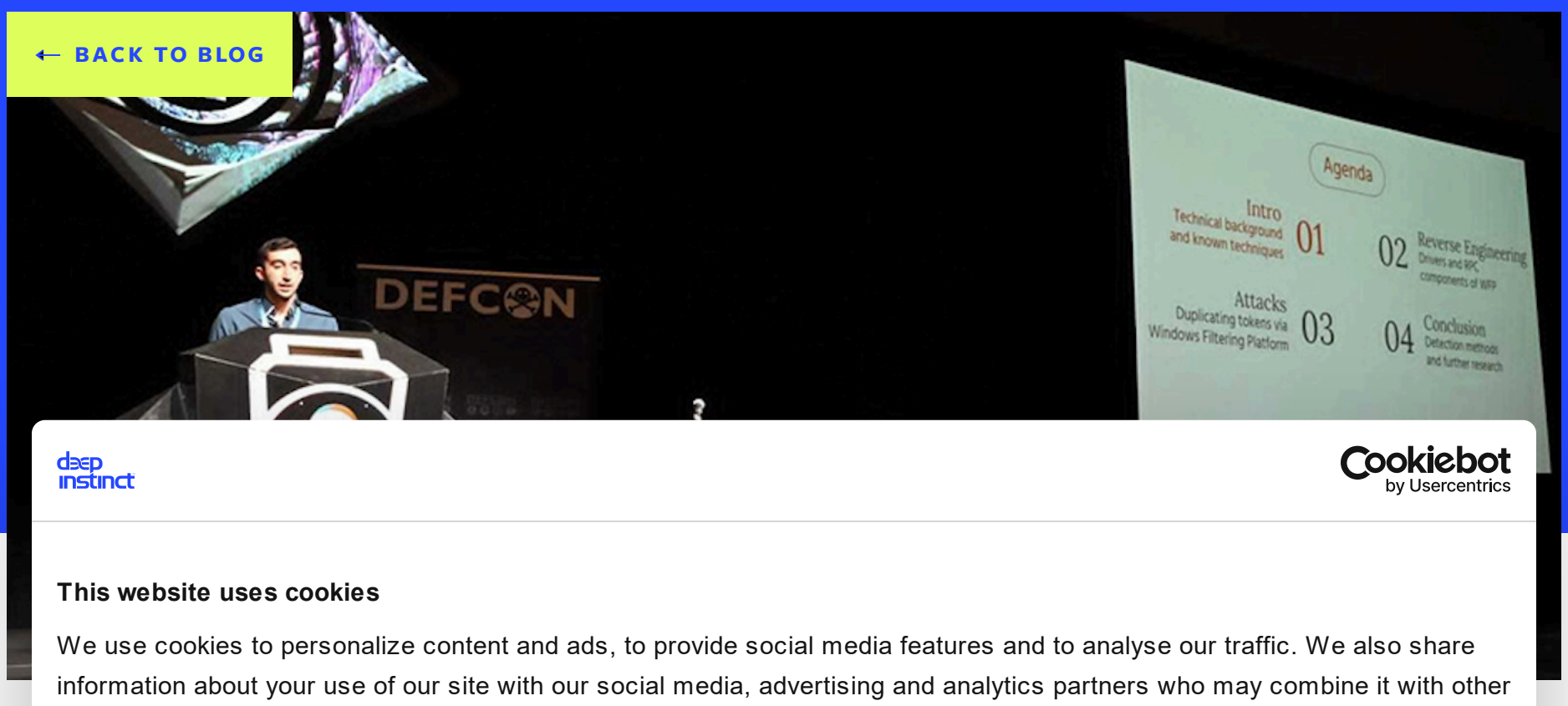


Intro

This blog is based on a session we presented at DEF CON 2023 on Sunday, August 13, 2023, in Las Vegas: #NoFilter: Abusing Windows Filtering Platform for Privilege Escalation

Privilege escalation is a common attack vector in the Windows OS. There are multiple offensive tools in the wild that can execute code as "NT AUTHORITY\SYSTEM" (Meterpreter, CobaltStrike, Potato tools), and they all usually do so by duplicating tokens and manipulating services. This allows them to perform attacks like LSASS Shtinkering.

This talk showcased an evasive and undetected privilege escalation technique that abuses the Windows Filtering Platform (WFP). Additionally, the various components of the Windows Filtering Platform were analyzed, including the Basic Filtering Engine, the TCPIP driver, and the IPSec protocol, while focusing on how to abuse them to extract valuable data. This blog digs into the specifics of the session.

