How to Replicate Emotet Lateral Movement

By Oleg Boyarchuk, Stefano Ortolani

Published: 2022-08-19 · Archived: 2026-04-05 18:45:31 UTC

Emotet is one of the most prominent multi-component threats globally thriving in recent years. Besides the main core module, which is often attached to a spam email or downloaded from a malicious URL, Emotet is known to retrieve from its C2 infrastructure additional modules; these modules can be either designed to propel its own operations by, for example, stealing email credentials to be used in future spam waves, or, when the attack is more targeted, engineered to be more a destructive artifact, like ransomware provided by an affiliated group.

The SMB spreader module belongs to the first category, and it is used to opportunistically laterally propagate East/West to infect additional hosts behind the perimeter. This is a clear step up from the usual info stealers that Emotet had been normally delivering; whenever a malware sample uses the network to its own advantage, researchers need to be able to replicate the network traffic in order to design new defences or update the existing ones.

Unfortunately, reproducing network traffic often depends on additional hosts matching the system configuration targeted by the attackers, an unlikely fit for a dynamic analysis system relying on sandbox technology. Further, as it is the case with threats made of multiple components, running a single module in isolation is often not possible, and it is often necessary to run the artifact in a specific way.

This is how you programmatically load the SMB spreader module and detail the system and network configuration required to replicate the malicious network traffic.

Loading Modules

While Emotet modules are delivered as PE DLL files, they cannot be loaded using standard tools like rundll32.exe or regsvr32.exe as the execution would either crash or silently terminate. The reason is the presence of a custom DLL entry point that requires specific data structures to be loaded in memory before executing the DLL entry point.

```
BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
  if ( fdwReason == 100 )
    sub_18006DBC4(lpReserved, 1047766i64, 678436i64);
  return 1;
}
```

Figure 1: Entry point of a 64-bit update component (3c729151d9d2d326a4a3772ee18a1c0ca5db55ce).

More specifically, as shown in Figure 1, the DllMain function (this holds for all modules since Emotet migrated to 64-bit executables) requires that:

1. fdwReason is set 100.

1. lpReserved points to a custom data structure.

While the first requirement is easy to satisfy, understanding what data structure needs to be passed as a reference is more challenging; a valid approach is to search in the code of the caller for all DllMain invocations, and from there understand what the data structure pointed by lpReserved should look like.

The first hurdle to overcome is how to access the code containing the DllMain invocations; as we explained in a recent blog post, an Emotet DLL payload contains within itself yet another DLL, which is the component responsible for retrieving and loading additional modules. Once the embedded DLL has been extracted (refer to the previously mentioned blog post for more details), we can disassemble it, and search for the DllMain invocations.

In general, there are two possible ways to transfer execution via a call instruction:

- call reg64: call by register
- call qword ptr [reg64 + offset]: call by pointer stored in memory

When looking at the disassembled code of the embedded DLL, there are unfortunately too many instances of call instructions (use the following regular expression *"call\s+(qword ptr \[|r[abcd]x|r[bs]p|r[sd]i|r\d{1,2})"* to find all matches) that are not actual DllMain invocations; however, there are only two functions calls that are compatible with the actual prototype of the DllMain function: "BOOL WINAPI DllMain(HINSTANCE hinstDLL, fdwReason, lpReserved);".

```
__int64 __fastcall sub_180011534(__int64 a1)
{
  int v2; // eax
  __int128 v3; // xmm1
  int v4; // eax
  __int64 v5; // rcx
  __int64 v7; // [rsp+30h] [rbp-40h] BYREF
  __int128 v8; // [rsp+38h] [rbp-38h]
  __int128 v9; // [rsp+48h] [rbp-28h]
  int v10; // [rsp+58h] [rbp-18h]
  int v11; // [rsp+5Ch] [rbp-14h]
  int v12; // [rsp+60h] [rbp-10h]
  __int64 v13; // [rsp+68h] [rbp-8h]

  if ( (*(unsigned int (__fastcall **)(_QWORD, __int64))(a1 + 88))(*(_QWORD *)(a1 + 24), 1i64) )// DllMain(..., DLL_PROCESS_ATTACH, ...)
  {
    v7 = sub_180003798();
    v2 = *(_DWORD *)(a1 + 96);
    v8 = *(_OWORD *)(qword_18002F050 + 560);
    v3 = *(_OWORD *)(qword_18002F050 + 584);
    v10 = v2;
    v11 = *(_DWORD *)(a1 + 40);
    v4 = *(_DWORD *)(a1 + 8);
    v9 = v3;
    v12 = v4;
    v5 = *(_QWORD *)(a1 + 24);
    v13 = *(_QWORD *)(qword_18002F050 + 576);
    (*(void (__fastcall **)(__int64, __int64, __int64 *))(a1 + 88))(v5, 100i64, &v7);// DllMain(..., 100, &LoaderData)
  }
  return 0i64;
}
```

