Address Table is located in the PE header of the binary and contains entries for every DLL that is loaded by the executable. The Import Address Table is identical to the Import Lookup Table until the binary is loaded into memory. During binding, the entries in the import address table are overwritten with the 64-bit addresses of the symbols that are being imported.[5] Since these entries are set when a DLL or executable is loaded, the DLL or executable needs to be put in the same location as it was located.

4.1.6 Windows API library *Ntdll*

Ntdll is the native Windows library and comprises many user-mode Windows API functions. User-mode applications use the native system services routines by calling the functions in the `Ntdll.dll` library.[6] As described in Section 4.1.3, `Ntdll.dll` is loaded in every process during process startup in stage 7.

4.2 Collecting memory pages

In this section, we describe how to collect all the memory used by a single process. The source process has its own virtual address space to store its data. More background information about the virtual address space can be found in Section 4.1.

In Section 3.3 we have shown how to find the process and collect the memory pages. In the previous chapter, we only checkpointed `PAGE_EXECUTE_READWRITE` memory pages. Now we checkpoint all memory pages regardless of protection.

[4] https://www.nirsoft.net/kernel_struct/vista/EPROCESS.html

[5] https://learn.microsoft.com/en-us/windows/win32/debug/pe-format?redirectedfrom=MSDN#import-address-table

[6] https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/libraries-and-headers

We checkpoint all reserved and committed pages. We also checkpoint the reserved pages to create allocations with the correct size. We collect all memory pages, as they are all possibly relevant to the process. The new step is marked in bold. The procedure is now as follows:

1. Find the source process.
2. Suspend threads of the source process.
3. **Checkpoint all memory pages.**
4. Checkpoint thread context of the main thread.
5. Resume threads.

4.3 Moving the memory of a process

The complete format can be found in Appendix A. Compared to the previous chapter, we extend our checkpoint file with the following:

1. All memory pages: their `MEMORY_BASIC_INFORMATION` and their content.
2. The Process Environment Block (PEB).
3. The Thread Environment Block (TEB) for the main thread.

The PEB and TEB are stored twice: once in the list of all memory pages and once separately. This is done for easier reference. The method of moving the memory of a process remains essentially and is described in the previous chapter (in Section 3.4).

4.4 Restoring the checkpoint

Compared to the previous migration method, we have extended the second step by now migrating all memory pages. The new steps are marked in bold and will be discussed in the next subsections. The current procedure is as follows:

1. **Create a suspended process of `Empty.exe` with `CreateProcessA`.**
2. **Execute process initialization before main().**
3. **Kill threads created by the Windows API library *Ntdll*.**
4. **Allocate and restore all memory in the correct position.**
5. **Adjust the PEB.**
6. **Adjust the TEB.**
7. Set the thread context.
8. Resume the main thread.

More information about the newly added procedures mentioned above can be found in the next subsections. In the previous chapter, we created a process running notepad.exe. For improved consistency, we now have written a special binary we start, namely `Empty.exe`. As the name may reveal, this program does nothing but loop infinitely, as shown in Listing 4.1.

```
void main() {
    while(true){}
}
```

Listing 4.1: All code of `Empty.exe`

4.4.1 Impact of Address Space Layout Randomization (ASLR)

To prevent an attacker can predict an offset into either the program code or a DLL, *Address Space Layout Randomization (ASLR)* was invented. ASLR randomizes the virtual starting address of the executable, thread stacks, heaps, and DLLs. At the start of every process creation, the executable, thread stacks, and heap starting addresses are randomized. For DLLs, the load offset is set during boot. This means the DLL starting address is random for every boot cycle, so during this boot cycle, the starting addresses for DLLs are the same in every process. The existence of ASLR is important for process migration, since reminiscent heaps, stack, executables, and binary from '`Empty.exe`' will most likely be in another location than those from the destination process. The same holds for DLLs being located at a different address when we migrate to another machine. This is positive since collisions will be unlikely due to ASLR.

4.4.2 Initialization before `main()`

From the perspective of a programmer, one may think a process starts at `main()`. But before `main()` is called, many other functions already have been executed. This initialization is an important procedure for processes to function properly. To trigger this initialization, we resume the main thread and wait for a few seconds to suspend all the threads of our destination process again. This initializes the essential components in step 7 from Section 4.1.3. In our case, an important step is to run the image-loader initialization routine in `Ntdll.dll`, as well as the system-wide thread startup stub in `Ntdll.dll` [35, p. 149]. What is executed here in Windows is largely undocumented and differs per Windows version. By trial and error, we found out that running everything before the main gave a more stable process migration resulting in fewer unexpected crashes. Also, we discovered we are now able to attach a debugger to migrated processes, causing the analysis of crashes to be a lot easier.

4.4.3 Kill threads created by the Windows API library *Ntdll*

During the initialization mentioned in the previous section, *Ntdll* creates a thread pool of worker threads in our process. These worker threads allocate heaps, and this creates a problem. When those heaps are freed after we have restored them to the new heap manager, the process crashes. After debugging and reverse engineering the crash with IDA Pro [47], we found out the segment code in the metadata of the heap is now invalid, resulting in a panic. To make sure a crash does not happen, we kill all worker threads before we restore the heap manager. This way we prevent the heap frees from happening after we restore the heap manager. Otherwise, the process will crash after several seconds.

4.4.4 Restoring the virtual address space

Memory must be restored on the same addresses. As one would expect, quite some operations require the absolute address of data in memory. Some examples:

1. Absolute jumps and external function calls require code to be in the same address.
2. The call stack requires the functions that have been called to be on the same address.
3. Pointers store the absolute address of the value they point to.

