Reconstructing Executables Part 1: Between Files and Memory

huntandhackett.com/blog/reconstructing-executables-part1

Denis Nagayuk & Francisco Dominguez

Denis Nagayuk & Francisco Dominguez @
Apr 17, 2024 9:40:16 AM

Malware developers have been known for a long time to incorporate various techniques that make analyzing their creations more challenging. We've observed countless attempts of different shapes and sizes to thwart defenders' missions by ensuring that even when malware analysts get their hands on the samples, logic obfuscation combined with polymorphic behavior complicates the path to understanding at every step.

But before we can start the analysis, merely **obtaining samples** from infected machines can present its share of challenges, especially when malicious applications actively hide their artifacts. Take self-deleting malware, for instance. These programs erase themselves from the disk after launch despite remaining running. There are also more advanced tampering techniques like Process Ghosting, Doppelgänging, and Herpaderping. Interestingly, all the examples we just listed have a common property: they require at least *some* interaction with the filesystem to create processes and start their masquerading attempts. Yet, none of the files they produce or modify remain in this transient infected state for long. They either get deleted or overwritten with benign data. So, we cannot just copy files from the machine and call it a day.

The classical approach for recovering code without a backing file is reconstructing it from the process's address space. However, its inherently volatile nature opens vast possibilities for evasion. In this article, we will discuss the shortcomings of existing techniques and **offer an alternative forensic solution** for dumping PE (EXE or DLL) images that neither relies on collecting files nor parsing attacker-controlled memory regions.

What's Wrong With Parsing Memory?

Memory is volatile. An attacker can instantaneously overwrite, deallocate, or otherwise make it inaccessible. Parsing an address space of a hostile process means, to some extent, playing by its rules. Of course, the fundamental limitations on what is allowed still come from the operating system, but these are relatively lax. Under extreme conditions, a parser needs to deal with race conditions, anti-inspection tricks, and other potential interferences. At the same time, the attacker gets to know when we perform the inspection, either via side-

channel timing checks or various system APIs. But more importantly, the code of interest still needs to be there. If a program always keeps its artifacts at least partially encrypted, their automated collection becomes virtually impossible.

Despite these problems, there are several successful open-source solutions for recognizing, reconstructing, and saving assembly code and PE files from the address space of infected processes. Probably the most well-known one is **PE-sieve**. This tool understands the layout of PE files and knows how to dump them back into the on-disk form.

Other tools that can help identify memory-based tampering are **Moneta** and **System Informer** with its *Image Coherency* feature. These programs rely on different but overlapping principles of operation, including checks for modified executable pages, missing PEB records, and comparing the bytes to the disk. Realistically, these tools do a great job. But it's always better to have more alternatives ready whenever existing solutions reach their limits.

Inside File-based Process Tampering

The list of techniques we mentioned earlier as our primary targets is far from arbitrary. They all create a process from a file that temporarily has the content we are interested in. Let's take a closer look at how it happens in each case:

- **Self-deleting executables** is a relatively vague category that we will use to describe techniques that create a process from a file and then (somehow) make this file disappear shortly after, all without terminating the process. It's possible to arrive at this result in various ways: deleting the file directly via alternative stream rotation, launching a program from a .iso file and then unmounting it, and so on. The exact implementation, however, is not significant for the sake of our discussion.

- **Process Ghosting** is a logical continuation of the previous idea that deletes the file even before launching it. As we discussed in a previous blog post, Windows doesn't delete files immediately upon request. Instead, it marks them for pending deletion and completes it with the last closed handle. Because the system also prevents anyone from reopening such files (returning `STATUS_DELETE_PENDING` error), we cannot pass them to CreateProcess or NtCreateUserProcess. Yet, we can use another syscall — NtCreateProcessEx. This function doesn't understand files; instead, it requires the caller to pass an image section (a memory projection object) created from it. Long story short, an attacker marks the file for pending deletion, prepares an image section, closes the file (completing its deletion), and ends up with a process without a backing file.