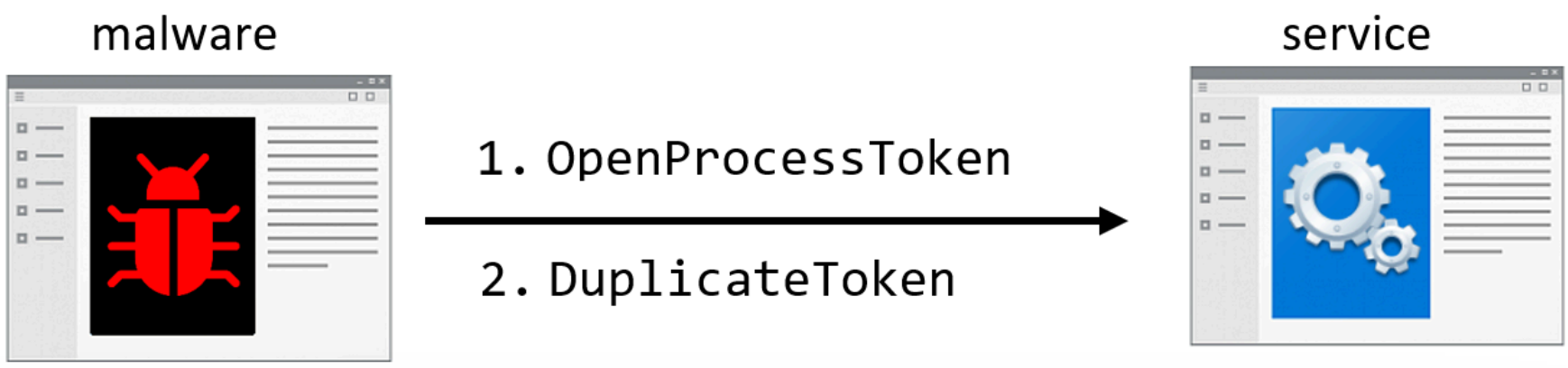
Access Tokens Background

An access token is a representation of the security context for processes and threads. When a thread executes a privileged task or interacts with securable objects, the access token serves to identify the user involved. It details the identity of a process and it is composed of

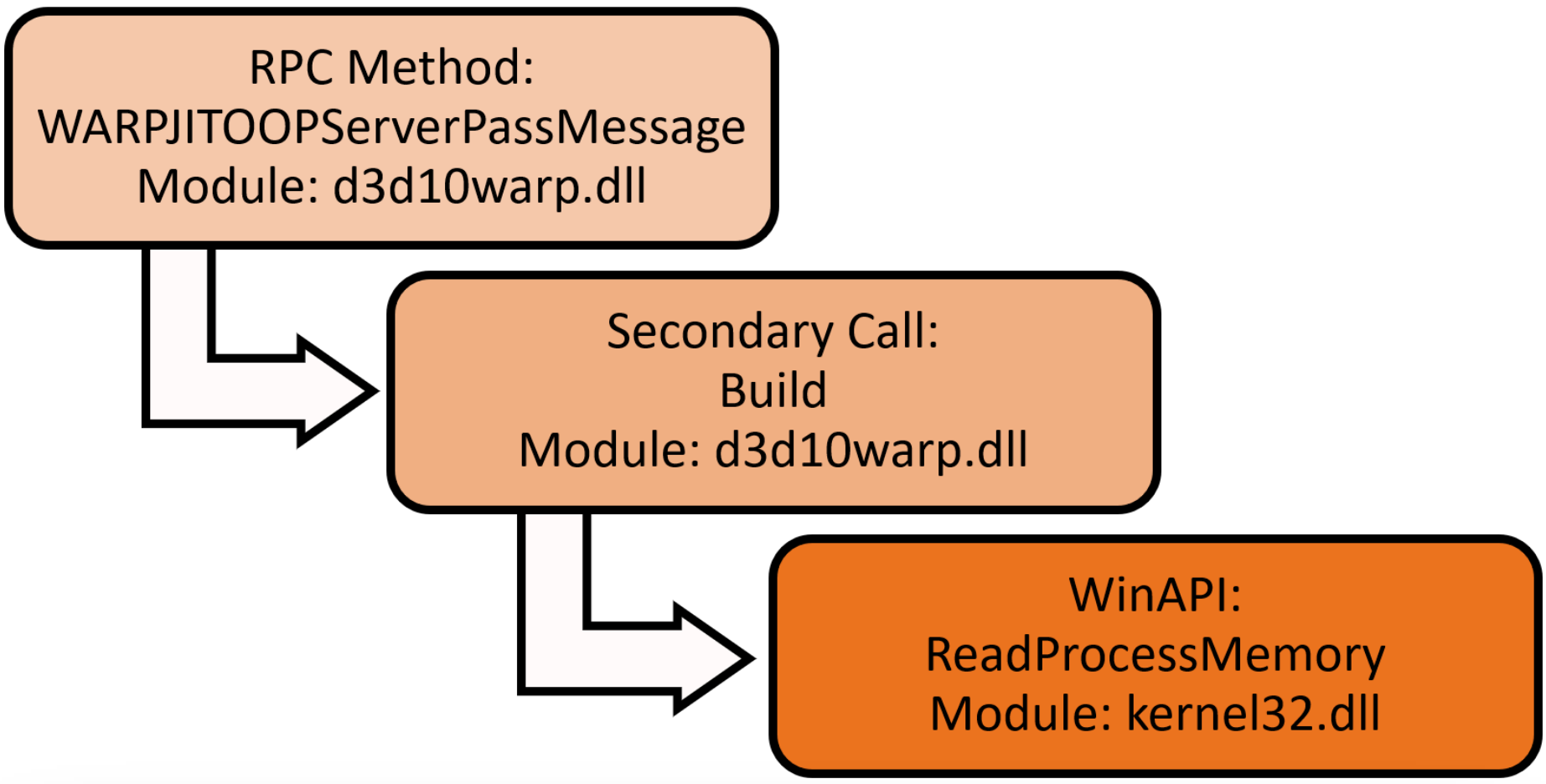
several things: the user that executed it, the groups and log-on session it belongs to, and the privileges of the process. When a thread tries to access an object like a device or a mutex the security identifiers of the token are checked to see if the access is allowed.

