A stealthier approach to spoofing process command line

krabsonsecurity.com/2020/02/23/stealthier-approach-to-spoofing-process-command-line

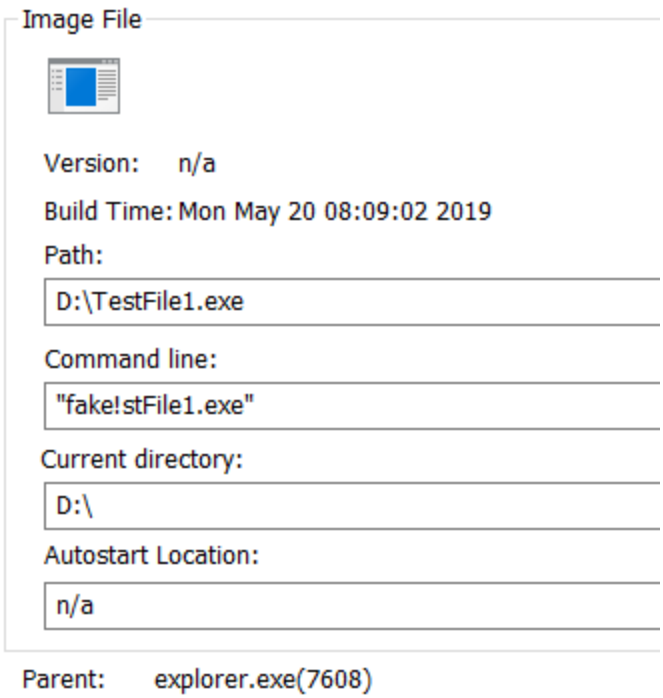
Posted on February 23, 2020

A well known trick that has been employed in malware for a very long time (this has been publicly discussed on HF since at least 2014: showthread.php?tid=4548593) is spoofing command line argument. More recently, this method got discovered by the wider infosec community and was incorporated into tools like Cobalt Strike. Implementations often go like this: you start a process as suspended with a fake command line argument, patch the PEB with the real argument, and then resume the process. However, this solution is not stealthy at all, as we will see below.