Figure 2: Function of the core DLL (d345c026e88f62044d2eb176192dad649b642911) which is calling DllMain of the downloaded component during the load operation.

These two matches are showed in Figure 2 and Figure 3. While it is normal to see multiple DllMain invocations (the entry point is also called when the DLL is attached and detached for example), only the second invocation in Figure 2 sets fdwReason to 100 (one of the requirements for a successful execution). Turns out that this is the perfect location where to set a breakpoint if we want to take a snapshot of the structure pointed by lpReserved.

```
void __fastcall sub_18001B3B8(unsigned int a1)
{
  __int64 *v2; // rdi
  __int64 v3; // rbx

  v2 = (__int64 *)(qword_18002F050 + 24);
  while ( 1 )
  {
    v3 = *v2;
    if ( !*v2 )
      break;
    if ( !*(_QWORD *)(v3 + 24) )
      goto LABEL_5;
    if ( (unsigned int)sub_180015340(828759i64, 49548i64, a1, *(_QWORD *)(v3 + 48)) )
    {
      v2 = (__int64 *)(v3 + 64);
    }
    else
    {
      (*(void (__fastcall **)(_QWORD, _QWORD, _QWORD))(v3 + 88))(*(_QWORD *)(v3 + 24), 0i64, 0i64) // DllMain(..., DLL_PROCESS_DETACH, NULL)
      sub_18000EAC0(*(_QWORD *)(v3 + 24), 496867164, 445503164, 653983164);
      sub_180028300(927342, *(_QWORD *)(v3 + 48), 719606, 410739, 597560);
LABEL_5:
      *v2 = *(_QWORD *)(v3 + 64);
      sub_180011CCC(689353, 353236, 850033, 290513, v3);
    }
  }
}
```

Figure 3: Function of the core DLL (d345c026e88f62044d2eb176192dad649b642911) which is calling DllMain of the downloaded component during the unload operation.

Once we execute the embedded DLL in a debugger, the break point triggers as soon as the sample is ready to transfer execution to the newly downloaded component, as shown in Figure 4.

