# Process Doppelgänging – a new way to impersonate a process

By Posted on

Published: 2017-12-18 · Archived: 2026-04-05 17:58:42 UTC

Recently at Black Hat Europe conference, Tal Liberman and Eugene Kogan form enSilo lab presented a new technique called Process Doppelgänging. The video from the talk is available here. (Also, it is worth mentioning that Tal Liberman is an author of the AtomBombing injection).

This technique is a possible substituent of the well-known Process Hollowing (RunPE), that is commonly used in malware. Both, Process Doppelgänging and Process Hollowing, gives an ability to run a malicious executable under the cover of a legitimate one. Although they both serve the same goal of process impersonation, they differ in implementation and make use of different API functions. This is why, most of the current antivirus solutions struggled in detecting Process Doppelgänging. In this post we will take a closer look on how the Process Doppelgänging works and compare it with the popular RunPE.

*WARNING: Running this PoC on Windows 10 may cause a BSOD – the reason is a bug in Windows 10. Details here.*

## Process Doppelgänging vs Process Hollowing (aka RunPE)

The popular RunPE technique substitutes the PE content after the process is created (suspended), but before it is fully initialized. In order to implement this technique, we need to do by our own the step that WindowsLoader took so far: converting the PE file from it's raw form into a virtual form, relocating it to the base where it is going to be loaded, and pasting into the process' memory. Then, we can awake the process from the suspended state, and the WindowsLoader will continue loading our (potentially malicious) payload. You can find a commented implementation here.

The Process Doppleganging, in contrary, substitutes the PE content before even the process is created. We overwrite the file image before the loading starts – so, WindowsLoader automatically takes care of the fore-mentioned steps. My sample implementation of this technique can be found here.

## NTFS transactions

On the way to it's goal, Process Doppelgänging uses a very little known API for NTFS transactions.

Transactions is a mechanism commonly used while operating on databases – however, in a similar way it exists in the NTFS file system. It allows to encapsulate a series of operations into a single unit. Thanks to this, multiple operations can be treated as a whole: they can either succeed as a whole – and be committed, or fail as a whole – and be rolled back. Outside of our transaction, the result of the operations is not visible. It starts to be noticeable after the transaction is closed.

Windows API makes several functions available for the purpose of transactions:

- CreateTransaction
- CommitTransaction
- RollbackTransaction
- CreateFileTransacted
- MoveFileTransacted
- DeleteFileTransacted
- CreateDirectoryTransacted
- RemoveDirectoryTransacted
- etc…

Briefly speaking, we can create a file inside a transaction, and for no other process this file is visible, as long as our transaction is not committed. It can be used to drop and run malicious payloads in an unnoticed way. If we roll back the transaction in an appropriate moment, the operating system behaves like our file was never created.

## The steps taken

#### Usage of NTFS transactions

Firstly, we need to create a new transaction, using the API CreateTransaction.

https://github.com/hasherezade/process_doppelganging/blob/master/main.cpp#L144

Then, inside of this transaction we will create a dummy file to store our payload (using CreateFileTransacted).

https://github.com/hasherezade/process_doppelganging/blob/master/main.cpp#L149

This dummy file will be then used to create a section (a buffer in a special format), which makes a base for our new process.

https://github.com/hasherezade/process_doppelganging/blob/master/main.cpp#L173

After we created the section, we no longer need the dummy file – we can close it and roll back the transaction (using RollbackTransaction).

https://github.com/hasherezade/process_doppelganging/blob/master/main.cpp#L188

#### Usage of undocumented process creation API

So far we created a section containing our payload, loaded from the dummy file. You may ask – how are we going to create a process out of this? The well known API functions for creating processes on Windows require file path to be given. However, if we look deeper inside those functions, we will find that they are just wrappers for other, undocumented functions. There is a function `Zw` / `NtCreateProcessEx` which, rather than the path to the raw PE file, requires a section with a PE content to be given. If we use this function, we can create a new process in a "fileless" way.

