Winsider Seminars & Solutions Inc. — Windows Internals Training & Consulting

Faxing Your Way to SYSTEM — Part Two

Yarden Shafir & Alex Ionescu April 30, 2020

Leave a comment

"Part two?", you ask. "Where's part one?", you wonder. In this blog post, we are doing things backwards — first publishing a *Part Two*, with a theoretical "What if?" scenario, and then we'll follow with a *Part One* to fill in our gap.

Posit a DLL Hijack

Let's say you have a way to dump a custom DLL in a privileged directory. You can name the DLL whatever you want and make a privileged process load it instead of one of its own, as part of a privilege escalation attack. This is most useful when there is a process looking for a DLL that is not usually found in the system, so you don't have to implement all the functionality of the DLL you're replacing and/or potentially have to deal with the DLL already being in use . This technique goes under a variety of names such as *DLL hijacking* and *binary planting*, and it's a method that has been known and used for many years. It can also be used a persistence mechanism, when the goal is to load every system start.

Unfortunately, there's not a whole lot of real world public information on actually implementing the technique end-to-end, especially for privilege escalation, without relying on gimmicks. To successfully execute your code, you need:

- A built-in, Windows native, privileged process that tries loading a non-existent DLL from a privileged directory (if it's from an unprivileged directory, you have an even bigger problem)

  - So, not something like an Intel Service or NVIDIA Display Driver — not all users have these!
- A way to *reliably* start the privileged process, from an unprivileged context
  - Online sources resort to gimmicks such as "run these commands in a loop and after 20 tries you'll get `Xxx.exe`" or "and now reboot the machine!"

This really doesn't sound hard, but we could not find anything online that accurately fulfilled these two requirements. So, while in this post, we're not claiming anything novel, we *will* combine some obscure Windows Internals together to weaponize a bind shell (see? we told you it wasn't novel — it's not even a *reverse shell*) with some neat EDR bypasses and forensic gotchas, in order to get some offensive capabilities out in the open and into defenders' mindsets. You'll see (and might learn) how to:

- Identify services that can be started by non-privileged users, so that you can repeat this research and potentially find your own service
- Talk about *trigger started services*, and provide another way to launch services from a non-privileged user account

- Use a previously unused service which is vulnerable to a DLL hijack, which reduces chance of detection, and introduces a reliable escalation vector
- Leverage the Windows Thread Pool API for additional stealth, leveraging arbitrary threads and harder-to-infer malicious behavior, often whitelisted by EDR
- Use some more esoteric, high-performance Windows Socket APIs, which results in less standard imports (no `socket, accept,` `recv`, or `send`) and simpler code
- Abuse the Windows Socket API to hide and misdirect the owner process from Netstat, Process Hacker, Process Monitor, and even WFP (Windows Filtering Platform) and BFE (Base Filtering Engine)-based firewall solutions.
- Escalate privileges from `NETWORK SERVICE` to `SYSTEM`, without any "bean" or "potato"-based DCOM/HTTP attacks
- Launch a process as `SYSTEM` in a non-traditional way using *process reparenting*
- Awesome DLL hijacking in Windows Defender ATP and Windows `21H1` ("Manganese"), for the *lulz*

We will be heavily relying on existing research from other people here, so we want to make sure there is no implied claim that these are hyped-up “never before seen” techniques. We just packaged them up nicely with a bow.

Surveying the Landscape

If you search online, you’ll find four commonly used built-in services (even more 3rd party) on Windows that are vulnerable to a DLL hijack:

1. `Wmiprvse.exe`, which likes to load loads of things from `c:\windows\system32\wbem\`, especially `Wbemcomn.dll`
    1. But it often impersonates the caller when you run WMI commands yourself, so now you need to get a privileged process to issue a WMI command to spawn a WMI Provider
    2. We could not find reliable sources online on how to operationally achieve this **`100%`** of the time
    3. This is a well-known service and target DLL, often abused by malware, and in almost everyone’s PoCs

2. `Ikeext.dll` (running in a `Svchost.exe`) which loads `Wlcsctrl.dll` from `c:\windows\system32\`
   1. This is already running in corporate environments with a VPN — online sources assume you can just `sc stop` it , but that privilege is only granted to `Administrators`.
   2. If it's not already running, you cannot just `sc start` it. The common technique is to use `Rasdial.exe` to trigger it to start.
   3. Extremely well-known, abused in the wild, a dozen blog posts on the topic
3. `Sessenv.dll` (running in a `Svchost.exe`) which loads `Tsmsisrv.dll` from `c:\windows\system32\`
   1. This one has the advantage of not typically running unless you've hit an RDP machine
   2. But it does not grant Start/Stop privileges to unprivileged users and does not have an obvious trigger to start it
   3. Well known and has been abused in the wild for persistence
4. `Searchprotocolhost.exe` and `Searchindexer.exe` will load `Msfte.dll` from `c:\windows\system32\`

1. Cannot be directly started by a non-privileged user, but can often be "triggered" by noisy file-system activity
2. Well known and catalogued, and also used in the wild by APT groups