The items mentioned above are critical to making execution work. So we need to restore the virtual address space and its content as accurately as possible. To achieve this goal, we have to understand the workings of the virtual address space.

```
typedef struct _MEMORY_BASIC_INFORMATION {
   PVOID  BaseAddress;
   PVOID  AllocationBase;
   DWORD  AllocationProtect;
   WORD   PartitionId;
   SIZE_T RegionSize;
   DWORD  State;
   DWORD  Protect;
   DWORD  Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

Listing 4.2: Struct of `MEMORY_BASIC_INFORMATION`

`VirtualQueryEx` returns a `MEMORY_BASIC_INFORMATION` struct. In Listing 4.2, the fields of `MEMORY_BASIC_INFORMATION` are shown.[7] Next to the size, protections, etc., it contains two starting addresses: `BaseAddress` and `AllocationBase`. `BaseAddress` contains the start of the region of pages. `AllocationBase` contains the base address of a range of pages allocated by the VirtualAlloc function.

[7] `https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-memory_basic_information`

These two values may be different, as shown in Figure 4.2. We describe why in the next paragraph.

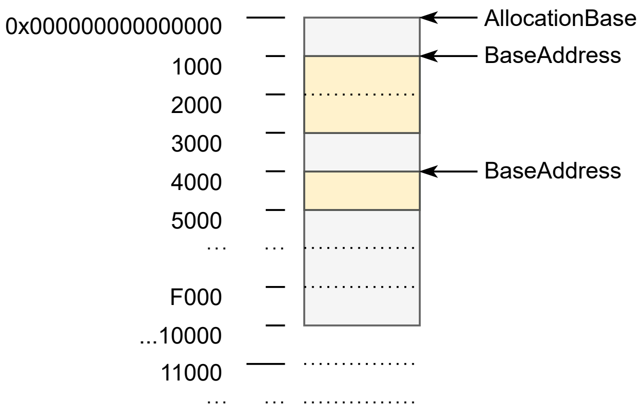


Figure 4.2: Visual representation of a memory allocation.

Reserving memory with `VirtualAlloc` has to obey the allocation granularity. The allocation granularity in Windows 10 (and almost every other version of Windows) is 64KB (0x10000 bytes) [46, p.107]. This means pages can only be reserved on addresses that align with this 64KB granularity. To allocate a page region inside this granularity (eg. a `BaseAddress` ending with 0x4000), we will start reserving at 0x0000. `VirtualAlloc` will align us to 0x0000. Luckily Windows itself also has to obey this rule, meaning all allocations have this alignment and thus can be restored. (all except USER_SHARED_DATA, more on this in Section 4.5.1)

To allocate and restore all memory in the correct position, we first reserve the page regions that share the same `AllocationBase`. This is done in the following steps:

1. Compute the required region size by finding the last page region with the same `AllocationBase` and computing: `TotalRegionSize` = `BaseAddress` - `AllocationBase` + `RegionSize`.
2. Reserve page regions with `VirtualAllocEx`.
3. Commit all page regions to memory for every page region we copied with `VirtualAllocEx`.
4. Write the contents to memory with `WriteProcessMemory`.
5. Set the correct protection with `VirtualProtectEx`.

With this procedure, we can completely recreate the virtual address space of the source process.

3rd party DLLs Many processes use libraries to access the functionality of the operating system. DLLs are used for modularization of code. Standard libraries such as `Ntdll.dll` are available on every Windows system. However, not every library will be available, so we have to migrate these as well. Migrated DLLs function without extra configuration: by migrating the memory because we also migrate the contents of the DLLs.

Due to ASLR (Section 4.4.1), standard libraries will be put in a different address than if they normally would be loaded with `LoadLibrary`. This is not a problem since the process expects the library to be available at the exact address it was loaded.

4.4.5 Adjusting the Process Environment Block (PEB)

The Process Environment Block (or PEB for short) is a data structure containing information about a process, including global context, startup parameters, data structures for the program image loader, and the program image base address. The PEB is a partially opaque data structure and is located in the virtual address space of the process itself. The PEB contains all kinds of process-specific data. For example, the `WindowTitle` is stored in the PEB. The definition of the PEB can be found online.[8]

At this stage of the restoration, the current PEB is largely incorrect, since the newly created one (from `Empty.exe`) is still used. Because the PEB Address is stored in the `EPROCESS` struct stored in kernel space, we cannot point to the migrated PEB. However, we can write to the newly created PEB and change its contents. In exploit development, this method is also called *PEB masquerading* [48]. In this section, we show how we alter the PEB to reflect the PEB of the source process.

We modify as many fields as possible, only skipping the fields that relate to machine-related values, e.g. the number of CPU cores available. We adjust the following fields in the PEB:

1. `ProcessParameters` (includes `WindowTitle`, `Environment` variables)
2. `ImageBaseAddress`
3. Heap manager fields (`ProcessHeap`, `HeapSegmentReserve`, `HeapSegmentCommit`, `HeapDeCommitTotalFreeThreshold`, `HeapDeCommitFreeBlockThreshold`, `NumberOfHeaps`, `MaximumNumberOfHeaps`, `ProcessHeaps`)
4. `LoaderData`
5. `KernelcallbackTable`

[8] https://www.vergiliusproject.com/kernels/x64/Windows%2010%20|%202016/2110%2021H2%20(November%202021%20Update)/_PEB

6. `UsersharedInfoPtr`
7. `Apisetmap`
8. Thread local storage fields (`Tlsbitmap`, `Tlsbitmapbit`, `TlsExpansionBitmap`, `TlsExpansionBitmapBits`)
9. `Ansicodepagedata`, `Oemcodepagedata` and `Unicodepagedata`

The most important adjustments are the fields related to the heap manager. After these adjustments to the PEB, the PEB now points to the migrated heap. The process now uses the migrated heap manager for future allocations and migrated heaps are recognized and fully usable.

We also set the process-wide environment variables. Thread local storage fields are set as these may contain essential information. The other fields are more of a nicety and set fields such as the program name and description.

4.4.6 Adjusting the Thread Environment Block (TEB)

Every thread has a Thread Environment Block (TEB). The Thread Environment Block (TEB) is a data structure that contains thread-specific data. The definition of the TEB can be found online.[9] As with the PEB, we cannot point to the migrated TEB, we have to adjust fields in the newly created TEB. We adjust the fields related to our scope and that are not related to the machine. What we do not adjust is the PEB address field, since we have not moved the PEB (see Section 4.4.5). We change the following fields in the TEB:

1. Stack related fields (`StackBase`, `StackLimit`, `DeallocationStack`)
2. `EnvironmentPointer`
3. Thread Local Storage related fields (`ThreadLocalStoragePointer`, `TlsSlots`, `TlsExpansionSlots`, `TlsLinks`)
4. `HeapData`
5. Network related fields (`ActiveRpcHandle`, `WinSockData`)

The TEB currently points to the old stack. By adjusting this, our restored stack now acts as a stack and automatically grows when needed. We also set the thread-specific environment variables. Thread local storage fields are set as this storage may contain essential information. Network-related fields are set to make networking possible in the future.

[9] https://www.vergiliusproject.com/kernels/x64/Windows%2010%20|%202016/2110%2021H2%20(November%202021%20Update)/_TEB

4.5 Verification

We have designed a program that uses the stack and heap to verify whether our migration succeeded and that the destination process is working as expected on the destination machine. The program shown in Listing 4.3 makes use of the stack (`recursive`), the heap (`calloc`), and a global variable (`sleep`). We also make use of the *Ntdll* function `Sleep`. As we observe the output shown in Figure 4.5, we notice that the output of both processes is the same. Even the random seed is restored.

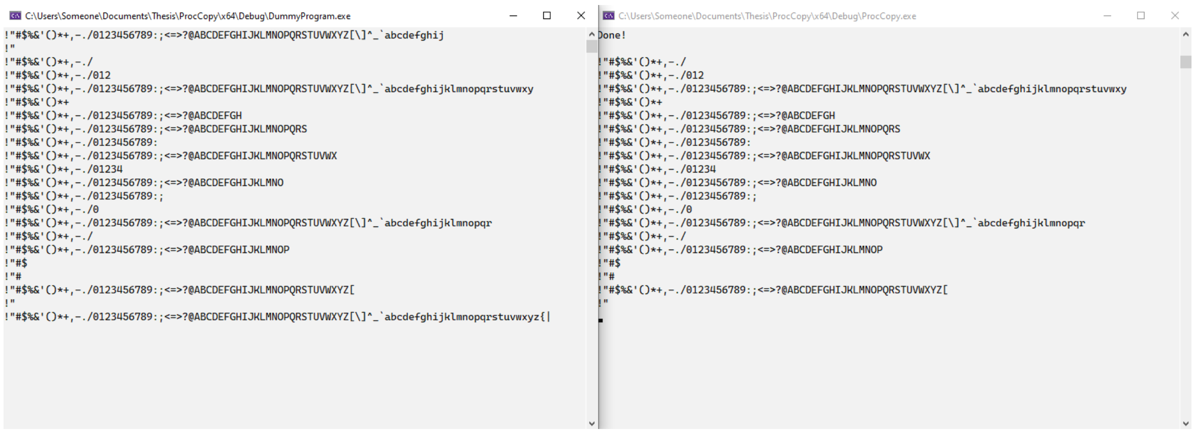
```c
#include <Windows.h>
#include <stdio.h>
#include <time.h>

