# Technical Report: Towards Understanding Program Execution Internals and Security in Windows 10

Adam Duby
University of Colorado, Colorado Springs
aduby@uccs.edu

***Abstract*— Linking and loading in the Windows operating system is an opaque process that lacks documentation. The internal data structures and native API must be reverse engineered for developers and researchers to gain insight into the technical details of linking and loading in Windows. In this paper, we document in-depth our reverse engineering findings and collaborate information from various sources to provide what we believe is the most comprehensive and authoritative source on the topic. This importance of this work is in the context of process security. Control flow violations, code reuse, and software exploitation often relies on linking loading DLLs and shared objects in the process address space of a target application. This paper provides an in-depth understanding of the process and insight into security considerations for Windows exploitation.**

## I. Introduction

Research involving process security on the Windows 10 operating system requires intimate knowledge of opaque data structures and processes. Time consuming reverse engineering and debugging is often required to understand the mechanics of process creation, linking, loading, and program execution. This paper was written to document our findings in exploring the internals of program loading with the hopes that it will aid researchers, forensic investigators, and security analysts in understanding the inner workings of a process.

Software must be compiled into a machine code that is compatible with the underlying architecture. Programs written in higher level languages are the input to the compiler, and executable binary code is the output. The binary output file must also be compatible with the operating system. For example, Windows programs must comply with the Portable Executable (PE) file format, while Linux programs comply with the Executable and Linkable Format (ELF). The compilation process is complex, transformative, and lossy. The compiler parses the source code into tokens via the lexical analyzer. Semantics are then represented through some lower level intermediate representation (IR). The IR is typically architecture agnostic, and used by the compiler to perform optimization. Operating system and architecture specific code generation routines are then performed to produce machine code (object code). This object file is then fed into the compile-time linker, which combines the various source code files together and packages the program into one of the operating system's files formats (PE or ELF).

When a program is executed, the operating system's loader loads the program into memory and creates a process address space. The runtime linker builds a linking table to replace the dynamically linked API calls with pointers to the appropriate external library routine. These dynamically linked shared libraries can provide additional functionality and are used to interface with the operating system. Control flow is managed through the instruction pointer (IP), which is a register that points to the next instruction to execute. The IP is manipulated through `call` and `ret` instructions, who use the stack to transfer function parameters and control flow information, such as the return address. User mode programs frequently make traps into kernel mode in order to interface with the operating system. In this paper we investigate this entire process in depth through a combination of reverse engineering tools to follow the execution of some simple programs.

In this paper, we specifically make the following contributions:

- explain process creation and the process address space;
- discuss library linking and loading in depth;
- document our findings from reverse engineering the Windows loader;
- and explain software security concepts such as control flow guard (CFG) and stack protections implemented in Windows 10.

Our motivation for this paper was driven by the countless man hours performing static code analysis, reverse engineering, and debugging to traverse the labyrinth maze of code that is the Windows operating system. It is our hope that this work saves forensic and security researchers time by offering a single reference on concepts that are otherwise lacking in detailed documentation. Clearly, not every concept can be covered in depth in a single technical report. Instead, we cover the fundamentals with enough depth to provide a solid foundation for future work.

## II. The Portable Executable File Format

## III. The Process Address Space

### A. Virtual Address Descriptors

### B. Process Environment Block (PEB)

Each process maintains a data structure of process related information called a Process Environment Block (PEB).

The PEB struct is defined in `winternl.h`, and the struct member fields can be viewed from WinDbg:

```
0:000> dt ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged : UChar
   +0x003 BitField : UChar
   ...
```

We address two ways to access the PEB. A pointer to the PEB of the current process can be found at a 48 byte offset from the value stored in the `FS` segment register in x86 mode or at a 96 byte offset from the value of the `GS` segment register in x64 mode. For example, the following function returns the address of the PEB on a 32-bit process:

```
int * getPEB(){
  __asm {
     xor eax, eax
     mov eax, DWORD PTR fs:[30h]
}}
```

Since the Visual Studio compiler does not support in-line assembly in x64 mode and the offset values can change between versions, the more portable implementation would be to use the native API call `ntdll!NtQueryInformationProcess`, shown below.

```
__kernel_entry NTSTATUS
NTAPI NtQueryInformationProcess(
  HANDLE      ProcessHandle,
  PROCESSINFOCLASS ProcessInformationClass,
  PVOID       ProcessInformation,
  ULONG       ProcessInformationLength,
  PULONG      ReturnLength);
```

When called with `ProcessBasicInformation` from the `PROCESSINFOCLASS` enumeration as the second argument, this function returns a pointer to the PEB.