- **Process Doppelgänging** is another `NtCreateProcessEx`-based tampering technique, this time mixed with filesystem transactions. An attacker chooses and overwrites a benign file inside a temporary transaction and creates a process from it. The idea here is that the isolation mechanism of transactions allows one file to exist effectively in two states at once. A transacted reader sees and uses its altered version while the rest of the system observes the unchanged original content. Once the attacker rolls the transaction back, recovering the modified data becomes problematic.

- **Process Herpaderping** is a technique that abuses image section caching logic and achieves similar results to Doppelgänging without transactions. An attacker, again, opens and temporarily overwrites a benign executable, creates an image section and a process from it, and then restores the original file content using the previously opened handle. The result of these operations is a process that has cached a transient view of the executable file.

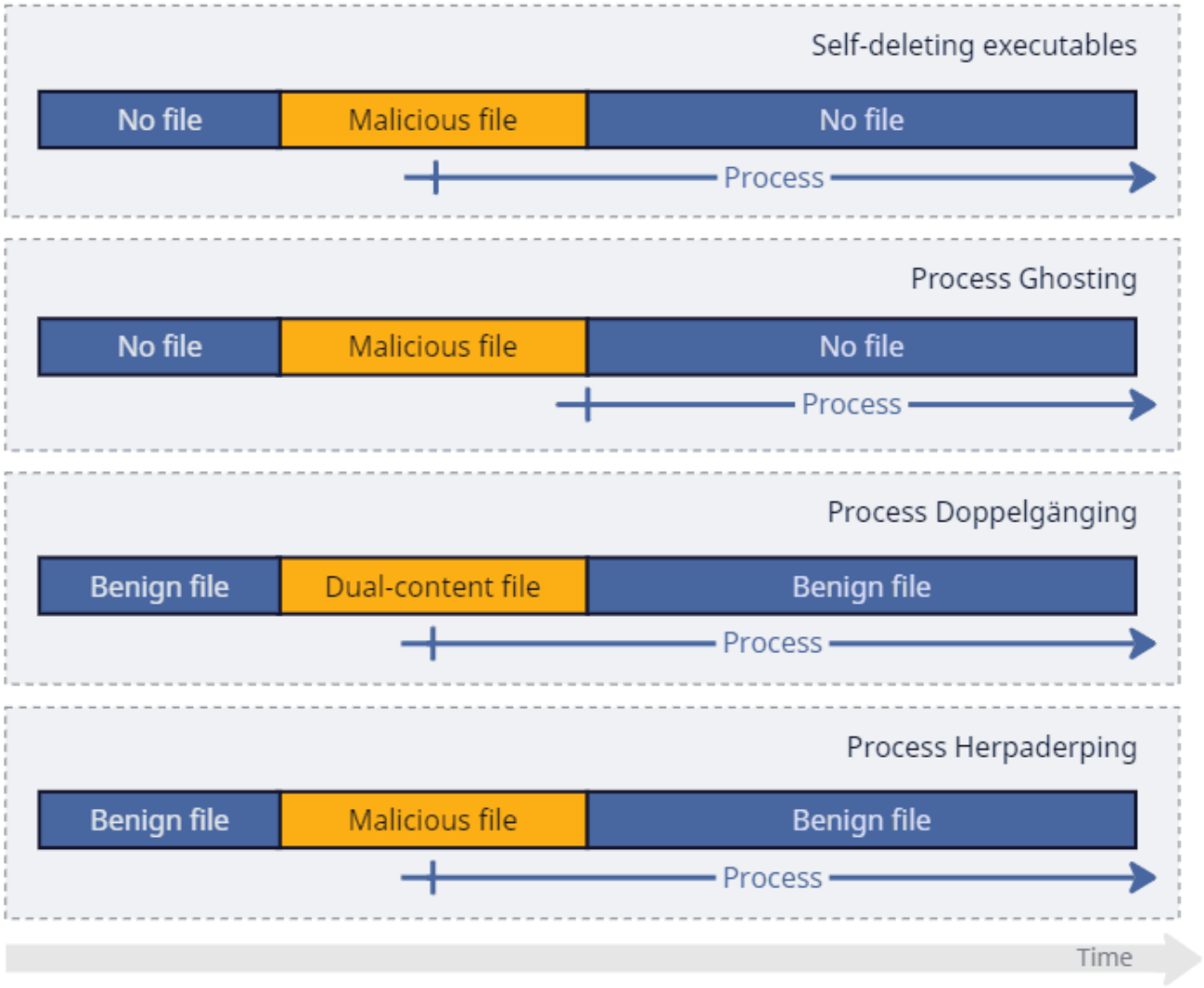

Figure: Timelines for Process Tampering techniques.