int sleep;

int recursive(int i,char* p) {
    p[i] = '!' + i;
    if (i > 0) {
        return recursive(i - 1,p);
    }
    else {
        Sleep(sleep);
        return 1;
    }
}

int main()
{
    char* p;
    sleep = 3000;
    srand(time(0));

    while (1) {
        p = (char*)calloc(1000,sizeof(char));
        int i = rand() % 94;
        recursive(i,p);
        printf("%s\n",p);
        free(p);
    }
}
```

Listing 4.3: The program to verify memory

4.5.1 What still goes wrong in memory

USER_SHARED_DATA At almost the start of the virtual address space, we find the page regions: 0x7ffe0000 and around 0x7ffe2000 to 0x7ffef000. These two regions contain the `USER_SHARED_DATA` and `HYPERVISOR_SHARED_DATA`.[10] Both contain various information such as the system time and process architecture. The pages are read-only and their protection cannot be altered. This means we are not able to restore these pages. Another interesting (and trivial)

[10]https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/ns-ntddk-kuser_shared_data


C:\Users\Someone\Documents\Thesis\ProcCopy\x64\Debug\DummyProgram.exe
C:\Users\Someone\Documents\Thesis\ProcCopy\x64\Debug\ProcCopy.exe

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
!"
!"#$%&'()*+,-./
!"#$%&'()*+,-./012
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy
!"#$%&'()*+
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRS
!"#$%&'()*+,-./0123456789:
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWX
!"#$%&'()*+,-./01234
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
!"#$%&'()*+,-./0123456789:;
!"#$%&'()*+,-./0
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqr
!"#$%&'()*+,-./
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOP
!"#$
!"#
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[
!"
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|
```

```
Done!

!"#$%&'()*+,-./
!"#$%&'()*+,-./012
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy
!"#$%&'()*+
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRS
!"#$%&'()*+,-./0123456789:
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWX
!"#$%&'()*+,-./01234
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
!"#$%&'()*+,-./0123456789:;
!"#$%&'()*+,-./0
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqr
!"#$%&'()*+,-./
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOP
!"#$
!"#
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[
!"
```

Figure 4.3: The output is the same on the two processes, even when a random seed was set. As expected, the destination process is slightly behind.

phenomenon is that the second page (around 0x7ffe2000 to 0x7ffef000) has an `AllocationBase` not aligned to 64KB.

File mappings What we may foresee as a problem in our process migration is that file mappings, such as DLLs, are not allocated by the file mapping API but by the Virtual API. We were warned this could be a problem, but we have not found this to be an issue.

Chapter 5

Migrating a process that uses external resources

Besides thread context and virtual memory, we also have to take into account the context outside of the process, such as opened files and network streams. Windows uses *handles* to communicate with such resources. In this chapter, we show how the most common handles can be migrated. In particular: file handles, console handles, events, mutexes, and semaphores. This chapter builds upon the work in Chapter 3 and Chapter 4. In the next chapter (Chapter 6) we test our final implementation on real-world malware.

We first provide background information about handles and how they are managed in Section 5.1. Section 5.2 describes how to collect the handles from a process and Section 5.3 describes what we add to our checkpoint file. Section 5.4 explains how to restore the handles we have checkpointed. At last, we verify whether our method works by migrating a test process in Section 5.5. As Section 5.6 we describe how to migrate multiple threads.

5.1 Background: handles

Windows implements an object model to provide consistent and secure access to the various internal services implemented in the operating system. In this model, objects are managed by the object manager, which is responsible for creating, deleting, protecting, and tracking objects [36, p.125]. Windows has about 69 object types [36, p.128]. In Table 5.1, we provide a summarized overview of the available object types.

Every process has a handle table to maintain a list of its objects. This handle table (shown in Figure 5.1) contains references to the respective objects. The handle table resides in kernel space and is referenced by the `EPROCESS` struct (described in Section 4.1.4). Every entry in the handle table contains an object

Object Type	Represents
File	An instance of an opened file or an I/O device, such as a pipe, socket, or console
Event	An object with a persistent state that can be used for synchronization or notification. A global key is used to reference an event.
Semaphore	A counter that provides a resource gate by allowing a maximum number of threads to access the resources protected by the semaphore.
Mutex	A synchronization mechanism used to serialize access to a resource.
Key	A reference to a registry key. Although shown in the object manager namespace, it is managed by the configuration manager. A key contains zero or more values.
Section	A region of shared memory (known as a file-mapping object in Windows).
Process	The virtual address space and control information of a certain process. This also could refer to the process itself.
Thread	An executable entity within a process.

Table 5.1: Some different types of handles. Other handles are omitted for brevity and are described in Windows Internals 7 Part 2 [36]

reference along with its permissions and flags such as inheritance of the object to child processes [36].

5.2 Collecting handles

As described in Section 5.1, a handle is an index in the handle table to another object pointer. Simply copying the handle table to migrate the handles would not suffice for two reasons. First, checkpointing the handle table would require a kernel driver on the source machine. Second, we cannot copy the object pointed to in the handle table, since they are bound to the components outside the process, e.g. files on the machine. We could write a kernel driver to retrieve the information about each object, but since kernel programming is more time-consuming and more challenging, we wish to abstain from developing one.

Instead, we opt for reconstructing the components outside of the process. By collecting as much information about an object as possible, we aim to reconstruct it as correctly as possible. We cannot always create a one-to-one copy of the external object, because not all information about an object can be collected or can be restored perfectly. More about these shortcomings can be found in **??**.

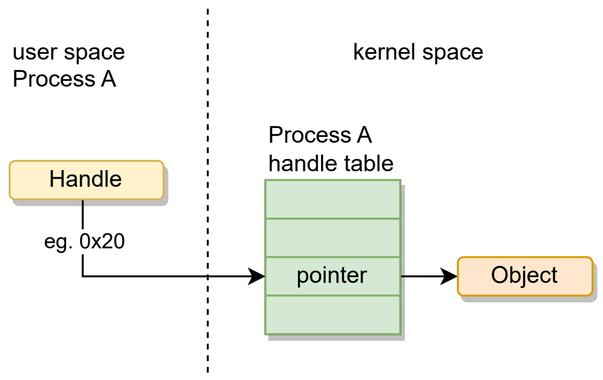


Figure 5.1: Process handle table

On the same machine, we can duplicate handles from one process to another using the Windows API in user mode. With `NtQuerySystemInformation` we can query all handles in the system. By iterating over them and matching the process ID, we can get a list of handles from our source process. With `DuplicateHandle` we can duplicate these handles to our checkpoint process for further inspection.[1] With `NtQueryObject` we can query general information about the object behind the handle, in particular:

1. Handle no.
2. Type of handle
3. Name (if given)
4. Granted Access

To correctly reconstruct the underlying object, we require more information. Which information depends on the handle type. In the next subsections, we describe how to collect information about specific handle types.

5.2.1 Collecting information about files

To migrate a file, we need to collect the handle, the file content, and the location inside the file. Different file types exist. With `GetFileType`[2] we can differentiate between a disk file (a regular file), console stream, socket/pipe unknown handles. In this thesis, we focus on regular files and console handles.