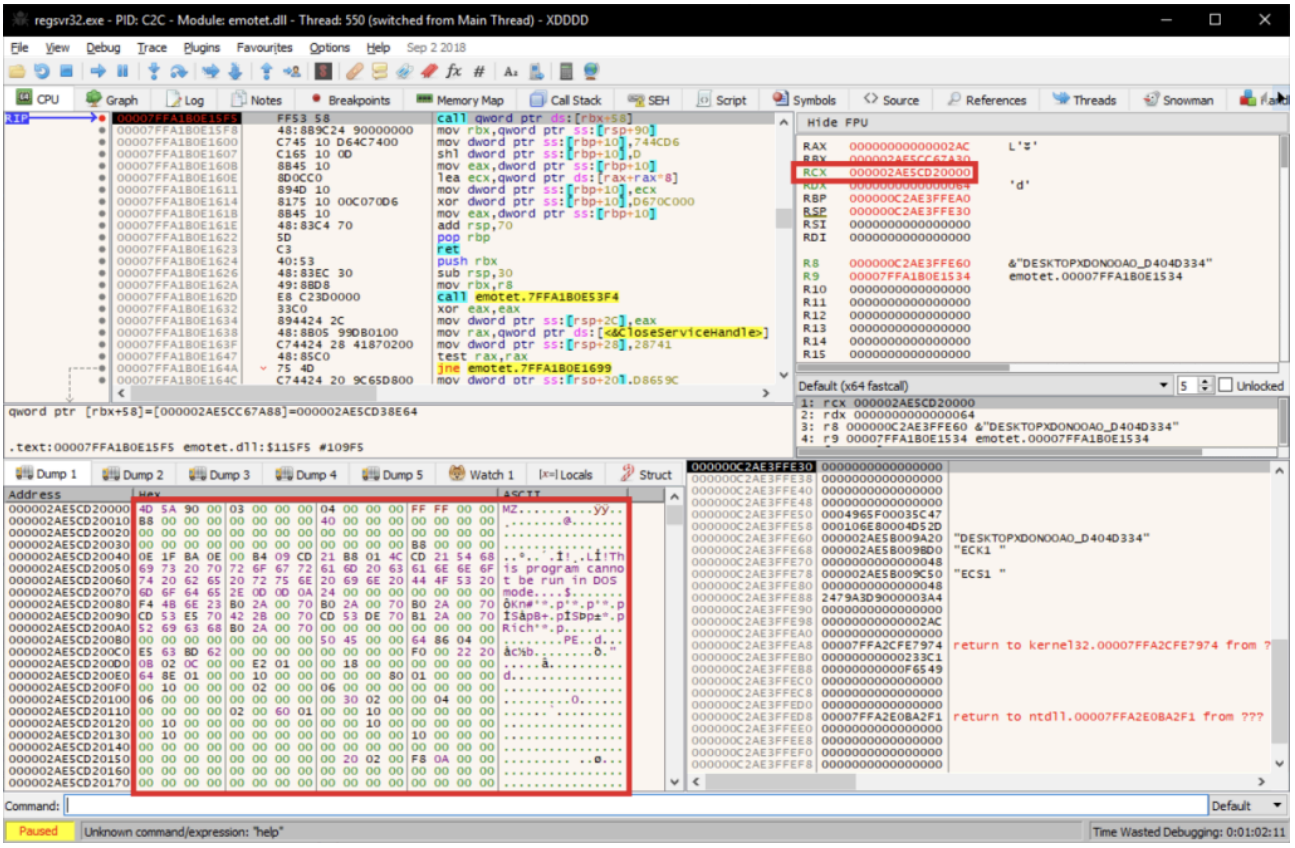

Figure 4: The core DLL (d345c026e88f62044d2eb176192dad649b642911) is about to call DllMain of the downloaded component. RCX register is pointing to the update DLL (highlighted).

We can see immediately the call to the DllMain entry point, and that the parameter hinstDLL is stored in register RCX, fdwReason is stored in register RDX, and lpReserved is stored in register R8. While hinstDLL is just the pointer to the mapped DLL, and fdwReason contains the value that the component expects (0x64, or 100 in decimal), lpReserved points to a data structure holding multiple fields (see Figure 5):

- A pointer to the string “DESKTOPXDONOOAO_D404D334”, a unique bot identifier, obtained by combining the computer name (with dashes replaced by “X” characters) and the C: volume serial number (DESKTOP-DONOOAO and 0xD404D334 in our tests).
- A pointer to a byte array containing the ECK1 key (note that the debugger adds as a comment the first part of the key, which is the string ‘ECK1 ‘).
- 0x48: the ECK1 key size, in bytes.
- A pointer to a byte array containing the ECS1 key (note that the debugger adds as a comment the first part of the key, which is the string ‘ECS1 ‘).
- 0x48: the ECS1 key size, in bytes.
- 0x2AC: a shutdown event handle created by CreateEventA(W).

There are also other fields in this data structure but based on our analysis they do not seem to affect the execution of the module.

