Payload Optimization

19-23 APR 21

Tasking

Develop a repeatable—and ideally automatable—process to reduce payload size.

Current Situation

Last week, we left off with a **3584B** x64 executable, reduced from nearly 100KB when statically linked. The format of a PE image, vastly oversimplifying, is:

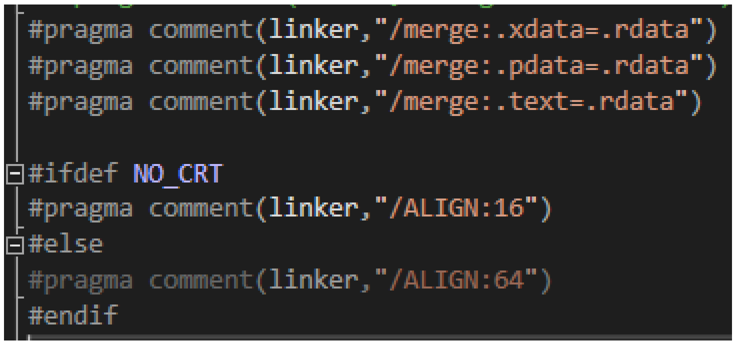
- DOS header (64B)
- DOS stub (128B)
    - "This program cannot be run in DOS mode" message display (64B)
    - Rich header (64B)
- NT header (264B)
    - File header (4+20B)
    - Optional header (128-256B)
        - Image data directory array (n*8B, 0≤n≤16)
- Section headers (m*40B, 1≤m≤n)
- Sections (m*x*size—each section must occupy a full multiple of section size)

Existing Tools

There was a significant amount of fat to be trimmed before modifying these files by hand. So turning to strip.exe (a Windows version of binutils' strip), we can remove the Rich header (64B) but not much else. With UPX ("the Ultimate Packer for eXecutables"), the slack between the headers and the first section is removed, some sections are merged (from 4 to 2, oddly still leaving a section header for a third, zero-size section), and several data directories are removed (reducing the list to export, import, resource, and exception, but still reserving all 16 slots).

Build Optimization

Much of this could be done from the initial build. First we can build against x86 rather than x64 for smaller code and some lifted PE format restrictions. We could also remove the manifest from the PE, taking out an entire 512-byte section and the corresponding section header. From there, we also merge the other sections. Since .idata (the imports list) is already merged in with .rdata., we'll just toss everything into that. This reduces our size by not including section slack for each of these. Thus, you have $0 \leq n_{bs} < 512$ bytes of slack for one section rather than the four that we had previously, and only one section header. A huge debug section (the fault of whole program optimization and link-time code generation) can be removed with /nocoffgrpinfo. Finally, if we modify the section alignment (from the default 512B to 16B in this case), the slack is largely negated. The x86 build PE is now reduced by 2496 bytes (70%) to **1088B** before applying either of the aforementioned tools or manual techniques. Of note, .xdata and .pdata are for exception handling and don't exist in the x86 binary.



```
#pragma comment(linker,"/merge:.xdata=.rdata")
#pragma comment(linker,"/merge:.pdata=.rdata")
#pragma comment(linker,"/merge:.text=.rdata")

#ifdef NO_CRT
#pragma comment(linker,"/ALIGN:16")
#else
#pragma comment(linker,"/ALIGN:64")
#endif
```

Putting It All Together

At this point, strip.exe fails to even remove the Rich header and instead breaks the file. As another unfortunate side-effect, UPX does not work when the FileAlignment is set to less than 512B (it's at 16B right now), although some other optimization causes it to misinterpret the header even when this isn't the case. So…we're solo here. Manually removing the DOS stub takes off 128B, bringing this down to hypothetical **960B**. With the changes made to the build, only the Import (2nd) and IAT (13th) indexes are used, while UPX was able to eliminate anything after the 4th when we could use it. As such, it's possible to reduce the block to only two sections, eliminating 14*8=112B and bringing us to **848B**.

The .idata section is 228B here, made up by 20B of directory table (with one extra blank) and *strlen(impFuncName) + 1 or 2* bytes of name string per DLL, plus 4B (8B if x64) of lookup table per function imported, and a *(2 + strlen(impFuncName) + 1 or 2)*-byte name table per function imported by name instead of ordinal. To import VirtualAlloc and VirtualProtect from kernel32.dll, then, this should take *(20B+14B for kernel32.dll)+(4B+16B for VirtualAlloc)+(4B+18B for VirtualProtect)*=76B. ws2_32.dll is still necessary for the networking functionality, but we could call these by the ordinals it's using with LoadLibrary, eliminating the rest of the imports. Leaving the strings and accounting for the extra code, removing the import table, and removing the final 2 image data directories (2*8B), we could remove about 120B, bringing us to **~728B**. However, at this point we're modifying the behavior of the executable, so this is somewhat out of scope. Similarly, using syscalls and such would also be out of scope.

Conclusion

After some exploring the PE file format and MSVC build options, we're able to reduce an off-the-shelf payload from roughly 100KB (statically linked) to less than 1KB without compromising its operation or performing any "surgery" on the binary. It seems possible, then, to consistently and repeatably apply this process to future capabilities.

Future Plans

1. Create a .props sheet for these optimizations to be added to projects up front.

2. Fix strip.exe and UPX (and maybe others) to handle these binaries.

3. Implement these tools into the build process.