With `NtQueryObject` we obtain a file path that corresponds to the handle. This function returns the file path in *Logical Pathing Format*. Effectively this means

[1] https://cplusplus.com/forum/windows/95774/

[2] https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-getfiletype

that ‘`C:\\`’ is renamed to ‘`\device\...`’. The Microsoft documentation[3] gives a method to translate from logical pathing to the pathing format we can use, also known as *Universal pathing*.

To properly restore a file handle we need to obtain the position in the opened file. A specific function to get this position does not exist. However with `SetFilePointerEx` and giving it a relative distance of 0 it returns the current position.[4] Finally, we read the contents of the file with `ReadFile`. We are now able to collect all the information we need to restore a file handle.

5.2.2 Collecting console handles

Console handles are used to print to standard output and read from standard input. We can find in the PEB whether the console handle is standard input, output, or error.

5.2.3 Collecting information about events, mutexes and semaphores

Mutexes and semaphores are both synchronization primitives designed for concurrency control [49]. To collect information about a mutex, we use `WaitForSingleObject` to acquire ownership and set the timeout to 1 ms. If the mutex is not yet acquired by another thread, we get the mutex and learn it was not yet acquired. Otherwise, we time out and learn the mutex is owned by another thread. If we acquired the mutex, we release the mutex afterward with `ReleaseMutex`.

To collect the current count and the maximum count of a semaphore, we use the `NtQuerySemaphore`[5] function to retrieve this information. We are now able to collect enough information about mutexes and semaphores. However, the mutexes and semaphores may be used by other processes we do not migrate. This may result in incorrect behavior since the mutex or semaphore will never be released.

5.3 Moving the handles of a process

In this section, we move the handles, as described in Section 5.2, to our destination machine. The complete format can be found in Appendix A. We extend our format with the following.

1. Array with object information per handle (type, granted access, name, and more info per handle type).

[3] https://learn.microsoft.com/en-us/windows/win32/memory/obtaining-a-file-name-from-a-file-handle

[4] https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-setfilepointerex

[5] https://undocumented-ntinternals.github.io/UserMode/Undocumented%20Functions/NT%20Objects/Semaphore/NtQuerySemaphore.html

5.4 Restoring the checkpoint

The list shown below describes our final method to restore the destination process. The new step is marked in bold and will be discussed in the next subsections. We set the handle table before the process initialization (see Section 5.4.1). This is because the process initialization before `main()` fills the handle table with other handles, which will prevent us from setting the handles in the correct location. The procedure is as follows:

1. Create a suspended process with `CreateProcessA`.
2. **Set the handle table**.
3. Execute process initialization before `main()`.
4. Kill threads created by the Windows API library *Ntdll*.
5. Allocate and restore all memory in the correct position.
6. Adjust PEB and TEB.
7. Set the thread context.
8. Resume the main thread.

5.4.1 Setting the Process' handle table

As described in Section 5.1, handles are merely pointers to addresses in the process' handle table. We cannot assign a handle to a preferred address. Luckily handle addresses are predictable. The addresses are incremented by 4 for every new handle, which is quite obvious since an object reference is a 32-bit integer and 4x8 = 32. By creating every handle in the correct order, and adding "filler" handles, we can assign every handle to the correct address.

Every resource is first created and initialized in the checkpoint process. Then we move the handle over with `DuplicateHandle` to the destination process. The initialization of the resources is described in the next subsections.

5.4.2 Restoring files

The `CreateFile` function can open (or create) files and returns a handle. Before duplicating this handle, we write the content of the file with `WriteFile` and set the position in the file correctly with `SetFilePointer`.

5.4.3 Restoring console handles

As is the case with our destination process, the console handles are inherited from our checkpointing process. The console handles will be at the beginning of the handle table. Modern console handles can be duplicated.[6] By duplicating

[6] `https://github.com/rprichard/win32-console-docs/blob/master/README.md`

an inherited corresponding console handle we can put the console handles in the correct position.

One downside of this method is that handles at the beginning of the handle table cannot be restored because the inherited console handles will be taking up their location.

5.4.4 Restoring events, mutexes, and semaphores

The creation of events is done with `CreateEvent`. We create mutexes with `NtCreateMutant` and semaphores with `NtCreateSemaphore`. Note that the events will be of little use since there is no other process or thread at the other end to receive from or send to. This may also be the case with mutexes and semaphores, since they are only used by the destination process, and not by any other process.

Handles that we are currently not able to migrate are restored as an event with the type of handle as the name, as a surrogate. An example of this can be found in Figure 5.2.

5.5 Verification

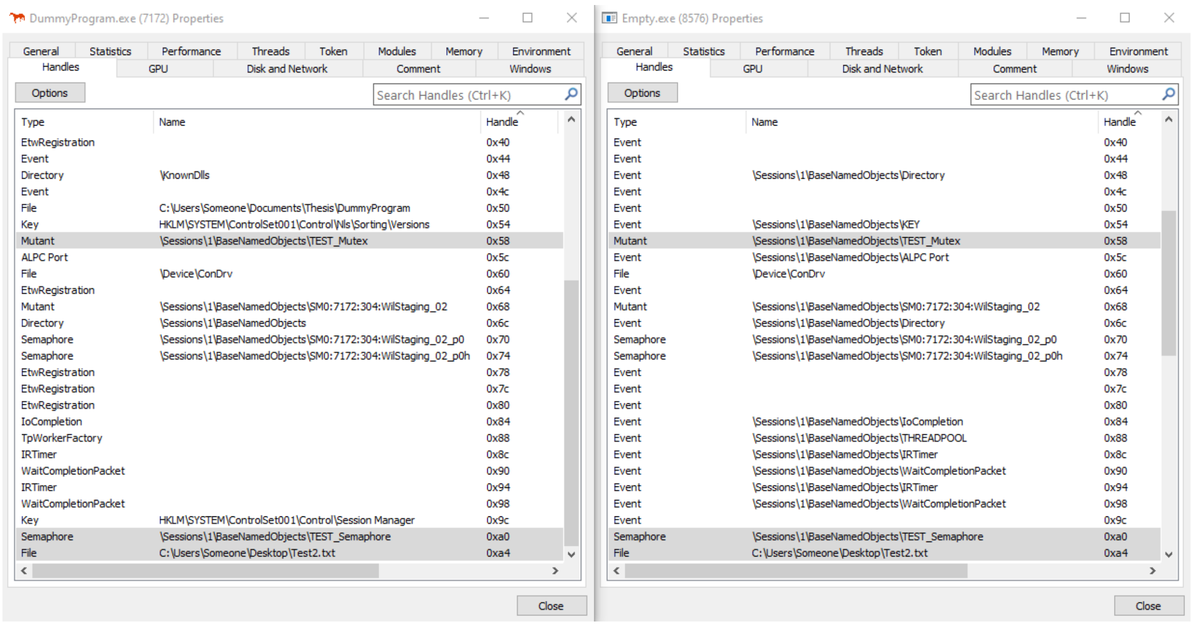
To verify whether all handles are correctly migrated, we have written a program shown in Listing 5.1 that uses the handle we have implemented: file handles, console handles, events, mutexes, and semaphores.