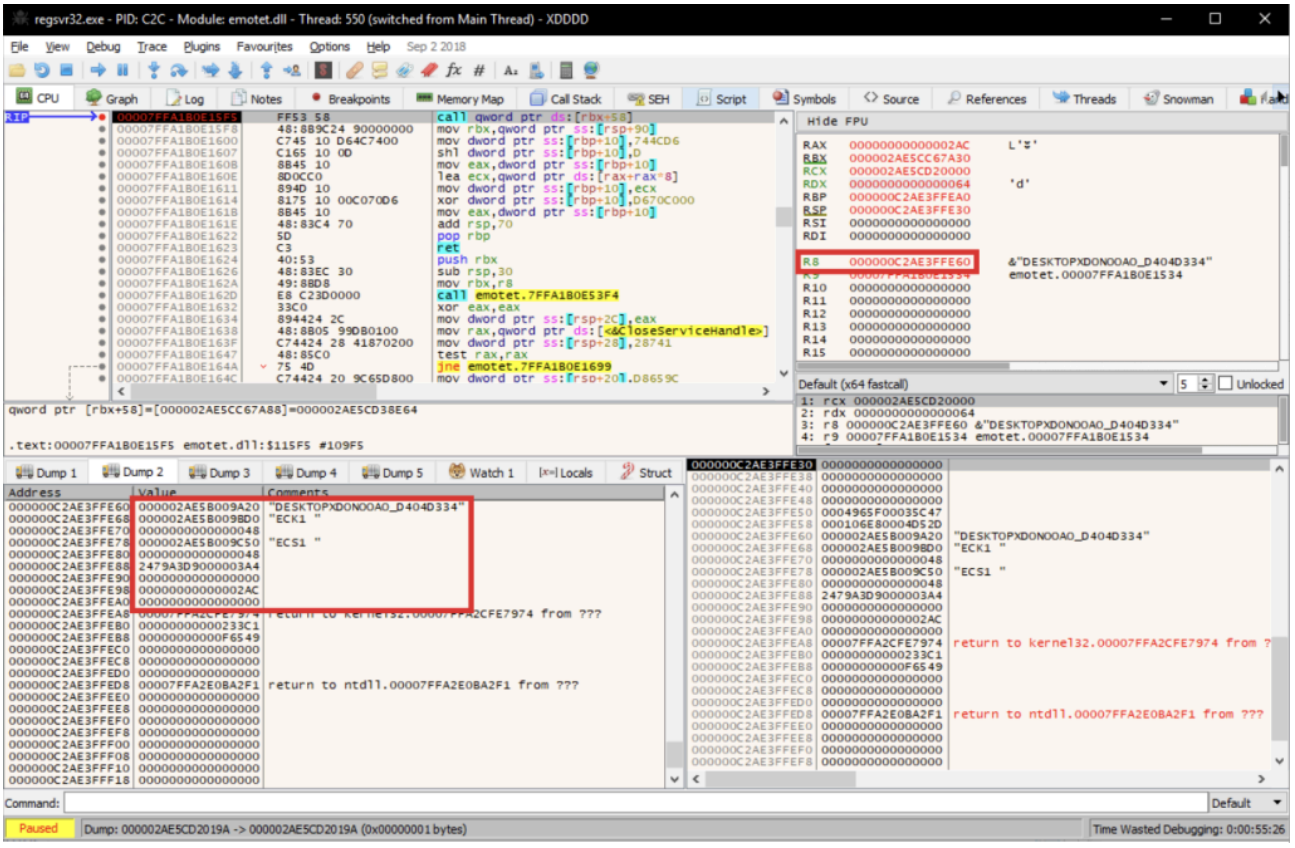

Figure 5: The core DLL (d345c026e88f62044d2eb176192dad649b642911) is about to call DllMain of the downloaded component. R8 register is pointing to the custom data (highlighted) passed from the core component to the update.

Armed with the knowledge of the structure passed by the core DLL to the downloaded module, it is now possible to build a bespoke loader able to (1) load the DLL component, (2) set both fdwReason and lpReserved to the required values, and (3) invoke the DllMain function. The module, receiving this call with the correct parameters, would then trigger the expected behavior.

Note that, in standard circumstances, the core Emotet DLL, as well as any other component executed by the core module, is running in a service hosted by regsvr32.exe under the SYSTEM account, which gives Emotet full privileges on the system. This can be achieved by relying on PsExec to run the loader via the -s argument.

Executing the Spreader

The SMB spreader relies on two different APIs (called iteratively) to enumerate the networks, and the comprised hosts: WNetOpenEnumA and WNetEnumResourceA. While the SMB spreader does not depend on specific resource names (either host names or share names), it is necessary that both APIs return successfully, and that the targeted host is included in the available resources returned by WNetEnumResourceA.