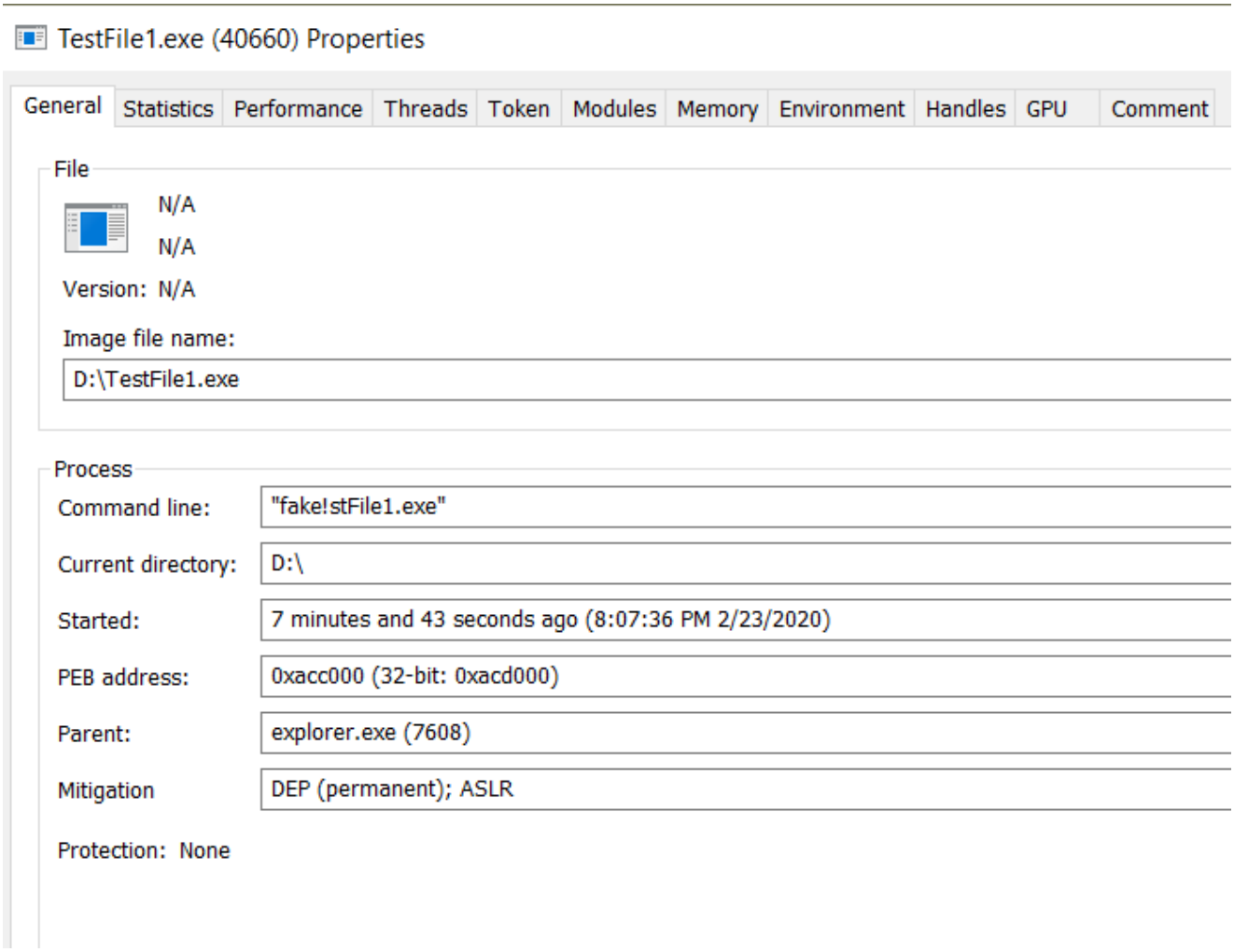
The problem with PEB patching

The method relies on patching the PEB of the process in order to spoof the environment variable prior to the process's start. This works as the command line argument is just a buffer that is passed to the process and can be modified from usermode. However, this causes some problems for those trying to hide it: it is very easy to spot.

As the kernel does not store a copy of the command line argument anywhere, the only place for monitoring tools to get the command line argument is through reading the process's PEB. Both Process Hacker and Process Explorer does this and shows the in-memory command line argument. Below is a simple example where I overwrite the first 5 characters with "fake!" and how it shows up in these tools.



Process Explorer



Process Hacker

Any competent sysadmin would open one of these tools and look at your spoofed powershell-or-whatnot command and recognize it for what it is.

Some work has been done by tampering with the UNICODE_STRING structure through manipulation of the length member (processes typically would ignore this internally when using GetCommandLineW/A APIs, however process explorers generally do not and follow the constraints of UNICODE_STRING). This however only allows us to hide a part of the command line parameter, and does not allow us to spoof the entire parameter. The entire string would also still be there in memory, and as it is virtually always NULL-terminated a tool could decide to ignore the length parameter and read until it reaches a NULL byte instead.

Command line internals

To look for alternatives, we must first understand how processes usually get access to the command line parameter, which is through the GetCommandLineW/A parameter. Probably for performance's sake, the function does not actually access the PEB, instead it reads from a hardcoded pointer to an UNICODE_STRING/STRING.

```
; LPWSTR __stdcall GetCommandLineW()
public _GetCommandLineW@0
_GetCommandLineW@0 proc near
mov     eax, _BaseUnicodeCommandLine.Buffer
retn
_GetCommandLineW@0 endp
```

This is initialized prior to the execution of the entrypoint of the process in BaseDllInitialize (or KernelBaseDllInitialize, as it's now called on Windows 10) and points to the same information that was in the PEB at this stage.

```
mov     dword ptr _BaseUnicodeCommandLine.Length, ecx
mov     eax, [eax+44h]
push    offset _BaseUnicodeCommandLine ; SourceString
push    offset _BaseAnsiCommandLine ; DestinationString
mov     _BaseUnicodeCommandLine.Buffer, eax ; PEB->ProcessParameters->CommandLine.Buffer
call    ds:__imp__RtlUnicodeStringToAnsiString@12 ; RtlUnicodeStringToAnsiString(x,x,x)
test    eax, eax
jl      loc_7C8406F2
```

As a result of this caching, modification of the PEB post-initialization does not affect most process's attempts to read their own command line parameter, so we can't just start a process unsuspended and then patch the PEB. However, this does introduce an interesting side effect: we now have two additional places where we can tamper with the command line parameter without it being visible to Process Hacker and similar tools.