You can **read more** about how these techniques work and how to detect them in our other blog post.

Between Files and Memory

Our discussion keeps returning to an essential memory management primitive on Windows — **a section object**. The two possible kinds of such objects are *data sections* that represent plain memory views of files and *image sections* that store projections of executables. Data sections are conceptually simple — they merely provide an interface to file I/O via memory operations. On the other hand, image sections need to transform between in-memory and on-disk layouts of PE files, automatically apply page protections based on the PE section attributes, and enforce copy-on-write semantics. Because of that, data stored in image sections gets decoupled from the underlying file. Modifying either of them doesn't change the other.

We already saw that advanced tampering techniques extensively use image sections because `NtCreateProcessEx` requires so. However, even the modern `NtCreateUserProcess` (which only accepts filenames) maintains a deep connection with them. Internally, this function opens the specified file, creates an image section from it, and then continues on a similar code path.

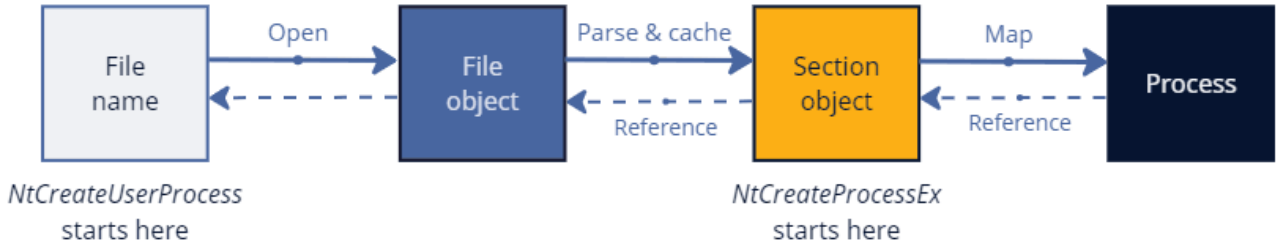

Figure: Section objects as an intermediate between files and processes.

Regardless of how we got there, the new process references the section object used during its creation inside the `EPROCESS->SectionObject` field. And, as opposed to files that an attacker can delete or overwrite, this object stores the cached copy of the original data for the entire lifetime of the process. So, the question becomes: *How can we extract this data for forensic purposes*?

Recover and Reconstruct

1. Section Handle

While kernel drivers can easily reference the object by pointer, user-mode callers must go through an API to receive a handle. Luckily, Windows exposes what we need via the ProcessImageSection info class of NtQueryInformationProcess. Quite inconveniently, the

corresponding switch case has an explicit check that disallows calling it against processes other than your own. Not everything is lost, though, as we can still rely on the classical trick of offensive security tooling: *if you cannot do something cross-process, inject some code to do it in-process!*