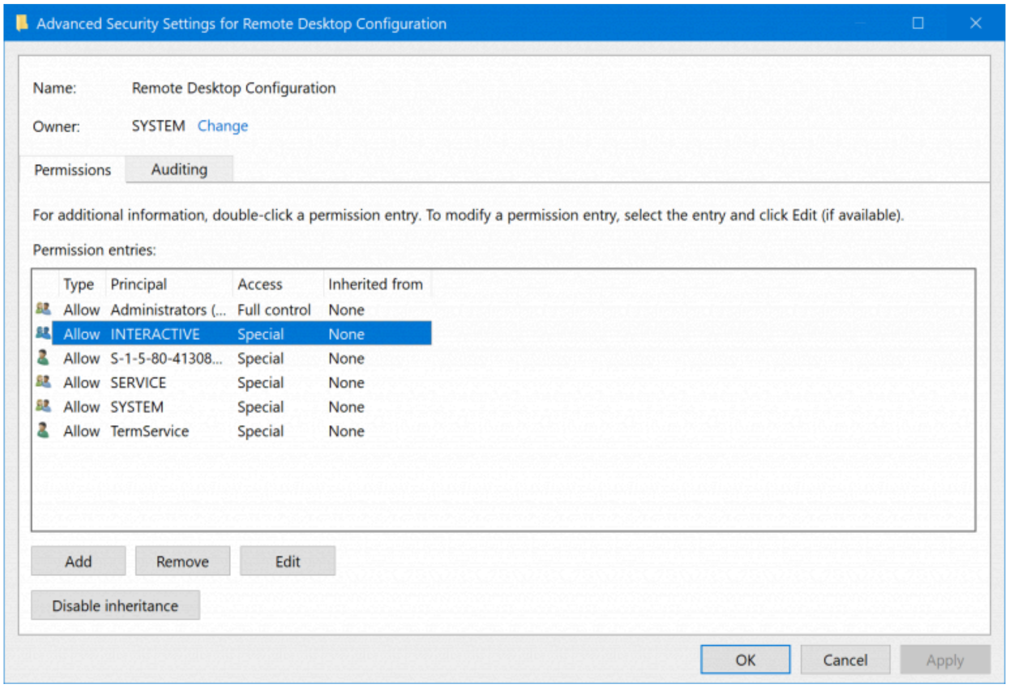
In all of these scenarios, `Administrator` access was already assumed (i.e.: these were mechanisms for persistence, not privilege escalation), or there were unreliable ways to "maybe" trigger the service to start. Additionally, these techniques were known and probably detected by major AV and EDR vendors. We wanted something a little bit more interesting.

Finding the Target — User Startable Services

First, our interest was to identify services that are vulnerable to DLL hijacking attempts other than the afore-mentioned ones. Figuring this out is old & tired infosec practice — run

Process Monitor with the right filters, start a bunch of services (or reboot the box), profit! Countless tutorials online can help you learn how to do this. We applied some different twists, however, which are worth going into. First, remember that a reboot is unacceptable in our use case — we want to elevate privileges *now*. So we had to rely on starting services that weren't already started — or finding a service that can be stopped by a standard user. Second, many online tutorials will have you only looking at `SYSTEM` processes. While that *is* the jackpot, many services run as `LOCAL SERVICE` and `NETWORK SERVICE` — two accounts that while not "privileged" from an `Administrator` Group perspective, can easily elevate to `SYSTEM` using a few different tactics.

Finally, starting a service typically requires administrative permissions, which defeats our purpose (and so does stopping a service in case it's already running). We needed to find exceptions to this rule. There are two great tools for looking into service permissions. One is Process Hacker, which allows you, from its Services tab, to double click on a service, and then click the Permissions button on the General tab. For example, here are the permissions for the `SessionEnv` service:



Name: Remote Desktop Configuration

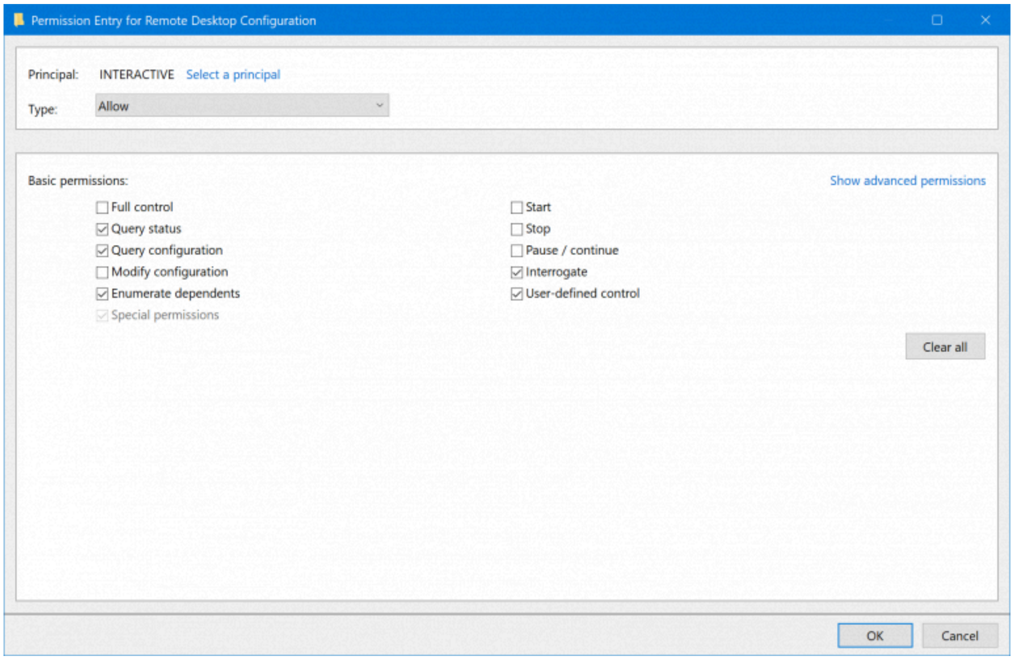
Owner: SYSTEM Change

For additional information, double-click a permission entry. To modify a permission entry, select the entry and click Edit (if available).

Permission entries:

Type	Principal	Access	Inherited from
Allow	Administrators (...	Full control	None
Allow	INTERACTIVE	Special	None
Allow	S-1-5-80-41308...	Special	None
Allow	SERVICE	Special	None
Allow	SYSTEM	Special	None
Allow	TermService	Special	None

Well, already, we see that there's no "`Everyone`", "`Users`" or "`Authenticated Users`", which are common groups that include unprivileged users. But there *is* "`INTERACTIVE`", a less commonly seen group that also includes unprivileged users. Now we can double-click on the ACE and see the following:



Permission Entry for Remote Desktop Configuration

Principal: INTERACTIVE Select a principal

Type: Allow

Basic permissions:

- □ Full control
- ☑ Query status
- ☑ Query configuration
- □ Modify configuration
- ☑ Enumerate dependents
- ☑ Special permissions
- □ Start
- □ Stop
- □ Pause / continue
- ☑ Interrogate
- ☑ User-defined control

So that's not great — all we can really do is query the service and talk to it through SCM control codes.