Method 1: Patching BaseUnicode/AnsiCommandLine

As seen in the disassembly above, GetCommandLine simply dereferences and returns a hardcoded pointer. While this address is not exported, we can easily retrieve the relevant offset from the GetCommandLine API itself. Frome Windows XP to 10 (I have not validated this on prior Windows versions, but feel free to), the functions has not changed and is as follow:

```
mov eax, dword ptr ds:[offset_to_buffer]
ret
```

And likewise on x64 environments,

```
mov rax, qword ptr ds:[offset_to_buffer]
ret
```

What this effectively means is that we can get the address of GetCommandLine, parse it and extract the pointer (the pattern is A1 **34570475** C3 for 32 bit processes), patch the pointer there and voila, any subsequent calls to GetCommandLine will return whatever we want it to.

An even neater thing is that everything up until the patching it does not require reading memory from a remote process, as ASLR does not randomize module base address across processes, and as long as you have the same bitness as your target process the address will be the same.

With this method, there are some nuances that needs to be addressed. The first is that from a certain Windows version the function no longer resides in kernel32 but instead in kernelbase. This however is easily addressed and takes no difficulty to deal with. The second is that this needs to be done after the process has started execution (as BaseDllInitialize would just overwrite the pointer otherwise, and kernel32 wouldn't be loaded yet), this can be remedied by patching the entrypoint with EB FE (a jump to self, an useful instruction to remember for unpacking certain malware), fixing the pointers, suspending the thread, restoring the original bytes and then resuming the main thread. It is not really possible to avoid starting the process as suspended unfortunately due to this timing issue.

Method 2: Modifying GetCommandLine

Another place we can intercept is the pointer to BaseUnicodeCommandLine.Buffer itself (rather than BaseUnicodeCommandLine.Buffer). This requires some code patching and is likely easier to detect via hook scanners which compares with disk, however it is still stealthier than modifying the PEB or simply detouring the API with a jmp. The function always take the form of A1 **xxxxxxxx** C3 and we can patch it to point to a buffer that we control. This requires two levels of indirection, and we need to patch it with a pointer *to a pointer* to the command line buffer.

Method 3: Getting BaseDllInitialize to do the dirty work for us

Another option we have is as follow: first start the process suspended, patch the PEB with the to-be-hidden command line argument, **patch the process entrypoint with EB FE**, resume, wait, restore the original command line argument in PEB, suspend the main thread, restoring the original entrypoint and then resuming the thread. This would work as BaseInitializeDll will copy the hidden command line argument for us to BaseUnicodeCommandLine and BaseAnsiCommandLine and does not require us to interact with them directly. This would also hide the command line from Process Hacker and other similar tools as the patched command line parameter is only pointed to by the PEB for a brief moment, and they would have to somehow read at exactly the correct time to obtain this information.

Some final notes on this method

This solution is not going to work with processes that do not use the GetCommandLine API to retrieve it's parameters. However, in practice I have not seen any tools or utilities which directly accesses the PEB to get this information about itself or use another method. If other

methods exists to retrieve this information, it can probably be similarly addressed through some sort of tampering, and if that is not possible, simply hooking the API would normally suffice. Likewise, if the time ever comes when the API itself changes (which I doubt it will), one can simply switch to hooking to return a different command line parameter.

Some possible injection-less detection vectors would be through comparing the pointer in PEB with the pointer in BaseUnicodeCommandLine and checking the integrity of the GetCommandLine function. Alternatively, one could inject into the process and compare the output of the API with PEB. Either way, it is much more complicated to obtain the real command line parameter than it would be if we simply patched the PEB.

PoC

A PoC will be published when I have more time on my hand to clean up my experimental code base. However, the information in this post alone is more than enough for anyone looking to implement the trick to spoof command line arguments in a way that is not easily seen during a red team engagement.

View Comments ...