The program creates a mutex and semaphore. In an infinite loop, it opens a file on the desktop and asks for user input. Then it acquires the semaphore and mutex, prints a part of the content of the file, and releases both the semaphore and the mutex. Afterward, the file is closed and the loop starts over again.

```c
#include <Windows.h>
#include <stdio.h>

#define FNAME "C:\\Users\\Someone\\Desktop\\Test2.txt"

int main(void)
{
    FILE* stream;
    HANDLE mutex = CreateMutex(0, true, L"TEST_Mutex");
    HANDLE semaphore = CreateSemaphore(0, 15, 15, L"TEST_Semaphore");
    char list[105]{};

    while (1) {
        if (fopen_s(&stream, FNAME, "rb") == 0)
        {
            printf("enter position: ");
            int pos = 0;
            scanf_s("%d", &pos);
            fseek(stream, pos, SEEK_CUR);

            ZeroMemory(list, 100);
            fread(list, sizeof(char), 100, stream);

            WaitForSingleObject(mutex, 0);
            WaitForSingleObject(semaphore, 0);
            for (int i = 0; i < 100; i++) {
                printf("%c", list[i]);
            }
            ReleaseSemaphore(semaphore, 1, 0);
            ReleaseMutex(mutex);
            printf("\n");

            Sleep(200);
            fclose(stream);
        }
    }
}
```

Listing 5.1: Our program to verify handles

After migrating this process, we can observe in *System Informer* [44] the handles of both the source process and destination process (as shown in Figure 5.2. The handles we migrated are correct and in the correct location. A file named "`Test2.txt`" is created on the desktop. By interacting with the destination process, we confirm that the console handles are functional on the destination process and work as expected.



Type	Name	Handle
EtwRegistration		0x40
Event		0x44
Directory	\KnownDlls	0x48
Event		0x4c
File	C:\Users\Someone\Documents\Thesis\DummyProgram	0x50
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions	0x54
Mutant	\Sessions\1\BaseNamedObjects\TEST_Mutex	0x58
ALPC Port		0x5c
File	\Device\ConDrv	0x60
EtwRegistration		0x64
Mutant	\Sessions\1\BaseNamedObjects\SM0:7172:304:WilStaging_02	0x68
Directory	\Sessions\1\BaseNamedObjects	0x6c
Semaphore	\Sessions\1\BaseNamedObjects\SM0:7172:304:WilStaging_02_p0	0x70
Semaphore	\Sessions\1\BaseNamedObjects\SM0:7172:304:WilStaging_02_p0h	0x74
EtwRegistration		0x78
EtwRegistration		0x7c
EtwRegistration		0x80
IoCompletion		0x84
TpWorkerFactory		0x88
IRTimer		0x8c
WaitCompletionPacket		0x90
IRTimer		0x94
WaitCompletionPacket		0x98
Key	HKLM\SYSTEM\ControlSet001\Control\Session Manager	0x9c
Semaphore	\Sessions\1\BaseNamedObjects\TEST_Semaphore	0xa0
File	C:\Users\Someone\Desktop\Test2.txt	0xa4

Type	Name	Handle
Event		0x40
Event		0x44
Event	\Sessions\1\BaseNamedObjects\Directory	0x48
Event		0x4c
Event		0x50
Event	\Sessions\1\BaseNamedObjects\KEY	0x54
Mutant	\Sessions\1\BaseNamedObjects\TEST_Mutex	0x58
Event	\Sessions\1\BaseNamedObjects\ALPC Port	0x5c
File	\Device\ConDrv	0x60
Event		0x64
Mutant	\Sessions\1\BaseNamedObjects\SM0:7172:304:WilStaging_02	0x68
Event	\Sessions\1\BaseNamedObjects\Directory	0x6c
Semaphore	\Sessions\1\BaseNamedObjects\SM0:7172:304:WilStaging_02_p0	0x70
Semaphore	\Sessions\1\BaseNamedObjects\SM0:7172:304:WilStaging_02_p0h	0x74
Event		0x78
Event		0x7c
Event		0x80
Event	\Sessions\1\BaseNamedObjects\IoCompletion	0x84
Event	\Sessions\1\BaseNamedObjects\THREADPOOL	0x88
Event	\Sessions\1\BaseNamedObjects\IRTimer	0x8c
Event	\Sessions\1\BaseNamedObjects\WaitCompletionPacket	0x90
Event	\Sessions\1\BaseNamedObjects\IRTimer	0x94
Event	\Sessions\1\BaseNamedObjects\WaitCompletionPacket	0x98
Event		0x9c
Semaphore	\Sessions\1\BaseNamedObjects\TEST_Semaphore	0xa0
File	C:\Users\Someone\Desktop\Test2.txt	0xa4

Figure 5.2: Sample of handles from both processes. The source process is on the left, destination process is on the right.

5.6 Migrating a process with multiple threads

Since we are now able to migrate mutexes and semaphores, multithreading is a logical addition. We collect the thread context for every thread instead of only the main thread. During restoration, we recreate all threads and set the thread context for each of them. This way we support multi-threaded processes.

Chapter 6

Migrating real-world malware

In this chapter, we use our implementation on real-world malware samples to validate our research. We try to migrate *Turian* malware designed for 64-bit Windows systems. We also tried to migrate Shadowspad malware, but this was deemed too hard to analyze. Later on in this chapter, we describe what can be done in principle and what fundamental showstoppers are in our process migration.

6.1 Migrating Turian Malware

Turian malware is attributed to the Chinese advanced persistent threat group called Playful Taurus, also known as APT15. The group routinely conducts cyber espionage campaigns. The group has been active since at least 2010 and has historically targeted government and diplomatic entities across North and South America, Africa, and the Middle East [50].

In June 2021, ESET reported [51] that this group has upgraded its arsenal with a new backdoor called *Turian*. We have downloaded a sample from Virustotal[1] and the sha256 hash is:
`6828b5ec8111e69a0174ec14a2563df151559c3e9247ef55aeaaf8c11ef88bfa`

To migrate the malware, we first start the sample and observe its behavior. Since the sample is in the form of a DLL, we start the malware with `Rundll.exe`. As we observed, the malware creates a service called *AppMgmt*. This service is then started and to connects to the C2 server located at `mail.indiarailways[.]net` at intervals.

When we migrate the process, we can observe the destination process contains all the correct memory pages. We also find two threads with the correct call stack, but when we resume the destination process, the process immediately crashes.

[1] https://www.virustotal.com/gui/file/6828b5ec8111e69a0174ec14a2563df151559c3e9247ef55aeaaf8c11ef88bfa

By attaching the debugger from *Visual Studio* [52] before we resumed the destination process, we get the following error message immediately after resumption:

```
(ImageAtBase0x7ff878480000) in Empty.exe: 0xC0020043:
  An internal error occurred in RPC  (parameters:
  0x0000000000000002, 0xFFFFFFFFC0000024).
```

In the call stack shown by the *Visual Studio* debugger, we find the following function that caused the crash: `LRPC_CASSOCIATION::AlpcCancelMessage()`. When we look at the handles created by the event, we see a special ALPC port handle, with the name: `\RPC Control\LRPC-c40de8e9f52c6a8bc3`. ALPC ports are used for inter-process communication. We do not migrate ALPC port handles, so closing a non-existing ALPC port will result in a crash.

6.2 What can be done in principle?

As we conclude in the previous section, our implementation is a start for process migration. In this section, we discuss what does not work in our implementation but can theoretically be done in practice. In the following sections, we gradually increase the complexity we estimate for every improvement.