While nice and graphical, this technique takes time — going down `200` services and clicking a bunch of boxes. So while Process Hacker is great to check one-off services, we wanted a tool to automate this. Enter the venerable Systems Internals Suite, with the `AccessChk` tool. The following command-line is a great way to get a one-line view of all service permissions:

```
accesschk.exe -c * -L > servsddl.txt
```

And you'll have output like this:

```
AJRouter
    O:SYD:(A;;CCLCSWRPWPDTLOCRRC;;;SY)
(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;BA)
(A;;CCLCSWLOCRRC;;;IU)(A;;CCLCSWLOCRRC;;;SU)(A;;CR;;;AU)S:
ALG
    O:SYD:(A;;CCLCSWRPWPDTLOCRRC;;;SY)
(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;BA)
(A;;CCLCSWLOCRRC;;;IU)(A;;CCLCSWLOCRRC;;;SU)S:
AppIDSvc
    O:SYD:(A;;CCLCSWRPWPDTLOCRRC;;;SY)
(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;BA)
(A;;CCLCSWLOCRRC;;;IU)(A;;CCLCSWLOCRRC;;;SU)S:
Appinfo
    O:SYD:(A;;CCLCSWRPWPDTLOCRRC;;;SY)(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;BA
(A;;CCLCSWRPLOCRRC;;;IU)(A;;CCLCSWLOCRRC;;;SU)(A;;CR;;;AU)S:
```

Reading SDDL strings can be a bit challenging, but what we’re looking for specifically is the “`RP`” right, which maps to `SERVICE_START`. And we’d like to see that next to either “`IU`“, which is the “`INTERACTIVE`” group, or “`BU`” for the “`Users`” group, or “`AU`“, which is the “`Authenticated Users`” group, or even better, “`WD`“, which is the “`Everyone`” group. You might even get lucky and find “`AC`“, which is the “`ALL_APPLICATION_PACKAGES`” group.

Once you find an interesting-looking service, say, “`DsSvc`“, you can replace the command-line command with a lower case `l` instead:

```
\sysint\accesschk.exe -c DsSvc -l
[4] ACCESS_ALLOWED_ACE_TYPE: Everyone
    SERVICE_QUERY_STATUS
    SERVICE_START
[5] ACCESS_ALLOWED_ACE_TYPE: APPLICATION PACKAGE AUTHORITY\ALL
APPLICATION PACKAGES
```

```
    SERVICE_QUERY_STATUS
    SERVICE_START
```

So this certainly sounds and seems like an interesting service! The next step is to then run it through the usual suspect — Process Monitor — and try to see any “`NAME NOT FOUND`” errors while looking for DLLs. You need to be a little careful here, as this is something a lot of blog posts don’t talk about: you might find “red herrings”. For example, Windows Defender does lookup a lot of DLL paths, as part of its sandbox/heuristics, but these aren’t actual `LoadLibrary` calls. We’ve also seen services loading `Mfc42.dll`, which looked promising, but a deeper analysis of the call stack showed the `LoadLibraryAsDataFile` function, which doesn’t actually execute code or call any entrypoints/exports.

Since `DsSvc` wasn’t fruitful, we moved on (our search query was to look for “`RP;;WD`“, just to go for the most egregious cases, but there are certainly other candidates too). Next up in our results was:

```
\sysint\accesschk.exe -c fax -l
[0] ACCESS_ALLOWED_ACE_TYPE: Everyone
    SERVICE_QUERY_STATUS
    SERVICE_START
```

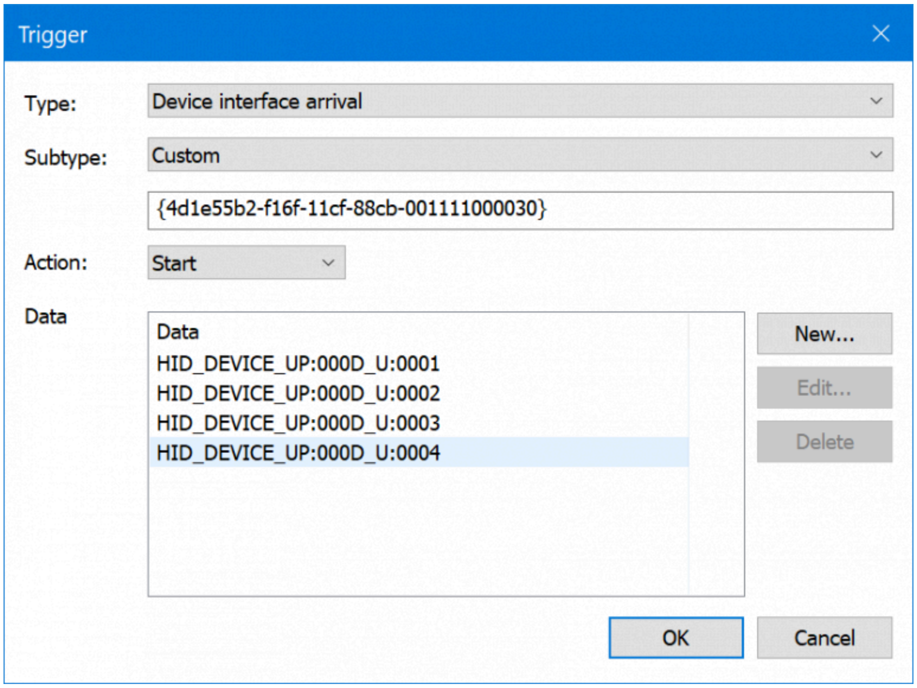
We didn’t know it yet, but we were about to hit a jackpot.

For completeness' sake, the only other 3 built-in Windows services which allow "`Everyone`" to launch them are `icssvc`, `PhoneSvc`, and `TabletInputService`. There are more that allow `INTERACTIVE`, `Authenticated Users`, and `Users`, however.

User Startable Services — Round Two

Before going deep into the `Fax` Service, it's worth talking about another way that a service can be started, regardless of the permissions associated with it. In Windows Vista, Microsoft introduced the Unified Background Process Manager (UBPM), which mimics the functionality of systemd on Linux systems or launchd on macOS — it supports a variety of "triggers", which can be associated with system events such as PnP Device Arrival Notifications, RPC Endpoint Lookups, WNF State Notifications, Socket Connections, or even ETW Events.