## IV. DLLs Explained

### A. A DLL Example

### B. DLL Load Count

Each DLL mapped to a process address space maintains a load count, or reference count. A call to `kernel32!LoadLibrary` will either map the DLL to the process address space and set its load count to one, or it will increment the load count if the DLL is already loaded. The count is decremented when the DLL is freed, such as through `kernel32!FreeLibrary`. When the count reaches zero, the library is unmapped from the process. The load count for each DLL is stored in the `LDR_DDAG_NODE` structure of the `LDR_DATA_TABLE_ENTRY`.

## V. Linking

### A. Delay Loading

## VI. The Windows Loader

### A. Enabling Loader Flags

The loader can provide verbose debugging output during the loading process by enabling the global `FLG_SHOW_LDR_SNAPS` flag. This can be done either through the GFlags [1] utility or by setting the `REG_DWORD GlobalFlag = 0x00000002` registry key value in `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<exename>`. This enables loader snap output to the debugger attached to the application. Figure 1 shows a snippet of loader snap output that was piped to WinDbg.

### B. Parallel Loading

### C. Prefetch

The operating system maintains a prefetch buffer for recently executed applications. It increases load time performance by pre-loading libraries and code pages required by the application. For each process, the cache manager the files referenced by the process and maps them to the application's prefetch file. The prefetch can be found in `C:\Windows\Prefetch`, and contains a list of up to 1024 prefetch files with the naming convention `(exename)-(hash).pf`. The prefetch files are compressed, and the `RtlDecompressBuffer` function with the `COMPRESSION_FORMAT_XPRESS_HUFF` parameter can be used to view the plaintext contents of the prefetch files.

### D. Address Space Layout Randomization

## VII. Program Execution

Understanding program control flow and all of the abstraction barriers in operating systems is a necessary prerequisite for further research in code randomization and control flow integrity. Since modern exploits rely heavily on memory disclosure [2], we must understand how the readable memory pages contain valuable information that can be useful for an attacker to craft a runtime code reuse and control flow hijacking attack. Readable information such as function pointers, relative offsets, and return addresses from the stack can glean information about the code layout, enabling the construction of ROP chains that are capable of bypassing some modern defenses.

## VIII. Control Flow Guard

## IX. Software Security

## X. Future Work

## XI. Summary

## References


[1] Microsoft. (2018) Gflags. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 48–62.

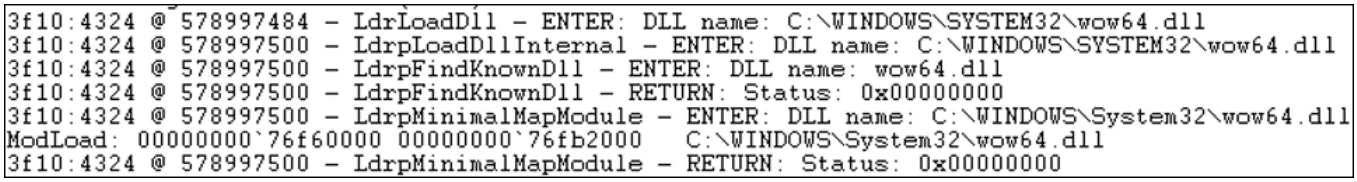


Fig. 1. Example of Loader Snap Debugging Output

[3] J. Choi, Y. Han, S.-j. Cho, H. Yoo, J. Woo, M. Park, Y. Song, and L. Chung, "A Static Birthmark for MS Windows Applications Using Import Address Table," in *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. Taichung, Taiwan: IEEE, Jul. 2013, pp. 129–134. [Online]. Available: http://ieeexplore.ieee.org/document/6603661/