6.2.1 Resolving that a library is loaded twice

At the creation of every process, the native Windows API library called *Ntdll* is loaded (see Section 4.1.3). The *Ntdll* library is also migrated into a restored process, resulting in two versions of *Ntdll* being loaded into memory: the migrated and the freshly loaded version. Important process state is stored inside this library. The migrated *Ntdll* will have this state, but the freshly loaded *Ntdll* will not. The problem is the OS/kernel will use the functions of the freshly loaded *Ntdll*, resulting in a missing process state. As we have seen in our experiments, this results in unexpected behavior and mostly crashes. By hooking every function in the freshly loaded *Ntdll* with the migrated *Ntdll*, one can prevent the ambiguous state.

6.2.2 Files and registry keys that are accessed after restoration

It is imaginable that a process created a configuration file, closes the file, and reopens the file again after a while. When the file is not open during migration, we do not migrate the file. We can solve this by migrating every file in the system possibly on an on-demand basis. The same holds for registry keys.

6.2.3 Migrating a wider range of handle types

As mentioned before, our process migration does not support the complete range of handle types. We expect that by adding support for a wider variety of handles, it will become possible to migrate even more complex processes.

Registry keys, named pipes, and thread handles are common handles in malware and can be restored as shown in Table 6.1.

Object Type	Represents
Registry key	Can be restored with `RegCreateKeyExA`.[2]
Named pipe	A pipe for inter-process communication. Can be restored with `CreatePipe`.[3]
ALPC Port	A method for inter-process communication used internally by Windows. Can be restored with `NtAlpcCreatePort` and `NtAlpcConnectPort`.[4]
Thread	An executable entity within a process. By collecting every thread ID first and translating those, the handle can be restored with `OpenThread`.[5]

Table 6.1: Migratable handle types, with varying complexity

6.2.4 Some handles require both a sending and receiving party to be restored

In Chapter 5, we describe that it is seldom possible to create a one-to-one copy of everything on the handle. For some handles, it will be possible to restore all functionality, as is the case with normal file handles. But as with events, for example, this may be a showstopper. As shown in Section 6.2.4, the other end of the events is currently not restored. One could restore the missing process with a mock-up, but this will not restore the complete functionality. Handle types that fall under this category are: events, mutexes, semaphores, named pipes, ALPC ports, and network sockets

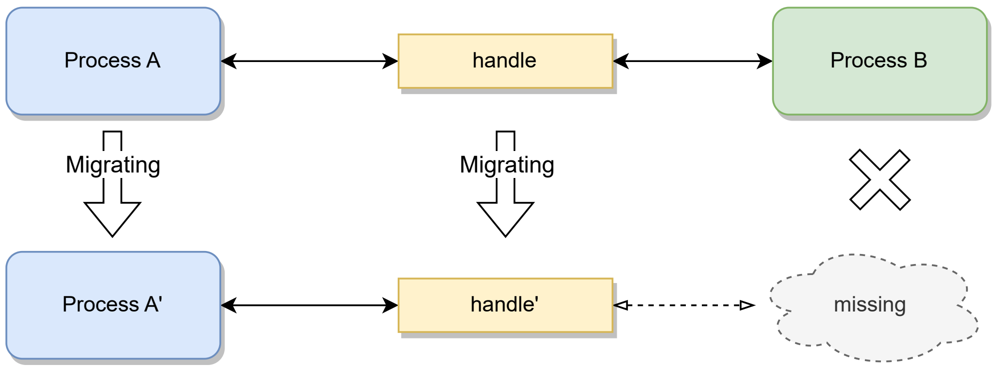


Figure 6.1: The current implementation does not also migrate other processes or drivers that utilize the same resource.

6.2.5 Restoring networking functionality

The main difficulty in migrating processes that use networking is the required initialization with `WSAStartup`. Windows networking operates via the *Winsock2* networking driver. We can migrate all states inside the process, but we cannot migrate the state in the Winsock driver in the kernel. As we can see in Figure 6.2 and depicted with a 1, the state inside the networking driver is not migrated. We propose that developing a kernel driver to modify the Winsock driver will solve this issue.

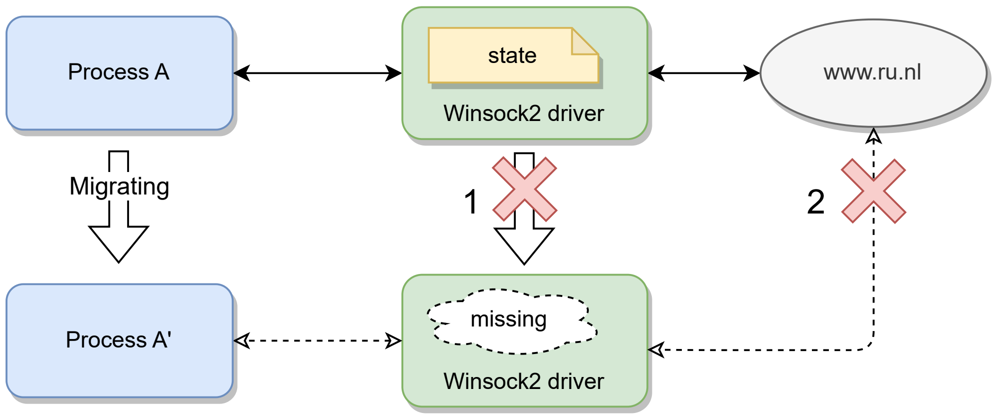


Figure 6.2: 1: The connection between Winsock2 driver and process is not restored. 2: Established connections are not restored.

When the networking functionality can be restored, listening sockets can be restored as well. Socket handles are file types with the name `/Device/Afd/`. Sockets cannot be copied with `DuplicateHandle`, but `WSADuplicateSocketA` has to be used instead.[6] With `GetExtendedTcpTable`[7] we then can retrieve information such as listening IP address and port. With `getaddrinfo`, `socket`, `bind`, and `listen` we can create a listening socket.

Established network connections As shown in Figure 6.2 depicted by a 2, a network connection provides communication between two parties and currently only one side is restored. This is the case with more handles (see Section 6.2.4). When malware is migrated while a TCP connection is active, the receiving server will not be able to process the restored connection. This may be a possible showstopper. Unless of course the malware is resilient to TCP connections dropping and then restarts them.

[6]`https://learn.microsoft.com/en-us/windows/win32/winsock/socket-handles-2`
[7]`https://learn.microsoft.com/en-us/windows/win32/api/iphlpapi/nf-iphlpapi-getextendedtcptable`

Chapter 7

Future work

7.1 Extension

Because of time constraints and the complexity of process migration, our implementation is incomplete. To migrate a broader set of processes, we recommend the following extensions.

A more sound migration of memory The migration of memory is already very sufficient. However, some improvements can be made for possibly a more stable destination process. Future research could look into the following:

1. Allocate file mappings with the file mapping API.
2. Also migrate the ASLR seed, to have future heap allocation and DLL loads equal on the source and destination process.
3. Perform process hollowing on the destination process first to create a more empty canvas to restore from.
4. Resolve the issue that the `USER_SHARED_DATA` and `HYPERVISOR_SHARED_DATA` are not migratble.