The Service Control Manager (SCM) was then updated to allow services to be started based on a trigger, and you can use Process Hacker for a nice GUI view of the triggers that a service has. Here are the ones for `TabletInputService`:



Type: Device interface arrival

Subtype: Custom

{4d1e55b2-f16f-11cf-88cb-001111000030}

Action: Start

Data

Data
HID_DEVICE_UP:000D_U:0001
HID_DEVICE_UP:000D_U:0002
HID_DEVICE_UP:000D_U:0003
HID_DEVICE_UP:000D_U:0004

Device Interface Arrival notifications aren't great, since there's no way to "fake" them from an unprivileged account (as far as we know). But let's take a look at another example, the `DsSvc` service — and let's actually showcase another tool that can dump trigger information: the `Sc.exe` built-in utility itself:

```
sc qtriggerinfo DsSvc
[SC] QueryServiceConfig2 SUCCESS
SERVICE_NAME: DsSvc
    START SERVICE
        NETWORK EVENT                : bc90d167-9470-4139-a9ba-
be0bbbf5b74d [RPC INTERFACE EVENT]
          DATA                       : BF4DC912-E52F-4904-8EBE-
9317C1BDD497
```

What does this tell us? First, the first GUID, labelled as `RPC INTERFACE EVENT` has this to say on MSDN:

"The event is triggered when an endpoint resolution request arrives for the RPC interface GUID specified by ***`pDataItems`****."*

Well, since any user account is permitted to resolve an RPC endpoint, then talking to the RPC endpoint mapper to resolve this GUID will launch the service — even if we don't ultimately have permissions to connect to it. Here's the service currently lying dormant:

```
sc query dssvc
SERVICE_NAME: dssvc
    TYPE               : 30  WIN32
    STATE              : 1  STOPPED
```

And here's us trying to ping the Interface ID that was specified:

```
rpcping -t ncalrpc -f BF4DC912-E52F-4904-8EBE-9317C1BDD497 -v 2

RPCPing v6.0. Copyright (C) Microsoft Corporation, 2002-2006
Trying to resolve interface BF4DC912-E52F-4904-8EBE9317C1BDD497, Version:
1.0
Completed 1 calls in 1 ms
1000 T/S or   1.000 ms/T
```

We can see that the interface replied back to our ping! Let's take a look at the service now:

```
sc query dssvc
SERVICE_NAME: dssvc
    TYPE               : 30  WIN32
    STATE              : 4  RUNNING
                         (STOPPABLE, NOT_PAUSABLE, ACCEPTS_PRESHUTDOWN)
```

Another type of accessible trigger is the ETW Trigger. Here's an example service that uses it, the Windows Error Reporting Service:

```
sc qtriggerinfo WerSvc
[SC] QueryServiceConfig2 SUCCESS
SERVICE_NAME: WerSvc
        START SERVICE
            CUSTOM     : e46eead8-0c54-4489-9898-8fa79d059e0e [ETW
PROVIDER UUID]
```

All it takes is a simple call to EventWrite with the correct ETW GUID, and the service will start. You can do this in C, or even in PowerShell. We modified the linked PS script to use the GUID below instead of the provided one:

```
new Guid(0xe46eead8, 0x0c54, 0x4489, 0x98, 0x98, 0x8f, 0xa7, 0x9d, 0x05,
0x9e, 0x0e);
```

And, sure enough, after launching the script:

```
sc query WerSvc
SERVICE_NAME: WerSvc
    TYPE               : 10  WIN32_OWN_PROCESS
    STATE              : 4  RUNNING
                            (STOPPABLE, PAUSABLE, IGNORES_SHUTDOWN)
```

There's a few other interesting triggers too — and Microsoft documents the official ones here. For example, you'll see that the `IKEEXT` service is spawned by `Rasdial.exe` due to a trigger on UDP port `500` (which you could fake in other ways than launching `Rasdial.exe`).

Abusing Fax

Going back to Process Monitor, when we ran the fax service, we noticed this:

`Fxssvc.exe` was looking for `c:\windows\system32\ualapi.dll` — unsuccessfully. So we placed our DLL in that location, started the service and sure enough, it was loaded into the process!

But then we had a few problems:

1. The service doesn't run under the `SYSTEM` account, but under `NETWORK SERVICE`. This isn't a truly privileged

account, so there's more work to be done.

2. The service looks up some exports using `GetProcAddress`, which it expects to find in `Ualapi.dll`
3. Unless you're actually queueing a fax, the service exits almost as soon as it starts (there are a lot of unfortunately named "`suicide`" variables in the symbols), meaning we can't have persistent threads lying around.

We wanted to solve for `2` & `3` together — normally, malicious privilege escalation attacks leverage `DllMain` in order to perform their next steps, but in our case, the need to elevate to `SYSTEM` makes things harder — plus the fact we want to have an embedded bind shell developed in a smarter way. Secondly, encoding an entire payload in `DllMain` is highly suspicious to anyone disassembling the binary. And finally, `DllMain` is called when the DLL is loaded, which means that the *loader lock* is held, greatly diminishing our capabilities.

Therefore, we skirted the entire problem by not having an entrypoint in the DLL at all, and leveraging the way the `Fax` service calls the `Ualapi.dll`, which you can see in the IDA screenshot below:

Since the service expects all three functions present, we export all of them, and then implement a `UalStart` function where we write our logic — safely away from the confines of the loader lock. Normally we'd have done all of our operational setup here, but we wanted to be sneaky, and leverage the Windows Thread Pool, which affords us some asynchronicity, makes call stacks harder to understand, and brings pain to EDR tools.