Definition of the NtCreateProcessEx:

https://github.com/hasherezade/process_doppelganging/blob/master/main.cpp#L196

Creation of process by this way requires more steps to be taken – there are some structures that we have to fill and setup manually – such as process parameters ( `RTL_USER_PROCESS_PARAMETERS` ). After filling them and wring into the space of the remote process, we need to link them to the PEB. Mistake in doing it will cause the process to not run.

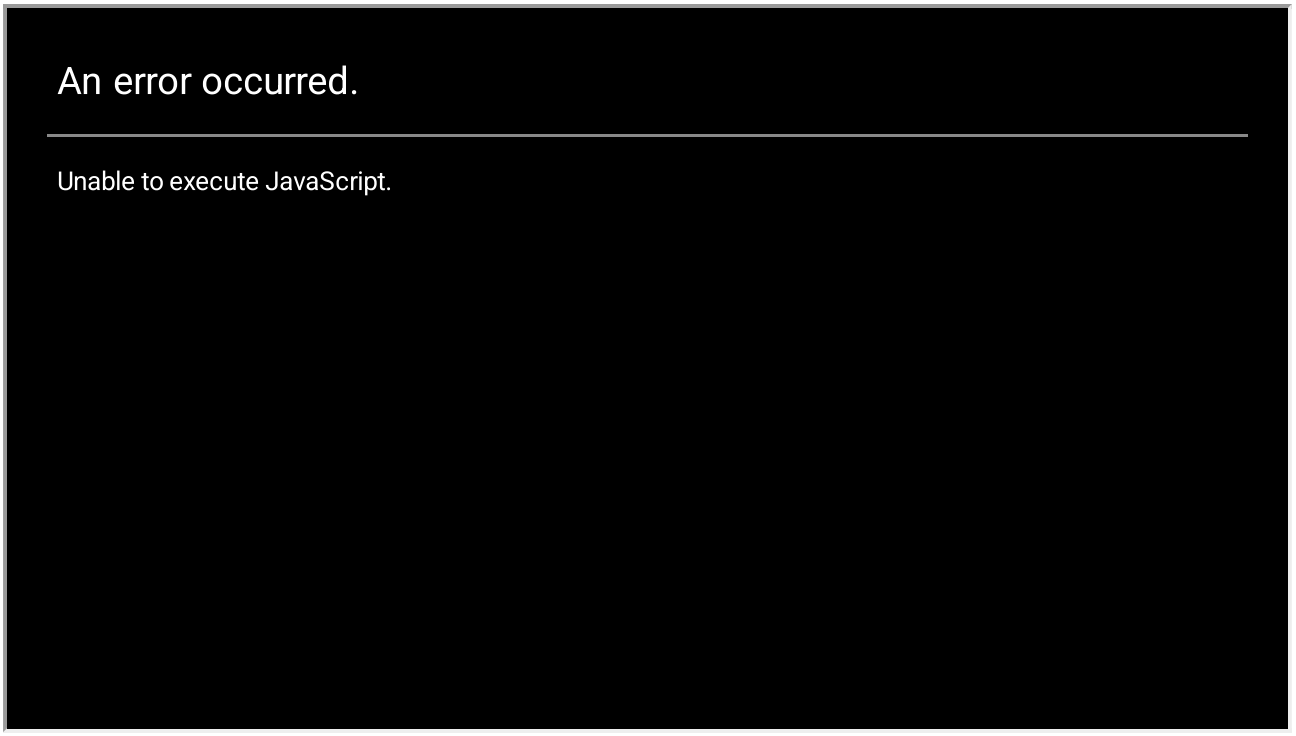
https://github.com/hasherezade/process_doppelganging/blob/master/main.cpp#L76

After setting everything up, we can run the process by creating a new thread starting from it's Entry Point.