To ease the setup of a fully configured network with a domain controller and a registered client, we used Detection Lab, which has been recently extended to support VMware NSX and vSphere besides VMware ESXi (see the following blog post to learn more on about building one of these detection labs).

In our tests, we discovered that [the example code provided by MSDN] can be used as a simple but effective test to verify this requirement (see Figure 6 for the output confirming that our target host, DC, was indeed available). Surprisingly, we had to enable SMB1 on both the client and the server for this enumeration to succeed (note that SMB1 is not required for the actual binary transfer and execution).

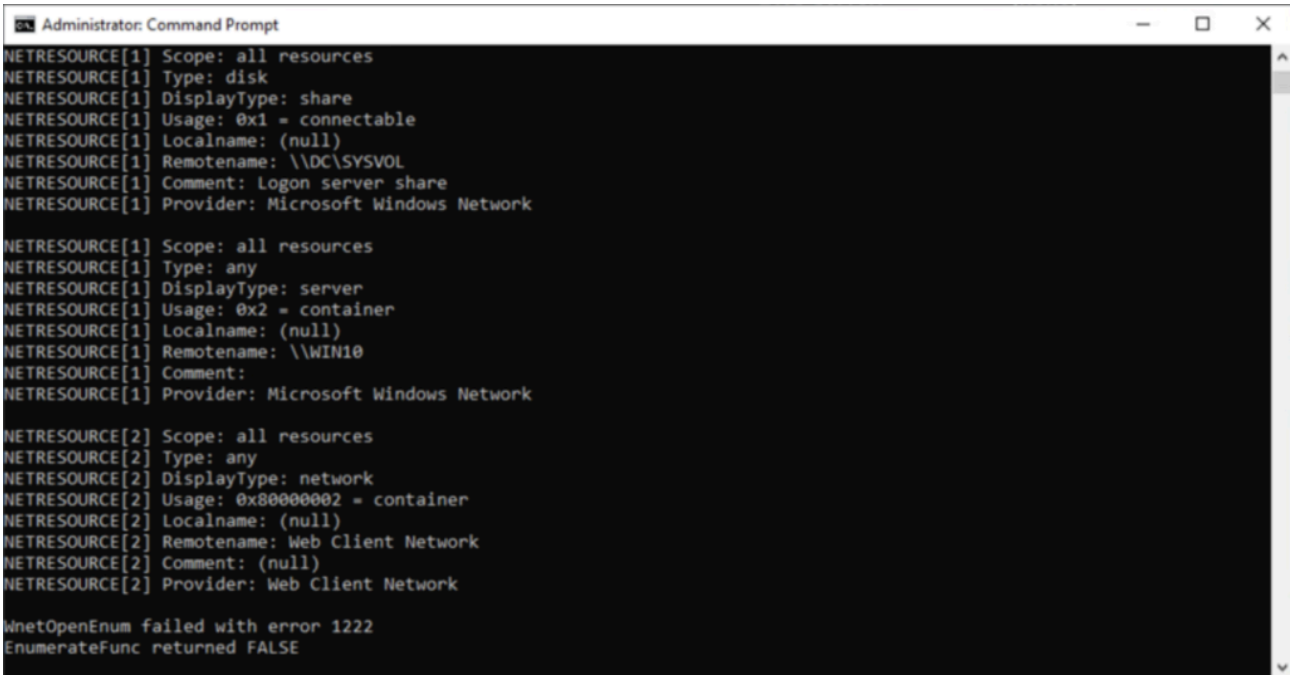

Figure 6: Output of the example code.

Once the network is configured correctly, we can execute the SMB spreader module using the loader described in the previous section. Figure 7 shows Wireshark recording the SMB spreader traffic, which represent the creation and the execution of a remote service (signalling that the SMB spreader is successfully propagating laterally).