The main body of our `UalStart` is actually quite simple:

```
//
// Create the thread pool that we'll use for the work
//
pool = CreateThreadpool(NULL);
if (pool == NULL)
{
    goto Failure;
}

//
// Create the cleanup group for it
//
cleanupGroup = CreateThreadpoolCleanupGroup();
if (cleanupGroup == NULL)
{
    goto Failure;
}

//
// Configure the pool
//
InitializeThreadpoolEnvironment(&CallBackEnviron);
SetThreadpoolCallbackPool(&CallBackEnviron, pool);
SetThreadpoolCallbackCleanupGroup(&CallBackEnviron, cleanupGroup, NULL);

//
// For now, always stay in this loop
//
while (1)
{
    //
    // Execute the work callback that will take care of
    //
    work = CreateThreadpoolWork(WorkCallback, NULL, &CallBackEnviron);
    if (work == NULL)
    {
        goto Failure;
    }

    //
    // Send the work and wait for it to complete
    //
    SubmitThreadpoolWork(work);
    WaitForThreadpoolWorkCallbacks(work, FALSE);

    //
    // We're done with this work
    //
    CloseThreadpoolWork(work);
}
```

It not only provides the benefits of the thread pool evasion/abstraction, but also means that `UalStart` will never return — keeping the `Fax` service from shutting down, and additionally putting it in a perpetual [`SERVICE_START_PENDING`] state, which is unstoppable through regular `Sc.exe` commands. We now have a persistent implant on the system — but we still want to get to a `SYSTEM` shell.

An Elevated Fax

Now that we have our `NETWORK SERVICE` implant, it's time to head on over to `SYSTEM`. When this account was first introduced in Windows XP, alongside its breatheren `LOCAL SERVICE`, the idea was to have service accounts with reduced privileges and permissions, most especially that would not belong to the `Administrators` group.

However, since these are services, they *were* given the `SeImpersonatePrivilege`, which means they can impersonate a more powerful token as long as someone more privilege connects and/or speaks to them, through Winsock, Named Pipes, or ALPC. Technically, this privilege *can* be dropped from a given `Svchost.exe` by using the `RequiredPrivileges` registry value, but few services do so., and as you can see below, `Fax` does not (in fact, it even has the `SeAssignPrimaryTokenPrivilege` too):

Therefore, our initial idea was to open a handle to the `RpcSs` service, which holds handles to lots of different tokens, including `SYSTEM` tokens:

The `Fax` service, which runs in `Fxssrv.exe`, has the impersonation privilege, and therefore we should be able to duplicate one of these tokens and impersonate it, elevating ourselves to `SYSTEM`. Unfortunately, unless you're running Windows XP (i.e.: reading this blog during a BlackHat Advanced Windows Exploitation Course), this simply won't work.

This is due to the fact that since Windows Vista, services have been hardened, as described in the Windows Internals books as well as in this excellent blog by James Forshaw. That being said, over the years, as was shown countless times, the "isolation" between the services did not truly mean much. Multiple attacks were shown, which we'll enumerate and reference here, alongside with mitigations:

- Simply spoofing an endpoint supposedly owned by another service, and getting a `SYSTEM` process to connect, then impersonating it
  - Service SIDs, introduced in Windows Vista, now allow checking that the right service owns/created/is listening on a port.
    - But not every service has mitigations for this
- Finding another service that shares the same Svchost.exe instance, and simply using its own `SYSTEM`-level impersonation tokens, since the handle table is shared
  - Windows `10` Redstone `2` now isolates services in their own separate `Svchost.exe` instances, on systems with over `3.5GB` of RAM
- Opening a handle to another `Svchost.exe` instance which has `SYSTEM`-level impersonation tokens, and duplicating them

    - In Windows Vista, each `NETWORK SERVICE` process has its own Logon ID (LUID), and the process object is ACL'ed such that only `SYSTEM` and the unique per-service Logon ID have access to it
- Opening a handle to a thread in another `Svchost.exe` instance and sending an APC to duplicate a `SYSTEM`-level impersonation token
    - In Windows Vista, the thread objects are all owned by `NETWORK SERVICE`, but use an `OWNER RIGHTS` ACE, also introduced in Vista, in order to strip out any privileged permissions.
- Leveraging loopback network authentication attacks to coerce a more privileged service from authenticating over NTLM with its `SYSTEM` token
    - In 1809, the popular "Juicy Potato" technique was patched — itself a follow-up of "Rotten Potato" and "Lonely Potato"
    - However, abusing WinRM still allows the attack to work on certain systems ("Bean" technique)
- Abusing the fact that the DOS Device Map is shared among all `NETWORK SERVICE` services, and performing a DLL path resolution attack
    - No mitigation

- Leveraging loopback named pipe authentication attacks to trick LSASS into returning a more privileged `NETWORK SERVICE` token
  - No mitigation, and the approach we chose. As always, James wrote another blog post describing this technique.

The idea is simple — while we can't directly open a handle to `RpcSs`, we can create a named pipe, then open it back using the `\\localhost` SMB namespace (instead of `\\.`), and then impersonate it. This will cause the SMB driver to call `AcquireCredentialsHandle` to obtain a `NETWORK SERVICE` token (our current account), which it does by passing in the LUID. In turn, LSASS returns the original token that was created to represent the logon session as whole — which just so happens to be the `RpcSs` token, since this is normally the first service running as `NETWORK SERVICE`. In other words, we just got the same LUID as `RpcSs`, and we can now open a handle to it!

Here's a screenshot of our worker thread's token after impersonating the named pipe. Notice how many more privileges it has, and the new `LogonSession` group it joined:

A SYSTEM Fax

Because we now have the same token as `RpcSs`, we can freely open a handle to it, with all the way up to `PROCESS_ALL_ACCESS`. We then implemented a handle scanning algorithm similar to previous ones demonstrated, but with a few twists that take advantage of more modern Windows functionality:

1. We use the `ProcessHandleInformation` class of `NtQueryInformationProcess` to enumerate the process

handles. Previous research and PoCs brute-forced each possible handle, which is a much slower approach. A few other sources used the `SystemHandleInformation` class of `NtQuerySystemInformation`, which is slower because it enumerates *all* handles – requiring filtering to find the right process.