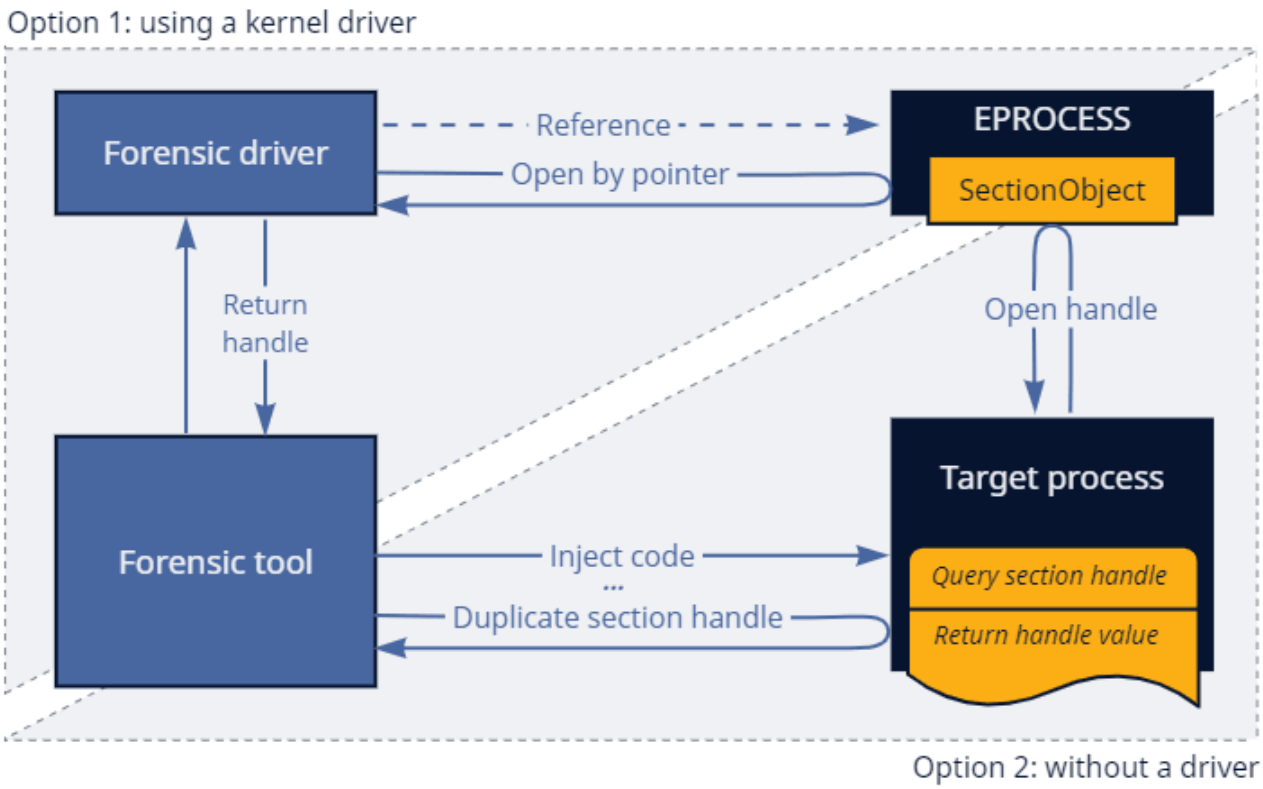

Figure: Flow comparison for section retrieval with and without driver support.

2. Layout

Once we get the handle, we can map it for parsing via NtMapViewOfSection. The mapped view would have the in-memory layout of the PE file (as opposed to the on-disk layout) prepared by the memory manager during object creation. The transformation it applies is (mostly) reversible since it primarily consists of adjusting the alignment of each PE section (I know, the terminology might be a bit confusing in this context) to the `SectionAlignment` value (usually a multiple of a page size). Returning to the file layout requires moving each region back to its more compressed form defined by the `FileAlignment` value from the PE headers.