Implement more handles As mentioned in Chapter 5, we have not implemented every type of handle. We recommend future work to strive towards implementing a more complete set of handles, for example, the ALPC port mentioned in Section 6.1. A kernel driver could be developed to retrieve more information about handles (see Section 5.1). The types of handles we recommend implementing first are networking, registry keys, named pipes, and thread handles.

GUI applications Intuitively we expect that graphics add a lot of complexity to the migration. Since most malware does not have a graphical interface, we left

GUI applications out of scope. With other applications in mind, this extension may be required.

32-bit applications and other Windows versions Malware is also developed as a 32-bit application for compatibility reasons, since next to 32-bit machines, 64-bit machines are also able to run 32-bit processes. It is also aimed toward other Windows versions. Our current implementation is architecture and version dependent. Future work could extend our implementation to support more architectures and Windows versions. The PE header, thread context, PEB, and TEB will be different compared to 64-bit. We expect porting to 32-bit processes would take 2 to 3 weeks. We expect porting to Windows 11 would take one week.

7.2 Different applications

In this thesis, we aimed towards using process migration for malware analysis. As shown in Chapter 2, process migration has uses in different applications. The list below describes possible different applications for process migration.

Use process migration for fuzzing Since we checkpoint a process we can use our implementation as a forking server to fuzz on a cluster of machines, as discussed in Section 2.3. Process migration may be useful when one aims to use a computer cluster for fuzzing.

For rollback recovery Since we have created a checkpointing method for running processes, this may also be used to recover processes from crashes in Windows. By checkpointing at intervals and recording I/O, we can roll back to the previous checkpoint when a crash has occurred and replay the I/O traffic. More information about rollback recovery can be found online [5].

Chapter 8

Conclusion

In this thesis, we worked towards migrating a running Windows 10 process from one machine to another. At the start of our research, we and others were skeptical about whether process migration would be possible on Windows. Now, we have demonstrated its feasibility by providing a proof of concept. We achieved this by checkpointing the state of a process and restoring a process from this state on the destination machine. Threads, memory, and some common external resources, such as files and semaphores, can be migrated by our implementation.

We divided the problem of process migration into smaller problems, starting with the simplest toy program we could imagine and gradually increasing the complexity. This approach served us well, as we were able to reach the first milestones early on in the research, confirming process migration was indeed possible.

We started this thesis with little knowledge about process migration and Windows internals. We spent around two months developing an understanding of Windows and related work. The official documentation from Microsoft for the Windows API [34] was very useful as most system calls are well documented. Furthermore, the book "Windows Internals" [35][36] was crucial for understanding the core components in Windows 10.

In Chapter 2, we describe tools for checkpointing in the literature. Most of them were developed for Linux [7][10][13] and those developed for Windows [5] are outdated and not publicly available anymore.

We began our implementation by migrating a toy program consisting of two assembly instructions in Chapter 3. We migrated the thread context and the memory page containing the toy program. Implementing this took less than two weeks. We learned that a small error already results in a crash. We expanded this by migrating all the virtual memory of a process in Chapter 4, such as the stack, heap, program image, and loaded DLLs. By performing better process initialization (see Section 4.4.2) and adjusting the field in the PEB (see

Section 4.4.5), we were able to restore the heap and stack. Most issues with migrating memory were solved, in around seven weeks.

In Chapter 5, we describe how to migrate resources outside the process, namely file handles, console handles, events, mutexes, and semaphores. Handles have been more difficult to implement than expected. Our solution in its current state, supports just the most common handle types. In Chapter 6 we attempted to migrate a real-world malware sample of *Turian* and demonstrate what still goes wrong. As a result, we conclude that further research is needed to figure out how to query every type of handle and especially to support migrating the networking capabilities.

Still, process migration is undoubtedly possible and we conclude our implementation is a start for the transfer of real-world malware. We expect that by adding support for a wider variety of handles, it will become possible to migrate even more complex processes.

Appendix A

Checkpoint file format

Type	Name	Description
`*VOID`	PebBaseAddress	Address of PEB
`PEB`	PEB	Process Environment Block
`TEB`	TEB	Thread Environment Block
`[]CONTEXT`	threads	Thread context of all threads
`[]Memory`	memory	List of all memory pages
`[]Object`	objects	List of all handle objects

A.1 Definition of `Memory`

Type	Name	Description
`MEMORY_BASIC_INFORMATION`		Details about the memory page
`[]BYTE`	buffer	contents of memory page

A.2 Definition of `Object`

Type	Name	Description
`ULONG`	address	Address of handle
`String`	type	type of handle
`String`	name	name of handle
`DWORD`	ftype	file type
`LARGE_INTEGER`	fpos	position in file
`[]BYTE`	buffer	contents of file

Bibliography

[1] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Comput. Surv.*, vol. 32, no. 3, pp. 241–299, Sep. 2000, ISSN: 0360-0300. DOI: 10.1145/367701.367728.

[2] J. Layton, "'Cyber battlefield' map shows attacks being played out live across the globe," *Metro*, news article. [Online]. Available: https://www.msn.com/en-gb/news/world/cyber-battlefield-map-shows-attacks-being-played-out-live-across-the-globe/ar-AA1dnaec.

[3] LordNoteworthy, *Al-khaser - public malware techniques used in the wild*, version 0.81, software. [Online]. Available: https://github.com/LordNoteworthy/al-khaser.

[4] AV-TEST, "Security report 2019/2020," report. [Online]. Available: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2019-2020.pdf.

[5] P. Chung, Y. Huang, D. Liang, C.-Y. Wang, and W.-J. Lee, "Winckp: A transparent checkpointing and rollback recovery tool for Windows NT applications," in *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, Jun. 1999, pp. 220–223. DOI: 10.1109/FTCS.1999.781053.

[6] P. Chung, Y. Huang, D. Liang, C.-Y. Wang, and C. Kintala, "NT-SwiFT: Software implemented fault tolerance on Windows NT," in *Proceedings of the 1998 USENIX WindowsNT Symposium*, 1998.

[7] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," *23rd IEEE International Parallel and Distributed Processing Symposium*, Feb. 2007. DOI: 10.1109/IPDPS.2009.5161063.

[8] A. Barak and O. La'adan, "The MOSIX multicomputer operating system for high performance cluster computing," *Future Generation Computer Systems*, vol. 13, no. 4, pp. 361–372, 1998, HPCN '97, ISSN: 0167-739X. DOI: 10.1016/S0167-739X(97)00037-X.

[9] A. M. Khidhir, B. A. Mustafa, and N. T. Saleh, "Design and Implementation of a CPU Bound Process Migration in Windows 7," in *2012 International Conference on Advanced Computer Science Applications*

and Technologies (ACSAT), Nov. 2012, pp. 360–365. DOI: 10.1109/ACSAT.2012.30.

[10] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, Sep. 2006. DOI: 10.1088/1742-6596/46/1/067.

[11] E. Park, B. Egger, and J. Lee, "Fast and space-efficient virtual machine checkpointing," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 75–86, ISBN: 9781450306874. DOI: 10.1145/1952682.1952694.

[12] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The sprite network operating system," *Computer*, vol. 21, no. 2, pp. 23–36, 1988. DOI: 10.1109/2.16.

[13] *Checkpoint/Restore In Userspace (CRIU)*, version 3.17.1, software, Jul. 23, 2022. [Online]. Available: https://criu.org/.

[14] O. Laadan and S. E. Hallyn, "Linux-cr: Transparent application checkpoint-restart in linux," in *Linux Symposium*, vol. 159, 2010.