2. We open our own token, then use `NtQueryObject`'s `ObjectTypeInformation` class to get the Object Type Index for Token Objects (which can vary from version to version, depending on initialization order). This allows us to filter the result list in #1 quickly without calling `DuplicateHandle` and then `DuplicateToken` on every handle, like past sources, nor do we need to do a name comparison on the Type Name.
3. Now that we know we are dealing with a token handle, we also check the `DesiredAccess` field to select only tokens where the granted access mask is `TOKEN_ALL_ACCESS`. This increases the chance that we find highly privileged interesting tokens that we can then impersonate.
4. On most systems, it then only takes us 2-3 calls to `DuplicateHandle` before we find an appropriate `SYSTEM` token.

What do we consider an "appropriate" token, by the way? First, we check the `AuthenticationId` (LUID) to ensure it is `0x3E7` (`SYSTEM_LUID`). Next, we check the `PrivilegeCount` to make sure it is equal to or above `22`, which is the normal amount of privileges that a Windows `10` `SYSTEM` token has – some services run with filtered tokens, so `RpcSs` may impersonate such reduced `SYSTEM` tokens from time to time. We wanted the real deal. Thankfully, both of these checks can be quickly done with the `TokenStatistics` class of `GetTokenInformation`.

Finally, after calling `SetThreadToken`, our thread now runs with a `SYSTEM` token that has all privileges present and enabled:

Armed with this token, we open a handle to yet another service: `DcomLaunch`. Once the handle's been opened, we revert the token back to the original `NETWORK SERVICE`. The short duration of our impersonation, and the fact we merely open a handle and nothing else, helps keep us low on EDR tool's visibility.

So – why `DcomLaunch`? We had two additional operational goals that we wanted to play with. First, we wanted to launch the perennial shell, but without having a `SYSTEM`-token'ed

`Cmd.exe` underneath the… Fax service, sticking out like a sore thumb.

Additionally, we wanted to avoid having to use `SeAssignPrimaryTokenPrivilege` and doing the obvious "impersonate a `SYSTEM` token and set it as a primary process token", so that we could use the sneakier `PROCESS_CREATE_PROCESS` technique. In case this doesn't ring a bell, it essentially relies on the Windows behavior of automatically launching children process with the token of their parent and combines it with the Windows Vista feature of allowing "re-parenting". The link above has James (again!) original presentation on this, which he also describes on a blog post (and related functionality in his PowerShell tools).

This capability means that all Unix-like `fork` behavior (environment variable inheritance, handle inheritance, standard input/out inheritance, and the token duplication) will be based on the chosen parent process, and not the actual creator process. It also evades many EDR solutions that automatically assume the parent is the creator, and ultimately will make it such that `Cmd.exe` will appear in the process tree of the `Svchost.exe` that hosts `DcomLaunch`.

Why did we pick *this* service? Well… just take a look at how its process tree *normally* looks like:

Would you notice another `Cmd.exe` window in all this mess?

Binding to a Socket

For an interactive local attacker, a `SYSTEM` `Cmd.exe` is great for privilege escalation, but a persistent backdoor that allows remote access is a lot more versatile (and a local attacker could bind to it as well).

In the real world, these types of shells are usually setup as "reverse shells" in order to avoid firewall rules around inbound connections. But we didn't want to fully weaponize the entire chain and create a beaconing & C2 infrastructure, so we wrote a simple bind shell instead.

While this isn't novel, we did want to use some Windows Internals knowledge to spice it up a little. First, we continued with our approach of leveraging the Windows Thread Pool API, and used the `AcceptEx` function which has a very different approach to establishing a Winsock connection vs. the usual BSD Socket API:

- Instead of creating and returning a client-side socket after a connection is made, `AcceptEx` expects the caller to have already created the (unbounded) socket and pass in as an input

- Instead of blocking, it pushes a completion packet to an I/O completion port ("overlapped I/O" in Win32 parlance), which can then be associated with a callback function using the Thread Pool API.
- It does not consider the connection accepted (and thus does not wake up the I/O completion port) until at least one packet has been sent by the client – and it returns back what the first client packet's data payload was.
- It automatically fills out the local and remote `SOCKADDR` structures that represent the server and client IP and Port tuple
- It's not directly exported by the Winsock library (`Ws2_32.dll`) because it is a specialized Microsoft Extension. Instead, you must use `WSAIoctl` with `SIO_GET_EXTENSION_FUNCTION_POINTER` to look it up by GUID (this isn't even documented on WSAIoctl's documentation as a valid command!)

As you can see, `AcceptEx` is quite strange – but also quite useful for what we were going for. Therefore, the last step our Thread Pool Work Callback will do is create two sockets

– a listening socket and an unbound socket, bind the listening socket, and pass both as input to `AcceptEx` after looking up its pointer. Looking up the local IP address and building the `SOCKADDR` for `bind` is done using [`GetAddrInfoW`](#) (vs. `gethostbyname`), a more modern and easier to use API, and the sockets are created with [`WSASocket`](#) instead of `socket` – you'll see why soon.

Finally, we pump an I/O completion into the thread pool and then wait for our callback to complete. Now `UalStart` is waiting on the work callback to return, and the work callback is waiting on the I/O callback to return. Thread stacks in Process Hacker won't immediately show anything nefarious going on (such as someone blocked on `accept` from within a DLL), and our operations are spread out over `3` different threads (none of which we directly created).

Creating the `SYSTEM` Bind Shell

Eventually, a client connects to our remote endpoint and sends a packet. At this point, our I/O callback will execute.

The reason we wanted this "send a packet" behavior is to avoid spuriously waking up due to someone doing port scanning and randomly trying to connect to our port. With `AcceptEx`, actual data must first be sent. This, in turn, also gives us the opportunity to validate that the input packet contains the right (expected) connection payload, which in our case is the string `let me in\n` – this made it easier to play with Netcat to test our shell out.

Once we validated the input payload, we can print out the local and remote endpoints with `GetNameInfoW`, another modern API that makes `SOCKADDR` translation to a string easy. But our real goal is to spawn that `Cmd.exe` attached to the accepted socket, reparented under `DcomLaunch`. The simple way of achieving this is as follows:

- Use `STARTF_USESHOWWINDOW` to indicate that `dwFlags` will have window flags, and use `SW_HIDE` to keep the window hidden. Also pass in `CREATE_NO_WINDOW` to make extra sure.
- Use `STARTF_USESTDHANDLES` to indicate that `hStdInput`, `hStdOutput`, and `hStdError` will have valid

handle values, and use the accepted socket handle to allow the other side to drive the shell.