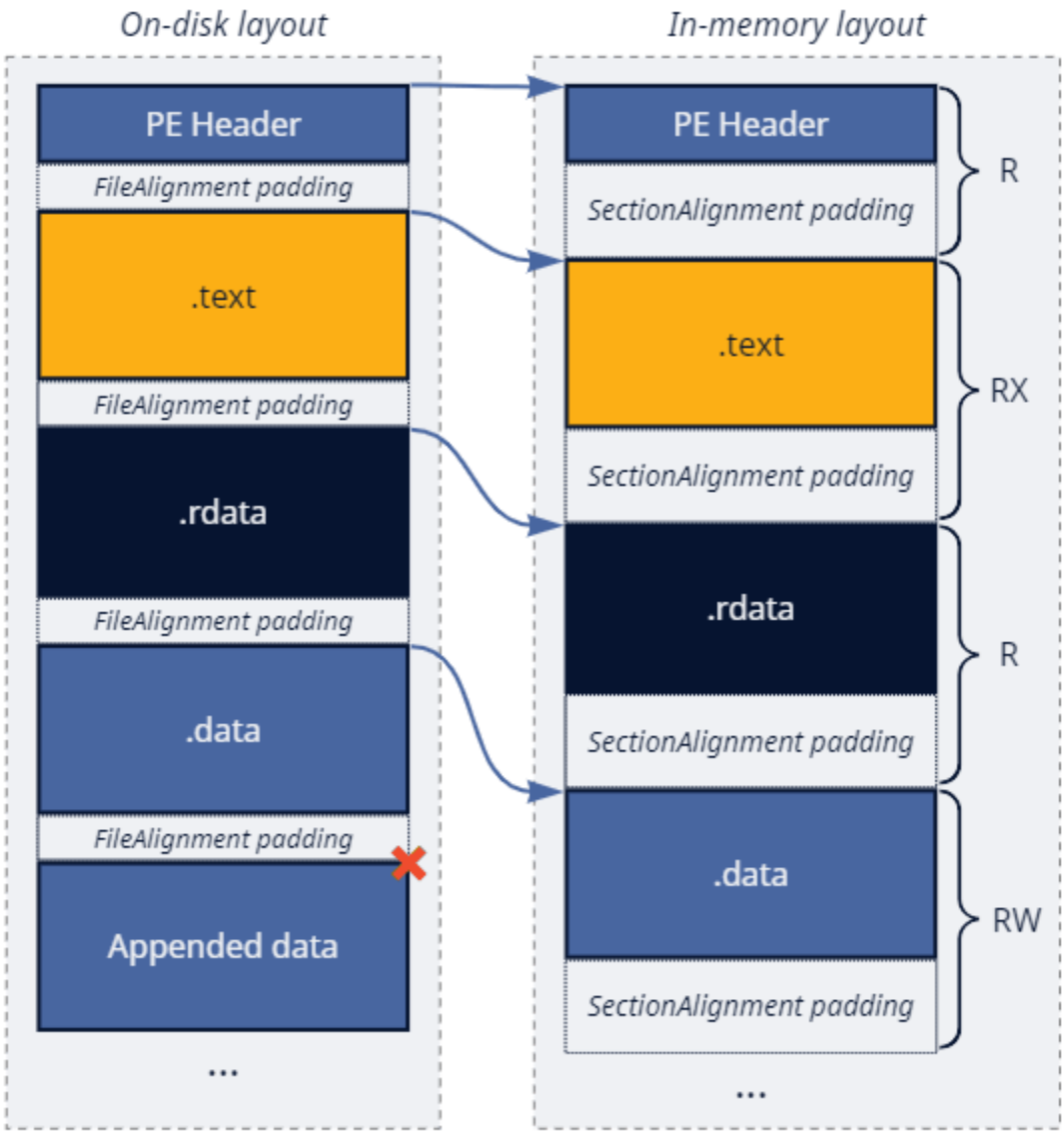

Figure: Different layouts of PE files.

3. Relocations

PE format specifies a so-called *preferred image base*. Predictability, however, comes at the cost of drastically simplifying binary exploitation, so the OS tends to move images away from this default address when possible (to facilitate Address Space Layout Randomization). There are three directions in which the system can proceed here:

- For executables with relocation information **stripped** from the binary, the image loader can only place it at the preferred base or fail (under enabled mandatory ASLR).

- For older files that don't specify `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE`, relocation happens entirely in **user mode** without much help from the kernel. First, the system maps the image section as-is, returning the address either via `PEB->ImageBaseAddress` (for EXEs) or as an output parameter of `NtMapViewOfSection` (for DLLs). If the mapped address differs from the preferred base, the LDR component from ntdll starts the relocation procedure. It parses the image relocation table, manually adjusts the protections of the corresponding memory pages to make them temporarily writable, applies the patches, and then reverts protection changes.
  A keen reader might spot two conceptual problems with this approach. First, this type of patching **triggers copy-on-write** and prevents efficient image sharing and caching across processes that load the same file. Secondly, it is incompatible with the Arbitrary Code Guard (ACG) mitigation that explicitly blocks modifying executable regions (or changing protection back to executable after modification, to be more precise). Addressing these issues requires help from the kernel.