There are two types of tokens: primary and impersonation. Primary tokens describe the security context of the user account associated with the process. Impersonation tokens give threads the ability to execute under a different security context than owning process. They are used to represent the security context of clients connecting to a server application.

The tokens of other processes can be accessed by calling DuplicateToken or DuplicateHandle. Also, threads can gain high-privilege impersonation tokens by manipulating RPC and COM servers running as "NT AUTHORITY\SYSTEM" and then calling APIs like ImpersonateNamedPipeClient, CoImpersonateClient, or RpcImpersonateClient.



All the RPC servers on the system were mapped and methods were marked if the parameters that will be sent to the WinAPI are controlled by the RPC client. The WinAPI could be called directly by the RPC method, or after several internal calls. RPC methods were also marked if specific keywords appear in their name. For example, the tool found that an RPC method from d3d10warp.dll leads to ReadProcessMemory:

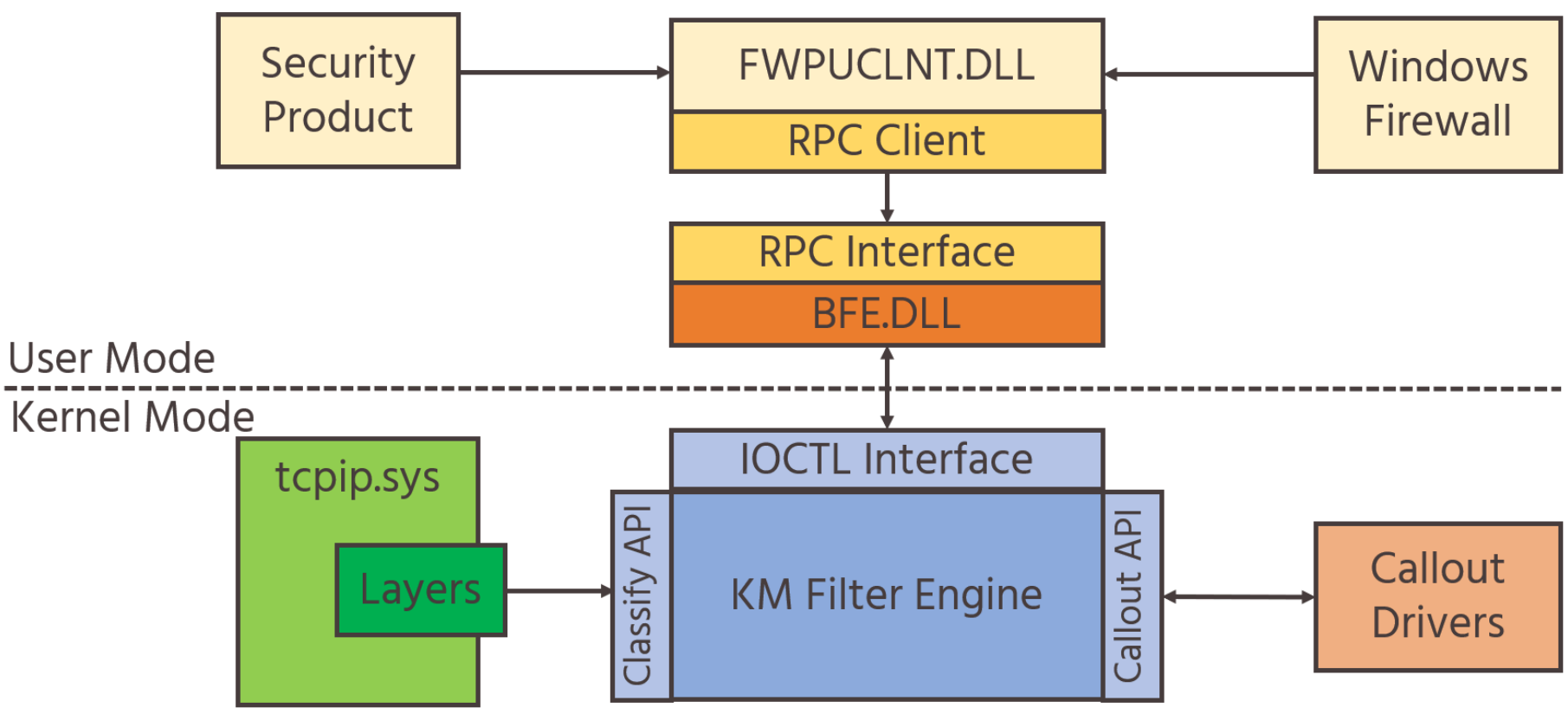


Callout drivers: User-defined drivers that can be loaded and integrated with the platform to extend its' capabilities. These drivers receive network data and can process it in custom ways that the platform doesn't offer: deep inspection of packets according to specific fields of a protocol, packet modification, or performing custom logging.

Filter Engine: A component designed to filter network data by using multiple layers from the OS network stack. The layers are set in user-mode and kernel-mode. The user-mode component filters RPC and IPSec network data. The kernel-mode engine filters the network and transport layers of the TCP/IP stack. It also sends the network data to the callout drivers.

Base Filtering Engine (BFE): A user-mode service that is implemented in BFE.DLL, that also exports management functions for user interaction. It executes under svchost.exe and controls the WFP components. It accepts commands to add or remove filters, offers data and statistics about the platform, and forwards configuration settings to other components in the system.