- And, as before, use `EXTENDED_STARTUPINFO_PRESENT` to set the `lpAttributeList` which contains the `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS` that has a handle back to `DcomLaunch`.

And when it works (it doesn't yet), the result should look something like this (do you even notice the `Cmd.exe`?)

However, such a shell will instantly exit. Recall that when reparenting, all `fork` like behaviors, including handle inheritance, will come from the parent, not the creator. And the handles we've passed in as `STDIN` and others *must* be inheritable, and *must* exist… in the *parent*.

Therefore, we must first make sure that the socket handles are inheritable, which is thankfully the default when using `WSASocket` (there is a flag, `WSA_FLAG_NO_HANDLE_INHERIT`, to *disable* this functionality). But, more importantly, we must make sure that the socket exists in `DcomLaunch` – not in `Fax`.

Unfortunately, if you search the Internet on how to duplicate a socket, you'll find the `WSADuplicateSocket` API. This API isn't "hands-free" – the receiving side must actively call `socket` *again*, and pass in a data structure that was returned (and somehow copied) by the sending side. Now we'd have to inject code into `DcomLaunch` and perform other highly suspicious action.

Hold on – if sockets are supposed to be inheritable by default, such that they can be used as input/output handles for a new process, doesn't this mean that the kernel (which handles process creation) can somehow duplicate the socket (inheritance is just another form of duplication) through the object manager, without specialized Winsock APIs? In fact, if you try using `DuplicateHandle` yourself on a socket, you'll see that it works just fine, despite repeated warnings from MSDN and other sources.

That's not to say those warnings or documentation are wrong. Yes, in certain cases, if you have various Layered Service Providers (LSPs) installed, or use esoteric non TCP/IP sockets that are mostly implemented in user-space, the duplicated socket will be completely unusable.

Ultimately, for sockets owned by `Afd.sys`, which is the kernel IFS (Installable File System) implementation of Windows Sockets, the operation works just fine, and the resulting socket is perfectly usable – and has certain perks. Therefore, we must set `hStdInput` to the socket's handle index in `DcomLaunch`, after we've duplicated it (thankfully, `DuplicateHandle` tells us what the resulting handle index is).

Recall that one of the advantages of `AcceptEx` is that it expects the accepted socket handle as *input*, unlike `accept` that returns it after the connection is made. This benefit means that we can actually open a handle to `DcomLaunch` while we impersonate `SYSTEM`, create the local accept socket, and then immediately duplicate it.

Merely duplicating an unbound socket doesn't notify any firewall/WFP/EDR callback, and isn't shown as being attached to anything (as is the case), and it also means that when our I/O callback function executes, we can actually immediately close our side of the accept socket, since the underlying AFD Endpoint is now being referenced by `DcomLaunch` too.

In our implementation, however, we chose to leave the socket alive until *after* we launch `Cmd.exe`, so that we could return error messages back to the client if needed.

Going back to our `CreateProcess` call, there's just one last step before we can use the duplicated socket. If you read various Internet sources on how to bind the shell to a socket, you'll see that the technique works fine when creating reverse shells, but not so much with bind shells (at least, according to Stack Overflow).

PoCs online and various forums suggest that the only way of achieving the intended result is to first create a series of named pipes, have threads pumping all the network I/O through the pipe, and then set the pipes as `STDIN/OUT` for the child process. Wow, that's a lot of work, and we're lazy.

Well, upon further reading, it turns out that the real problem is this: standard terminal handles are meant to be fully synchronous ("non-overlapped"), and `socket` creates overlapped ("non-blocking") socket handles. The solution is to then use `setsockopt` to bring them back to "blocking" mode – or, to leverage the simple fact that `WSASocket` does not have this behavior, unless `WSA_FLAG_OVERLAPPED` is passed in, which is not the default, but which our code *was* using.

You see, what's tricky is that `AcceptEx` itself is an Overlapped I/O API – that's why it works with our entire thread pool based approach. So not passing in `WSA_FLAG_OVERLAPPED` means that we can no longer use the API, or a thread pool, or the entire approach we're going for. That said, once again, the benefit of `AcceptEx` separately accepting the other socket (the one that will be bound to the client, and duplicated into `DcomLaunch` to serve as the `STDIN/OUT` handle) as input is a life saver. We can create the *listening* socket as overlapped, and then create the *accepting* socket as non-overlapped, having our cake and eating it too.

As last, we now combine everything together and have a functional `CreateProcess` call which creates a hidden `Cmd.exe` that's bound to the client socket, and the client can start manipulating our remote machine. Now sounds like about the right time to dump a demo screenshot to get that conference applause.

But, this blog post isn't quite 6000 words yet, so we're not done with the Windows internals, as there's a few extra tidbits.

Duplicated Sockets and Evasion

First, if you use `Netstat` with the “`-b`” flag, or Process Hacker, or Process Monitor, you’ll not see a single socket inside of `DcomLaunch`. Indeed, the entire connection still appears as if driven by from `Fxssvc.exe`. Even better, if we’d allow the Fax service to exit (which we didn’t want in our implementation), `Netstat` will show `System` and Process Monitor seems to completely hide the network I/O. Additionally, any BFE or WFP-based tools will see traffic as if coming from `Fxssvc.exe`, and Windows Firewall rules will apply to *that* process, and not `DcomLaunch`. Look at this screenshot below, of our `Netcat` connection above:

This behavior is due to a glaring oversight in allowing `DuplicateHandle` on sockets but not fully making `Afd.sys` capable of correctly handling the security implications. Ultimately, because the AFD Endpoint is the same, the duplicate handle is just an additional reference – and all ownership of the socket still belongs to the original creator – even when the creator exists (and actually, because `Netio.sys` is still referencing the original `EPROCESS`, the creator and the PID become "zombies" and leak resources).