- Newer executables with the `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` flag can use the improved mechanism called **dynamic relocation**. Whenever somebody creates an image section from a compatible executable, the memory manager automatically selects a different base address, applies relocation patches, and only then enables copy-on-write. This operation happens transparently to the caller as if the file already happened to specify a pre-randomized preferred base.

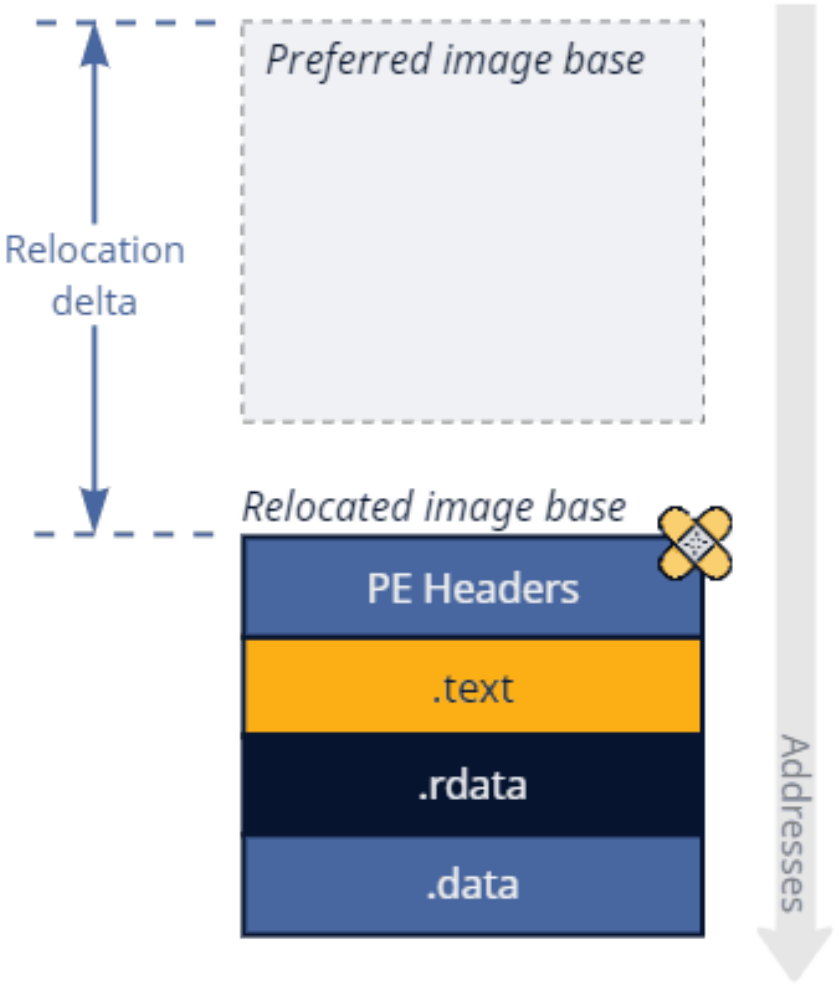

Figure: The meaning of image relocation delta.

Since we want to reconstruct the original file as closely as possible, we also need to undo changes made by relocations. Conveniently, old-style user-mode relocations are effectively irrelevant since they apply on a per-view basis. Unlike in classical memory dumping, we are not reading anything from the process's private view and instead rely on the data cached in the section object. Dynamic relocations do, however, require additional handling. Given a section handle, Native API allows us to query two types of details about a dynamically relocated image via NtQuerySection. SectionOriginalBaseInformation gives us, you guessed it, the on-disk base address; SectionRelocationInformation provides us with a delta between the original and the new bases. We only need one of these values since we can read the new address from the PE headers during parsing. `SectionRelocationInformation` is a better choice because it has been around since Windows 7, as opposed to `SectionOriginalBaseInformation`, which appeared later in Windows 10 RS1.

Continuation

In this blog post, we established **the concepts** required to implement a solution capable of reconstructing the original executable even after an attacker tries to hide the artifacts by erasing the file and overwriting the process memory. In the **next part** of this series, we will focus on the implementation, look more in-depth into the oddities of the PE file layout transformations, learn how custom hand-crafted binaries can break parser assumptions, and figure out how we can overcome these challenges to create a reliable forensic tool.