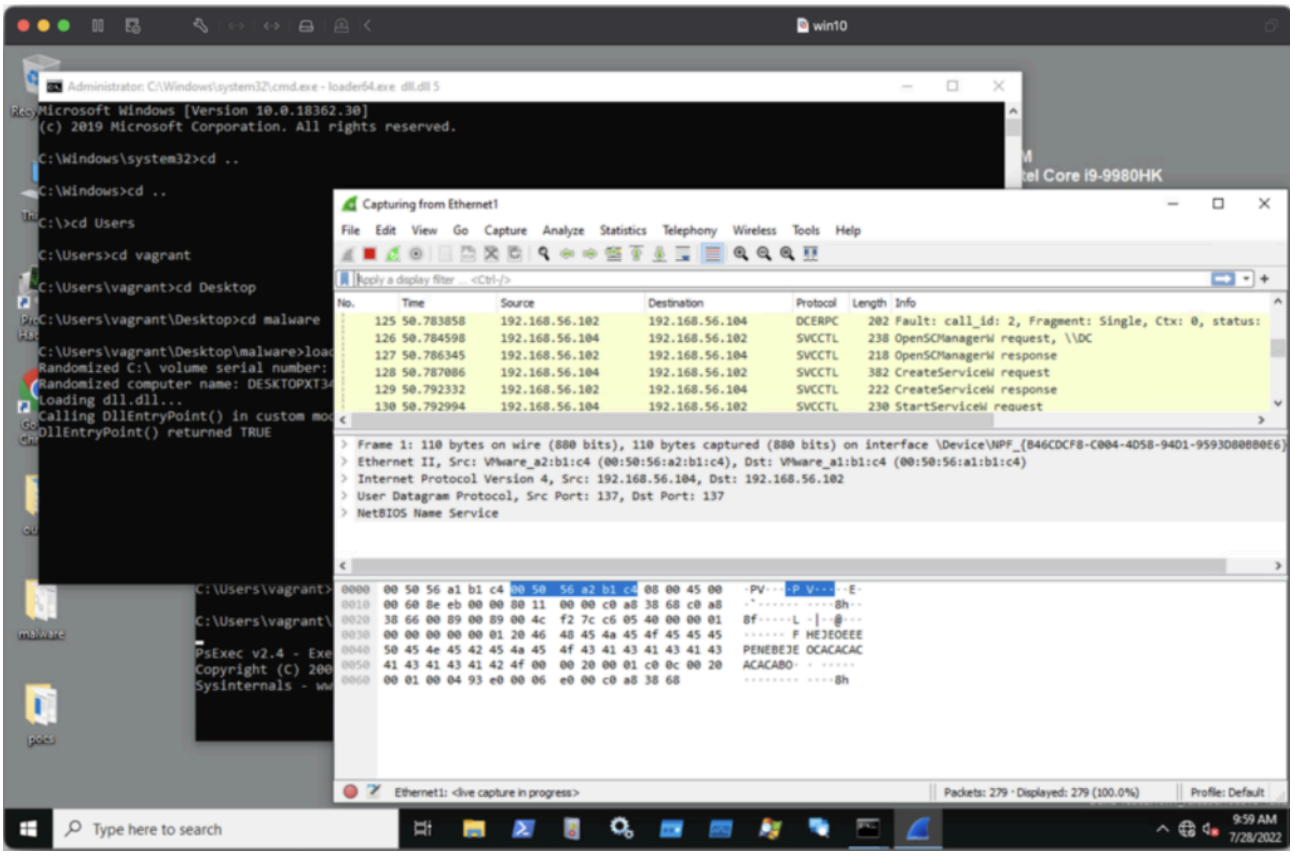

Figure 7: Wireshark running on the source host showing that the malicious service is created and started on the target host.

We can verify that the propagation led to code execution on the target system by verifying that the regsvr32.exe is spawned a few seconds after the creation of the remote service (see Figure 8).