[15] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems.," in *USENIX Annual Technical Conference*, 2007, pp. 323–336.

[16] H. Zhong and J. Nieh, "CRAK: Linux checkpoint/restart as a kernel module," *Department of Computer Science Columbia University*, Nov. 2001, Technical report.

[17] A. Sardan, *Transparent checkpointing for Windows applications based on the DMTCP project*, software, Jun. 16, 2013. [Online]. Available: https://github.com/alexsardan/MTCP-windows (visited on 03/09/2023).

[18] *vmadump - Dump the shared VMA of a process*, version 0.0.1, software, Sep. 10, 2020. [Online]. Available: https://rkoucha.fr/freeware/vmadump/vmadump.html (visited on 03/09/2023).

[19] P. H. Hargrove, *Berkeley Lab Checkpoint/Restart (BLCR) for LINUX*, software. [Online]. Available: https://crd.lbl.gov/divisions/amcr/computer-science-amcr/class/research/past-projects/BLCR/ (visited on 03/09/2023).

[20] O. Laadan, D. Phung, and J. Nieh, "Transparent checkpoint-restart of distributed applications on commodity clusters," in *2005 IEEE International Conference on Cluster Computing*, 2005, pp. 1–13. DOI: 10.1109/CLUSTR.2005.347039.

[21] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 361–376, Dec. 2003, ISSN: 0163-5980. DOI: 10.1145/844128.844162.

[22] Linux Foundation, *ptrace(2) - Linux man page*, software, 2023. [Online]. Available: https://linux.die.net/man/2/ptrace.

[23] H. Htet, N. Funabiki, A. Kamoyedji, X. Zhou, and M. Kuribayashi, "An implementation of job migration function using CRIU and Podman in Docker-based user-PC computing system," in *Proceedings of the 9th International Conference on Computer and Communications Management*, ser. ICCCM '21, Singapore, Singapore: Association for Computing Machinery, 2021, pp. 92–97, ISBN: 9781450390071. DOI: 10.1145/3479162.3479176.

[24] Lucent Technologies, *SwiFT for Windows NT*, version 1.0, website, Aug. 31, 1999. [Online]. Available: http://web.archive.org/web/20070912023458/http://www1.bell-labs.com/project/swift/ (visited on 03/09/2023).

[25] C. Vinschen, *Cygwin, Get that Linux feeling - on Windows*, version 3.4.6, software, Red Hat, Inc. [Online]. Available: https://cygwin.com/.

[26] S. Hufnagel, "WSL System Calls," *Microsoft*, Aug. 6, 2016, blog. [Online]. Available: https://learn.microsoft.com/en-us/archive/blogs/wsl/wsl-system-calls (visited on 03/09/2023).

[27] midipix, *cross-platform programming tool for Windows using the standard C and POSIX APIs*, version pre-alpha, software, May 28, 2023. [Online]. Available: https://midipix.org/.

[28] P. Krueger and M. Livny, "A comparison of preemptive and non-preemptive load distributing," in *[1988] Proceedings. The 8th International Conference on Distributed*, 1988, pp. 123–130. DOI: 10.1109/DCS.1988.12509.

[29] A. Barak, *MOSIX Cluster Management System*, version 4.4.4, software, Oct. 24, 2017. [Online]. Available: https://mosix.cs.huji.ac.il/ (visited on 03/09/2023).

[30] M. Zalewski, *American fuzzy lop (AFL)*, version 2.52b, software, Jul. 6, 2022. [Online]. Available: https://lcamtuf.coredump.cx/afl/.

[31] Google Project Zero, *WinAFL*, software, Dec. 20, 2022. [Online]. Available: https://github.com/googleprojectzero/winafl.

[32] Metasploit Framework, *Metasploit Documentation - Meterpreter*, software, 2022. [Online]. Available: https://docs.metasploit.com/docs/using-metasploit/advanced/meterpreter/meterpreter.html.

[33] *Metasploit — penetration testing software, pen testing security*, website, 2023. [Online]. Available: https://www.metasploit.com/.

[34] *Programming reference for the Win32 API*, documentation. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/ (visited on 04/25/2022).

[35] P. Yosifovich, D. A. Solomon, A. Ionescu, and M. E. Russinovich, *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more.* Microsoft Press, 2017, book. [Online]. Available: https://learn.microsoft.com/en-us/sysinternals/resources/windows-internals.

[36] P. Yosifovich, D. A. Solomon, A. Ionescu, and M. E. Russinovich, *Windows Internals, Part 2.* Microsoft Press, 2022, book.

[37] *ReactOS Project*, version 0.4.14, software. [Online]. Available: `https://reactos.org/`.

[38] *WineHQ - Run Windows applications on Linux, BSD, Solaris and macOS*, version 8.3, software, Feb. 2, 2023. [Online]. Available: `https://www.winehq.org/` (visited on 03/03/2023).

[39] *NTAPI Undocumented Functions*, website, fallback: `https://undocumented-ntinternals.github.io/`, Apr. 25, 2022. [Online]. Available: `http://undocumented.ntinternals.net/`.

[40] Process Hacker, *Documentation : Data Structures*, website. [Online]. Available: `https://processhacker.sourceforge.io/doc/annotated.html`.

[41] WinDbg, *Windows Debugger*, software, Apr. 13, 2023. [Online]. Available: `https://windbg.org/`.

[42] Mark Russinovich, *Process Explorer - Sysinternals*, version v17.04, software, Mar. 30, 2023. [Online]. Available: `https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer`.

[43] Winsider, *Process Hacker*, version 2.39, software. [Online]. Available: `https://processhacker.sourceforge.io/`.

[44] Winsider, *System Informer*, version 3.0.6550, software, Apr. 15, 2023. [Online]. Available: `https://systeminformer.sourceforge.io/`.

[45] D. Ogilvie, *x64dbg - an open-source x64/x32 debugger for windows*, software, May 7, 2023. [Online]. Available: `https://x64dbg.com/`.

[46] E. Martignetti, *What makes it page? The Windows 7 (x64) Virtual Memory Manager.* 2012, book.

[47] Hex-Rays, *IDA Pro - state-of-the-art binary code analysis tools*, software. [Online]. Available: `https://hex-rays.com/IDA-pro/`.

[48] Red Team Notes, "Masquerading Processes in Userland via _PEB," Feb. 1, 2019. [Online]. Available: `https://www.ired.team/offensive-security/defense-evasion/masquerading-processes-in-userland-through-%5C_peb`.

[49] E. W. Dijkstra, "The structure of the "THE"-multiprogramming system," *Communications of the ACM*, vol. 11, no. 5, pp. 341–346, 1968.

[50] Unit 42, "Chinese Playful taurus activity in Iran," *Palo Alto Networks*, Jan. 18, 2023, malware analysis report. [Online]. Available: `https://unit42.paloaltonetworks.com/playful-taurus/`.

[51] A. Burgher, "BackdoorDiplomacy: Upgrading from Quarian to Turian," *ESET*, Jun. 10, 2021, malware analysis report. [Online]. Available: `https://www.welivesecurity.com/2021/06/10/backdoordiplomacy-upgrading-quarian-turian/`.

[52] Microsoft, *Visual Studio 2022 - IDE and Code Editor for Software Developers and Teams*, version v17.5. [Online]. Available: `https://visualstudio.microsoft.com/`.