https://github.com/hasherezade/process_doppelganging/blob/master/main.cpp#L246

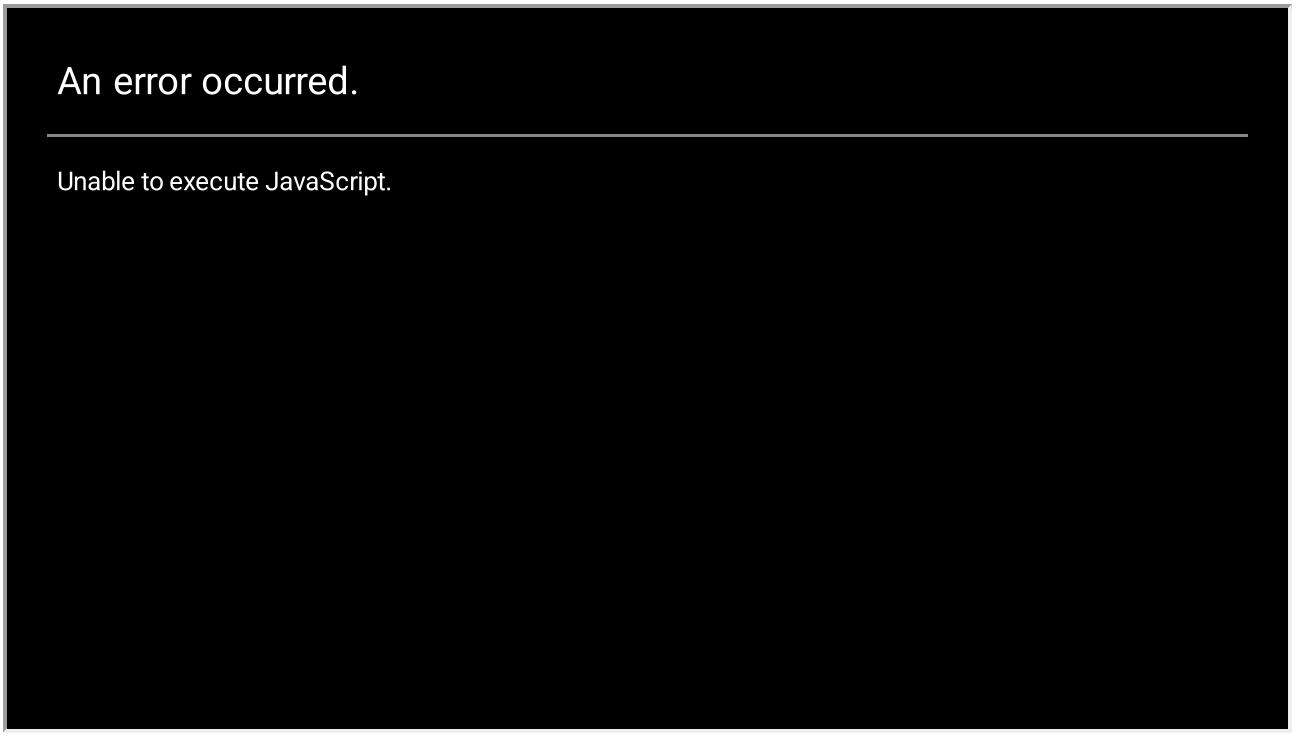
Despite some inconveniences, creating the process by a low-level API gives also interesting advantages. For example, we can set manually the file path – making an illusion, that this is the file that has been loaded, even if it was not. By this way, we can impersonate any windows executable, but also we can make an illusion, that the PE file runs from a non-existing file, or a file of a non-executable format.

Below you can see an example where the illusion was created, that the PE file runs from a TXT file:




**How to detect?**

Although this technique may look dangerous, it can be easily detected with the help of any tool that compares if the image loaded in the memory matches the corresponding file on the disk. Example: detection with PE-sieve (former hook_finder):




The process of injection is also not fully stealthy. It still requires writing into the memory (including PEB) of the newly created process, as well as creating a remote thread. Such operations may trigger alerts.

In addition, the mechanism of NTFS  transactions is very rarely used – so, if any executable call the related APIs, it should become an object of a closer examination.

So far this technique is new, that’s why it is not broadly recognized by AV products – but once we are aware of it’s existence, implementing detection should not be difficult.

---