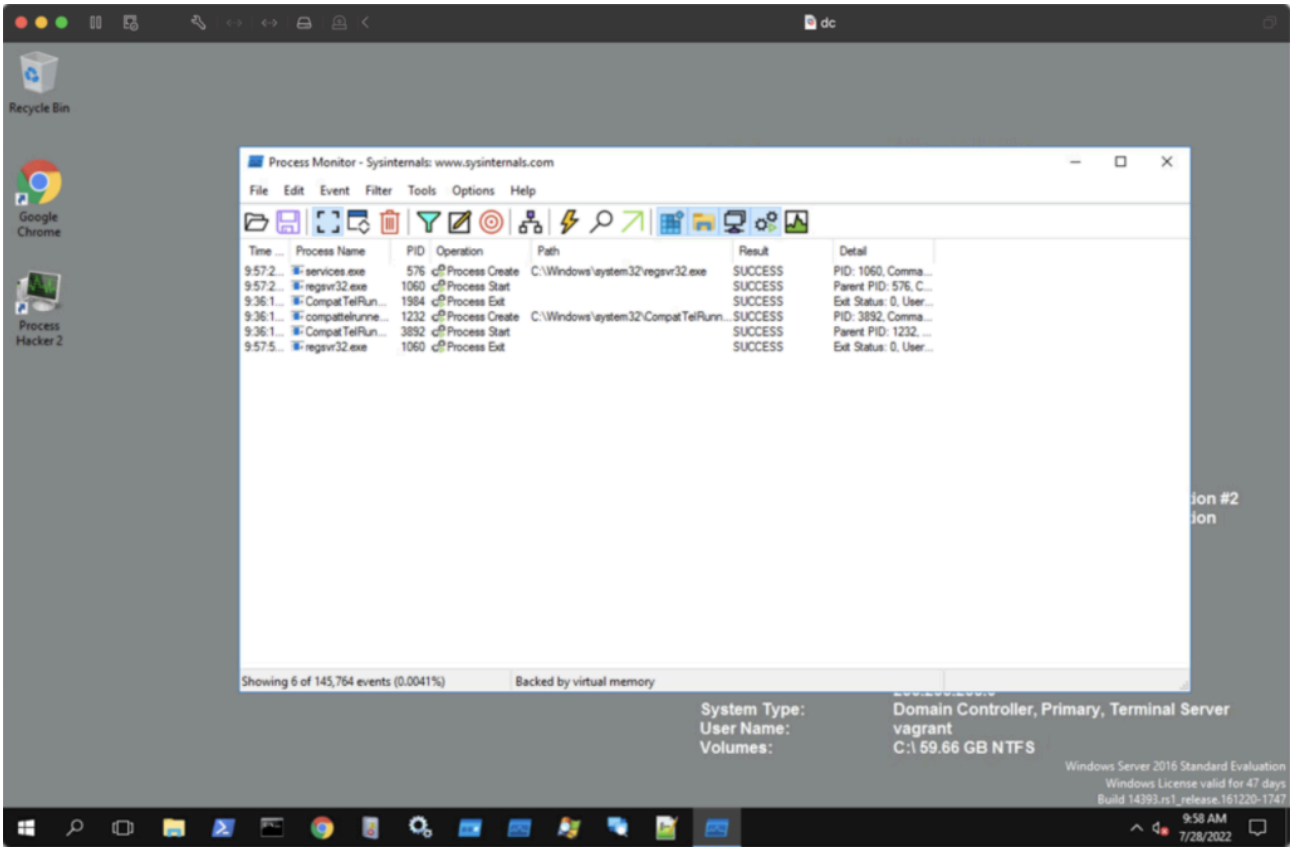

Figure 8: Process Monitor shows that the regsvr32.exe has been successfully started, and that the spreading succeeded.

We collected the resulting traffic in a PCAP that we make available to fellow researchers looking to improve or test their own defences. Note that in our detection lab, since the user was already authenticated against the domain controller, there was no credentials brute forcing as that only happens if no valid credentials are found. The PCAP can be downloaded from our VMware TAU research repository.

Conclusions

Now that the SMB spreader is part of Emotet's toolset, we wonder whether the authors are just diversifying their delivery options, or rather they are exploring how to make Emotet "wormable." Our telemetry seems to support this second option, as we could observe SMB spreader being regularly deployed, rather than being distributed in a few isolated exceptions.

As we mentioned, producing PCAPs from lateral movement's attempts is a challenging endeavour because (1) triggering the propagation logic often requires reverse engineering, and (2) figuring out the right conditions needs extensive domain expertise. On the other hand, lateral propagation techniques are not easily updated nor modified, so it is usually possible to build robust network detection signatures. VMware NSX customers are already protected against this specific lateral propagation technique via a combination of anomaly-based and signature-based detectors.