Here's Windbg showing `Fxssvc.exe` and its reference count while it's running:

And here it is after terminating the process — notice how there's still 8 leaking references:

This behavior was actually discovered and told to us by a good friend – the creator of Process Hacker. It was submitted to Microsoft years ago, but – stop us if you've heard this one before – it's not a security boundary, it's *by design*. Certainly, a design which all EDR/Firewall/DFIR vendors all know about, since it's so clearly documented, right?

The last internals behavior we use is in how we send data back to the client in error situations (a lot can go wrong with creating our `Cmd.exe`) – we don't use the `send` API. Instead, we use yet another "lookup-by-GUID" functionality of Winsock 2.2, which is `TransmitPackets`. This is a more generic version of `TransmitFile`, an API that once got Microsoft in trouble, for building end-to-end file transfer directly into the kernel, which was once considered anticompetitive and dangerous (these days, Linux has exactly the same functionality).

`TransmitPackets` allows you to specify a set of virtual addresses — or file handles — and has a dozen flags to fine tune how this data should be sent – including through worker threads (the default) or through Kernel APCs (the faster way). We thought it'd be fun to use it, which again makes the payload import less obvious socket APIs, makes analysis a bit harder, and has a minute performance again in the off chance there's an error packet to send. It also avoids LSPs or other EDR hooks on traditional APIs like `accept`, `recv`, `send`, `socket` — and even the IOCTLs sent to `Afd.sys` are different.

Putting this all together, we now have our I/O callback calling `WaitForSingleObject` to wait for the `Cmd.exe` to exit when the client disconnects. We're good citizens and use the `CallbackMayRunLong` thread pool API not to hold things up — note that we *could* have used the `WaitCallback` functionality of the thread pool, to asynchronous be notified when the shell exits, but that would've added more complexity that at this point just wasn't worth it.

Once the `Cmd.exe` terminates, the I/O callback completes, which then wakes up the work callback, which then wakes up the `UalStart` thread. In our code, it goes back into a loop, and starts the whole operation again. Certainly, we could've cached a bunch of data to make this easier, but we opted for the simpler approach. And you could also make it to that `Fxssvc.exe` exits and this while logic is hosted somewhere else, or etc., etc., etc. We're not *actually* NSA operators, so we'll leave that to the real implant writers.

A last note on this: if you like using this unknown DLL but, unlike us, don't mind restarting the machine, you can always restart and let `Spoolsv.exe` load `Ualapi.dll` when it starts running. This process starts on boot and runs as `SYSTEM`, which saves us a lot of the work — in that case we will just need to open our bind shell:

Of course, most people do notice when their computer restarts out of nowhere. And if you plan on waiting for the machine to restart for an unrelated reason (update, crash, etc.) you might be waiting a very long time, as many servers only go down a few times a year for a scheduled update and neither of us can remember the last time we restarted our computers. But hey, maybe you're playing the long game. We don't judge. Much.

ATP Bonus Round

This was a lot of reading and effort for a simple DLL hijacking attack. Maybe you just want something *a lot* simpler. And not have to worry about custom exports and a funny named DLL. Well, Windows `10` provides exactly what you need, and takes you straight to `SYSTEM` without any of this work. How could something like this work? Well, you've probably heard of Windows Defender ATP. What you might not know is that "ATP" stands for "Accommodating To Planting".

In fact, every single DLL that it loads suffers from a load ordering issue, where the current directory takes precedence over `System32`. But that's OK — this is clearly a 3rd party tool, not from a security-focused team, and understanding the internals of load ordering is hard, so we can be understanding:

Of course, things aren't as easy as they might seem at first, as ATP does have a number of mitigations in place to avoid nonchalant abuse of this behavior:

- The Service Control Manager (SCM) will start it as a Windows Protected Process Light (PPL) which will require your DLL to be Microsoft-signed (or some PPL/signature bypass, like the ones shown at Recon 2019 by James Forshaw and Alex).

- `Mssecflt.sys` / `Sgrmagent.sys` have capabilities to detect this type of attack, in combination with Windows Defender and System Guard Runtime Monitor Attestations (Octagon).

That being said, using the PreferSystem32Images mitigation would certainly clean up this behavior.

Windows Manganese (21H1) Post-Credits Scene

OK, OK, let's stop making fun of the OS Vendor's EDR tool. The team was acquired, not native to Microsoft, and DLL hijacking isn't even a security boundary. It's not like the OS itself would ever have issues like these… right? Right??? Continuing in the tradition of ever-increasing quality and static analysis tools and totally-not-throwing-the-SDL-out-the-Window, the next version of Windows `10` just adds a built-in DLL planting vector to every privileged process — `EdgeGdi.dll`. The latest builds now hard-code loading this DLL directly into `Gdi32.dll` — a fact which we noticed alongside @decoder_it on Twitter:

Yep — a new function `CheckIsEdgeGdiProcessOnce` was added — which makes every GUI process now vulnerable to this DLL planting attack. Ah, security… why even bother?

Show Me The Code!

We've implemented the end-to-end functionality described here in our GitHub project `Faxhell`, which is a pun on the pronunciation of the word "Fax" (Facs) and "Shell" — while also making the words "Fax Hell". Because Alex likes naming things in silly ways.

Read our other blog posts:

- Secure Kernel Research with LiveCloudKd
- Troubleshooting a System Crash

- KASLR Leaks Restriction
- Investigating Filter Communication Ports
- An End to KASLR Bypasses?
- Understanding a New Mitigation: Module Tampering Protection
- One I/O Ring to Rule Them All: A Full Read/Write Exploit Primitive on Windows 11
- One Year to I/O Ring: What Changed?
- HyperGuard Part 3 – More SKPG Extents
- An Exercise in Dynamic Analysis

Yarden Shafir & Alex Ionescu

April 30, 2020 Windows Internals

Next Post—

PrintDemon: Print Spooler Privilege Escalation, Persistence & Stealth (CVE-2020-1048 & more)

—Previous Post

Symbolic Hooks Part 4: The App Container Traverse-ty

Leave a comment

You must be logged in to post a comment.