The following schema is an overview of the platform:



https://learn.microsoft.com/en-us/windows/win32/fwp/windows-filtering-platform-architecture-overview

FWPUCLNT.DLL

FW

Fw
cu
ac

Th


```
    DuplicateHandle(hSourceProcess, hSourceHandle, GetCurrentProcess(), accessToken,
       desiredAccess, 0, 0);
    [snip]
}
```

BFE.DLL

BfeRpcOpenToken calls BfeDriverTokenQuery and if there is no error the process ID of the BFE service is returned to the RPC client along with the handle to the token.

```
int BfeRpcOpenToken(RPC_BINDING_HANDLE bindingHandle, int engineHandle, LUID modifiedId,
    DWORD* BfePid, HANDLE* HandleToDuplicate, HANDLE* HandleToClose)
{
    [snip]
    if (!BfeDriverTokenQuery(modifiedId, &tokenHandle))
    {
        CurrentProcess = GetCurrentProcess();
        *BfePid = GetProcessId(CurrentProcess);
        *HandleToDuplicate = tokenHandle;
    }
    [snip]
}
```

BfeDriverTokenQuery sends a device IO request to the device "\\.\WfpAle." The input buffer is the modifiedId value and the output buffer is the handle to the token.

```
int BfeDriverTokenQuery(LUID ModifiedId, HANDLE* TokenHandle)
{
    LUID modifiedId = ModifiedId;
    return BfeDeviceIoControl(g_WfpAle, 0x124008, 8, &modifiedId, 8, TokenHandle, 0, 0);
```


0x124020	sets tcpip!gMaxInboundSeqRanges global variable	BfeRpcEngineSetOption
0x128000	WfpAleProcessTokenReference	BfeDriverTokenAddRef
0x128004	WfpAleReleaseTokenInformationById	BfeDriverTokenRelease
0x128010	WfpAleProcessExplicitCredentialQuery	BfeRpcAleExplicitCredentialsQuery

Tcpip.sys

The function the BFE service invokes is WfpAleQueryTokenById. It uses an undocumented structure that was named TOKEN_ENTRY for the purposes of this research.

It will try to find an entry based on the LUID it received, which is the modifiedId value sent in the device IO request. If it is found DuplicateToken is called. The desired access is hard coded to be TOKEN_DUPLICATE and the token type will be TokenPrimary.

Invoking APIs in the kernel by sending a device IO request is useful in bypassing user-mode hooks!

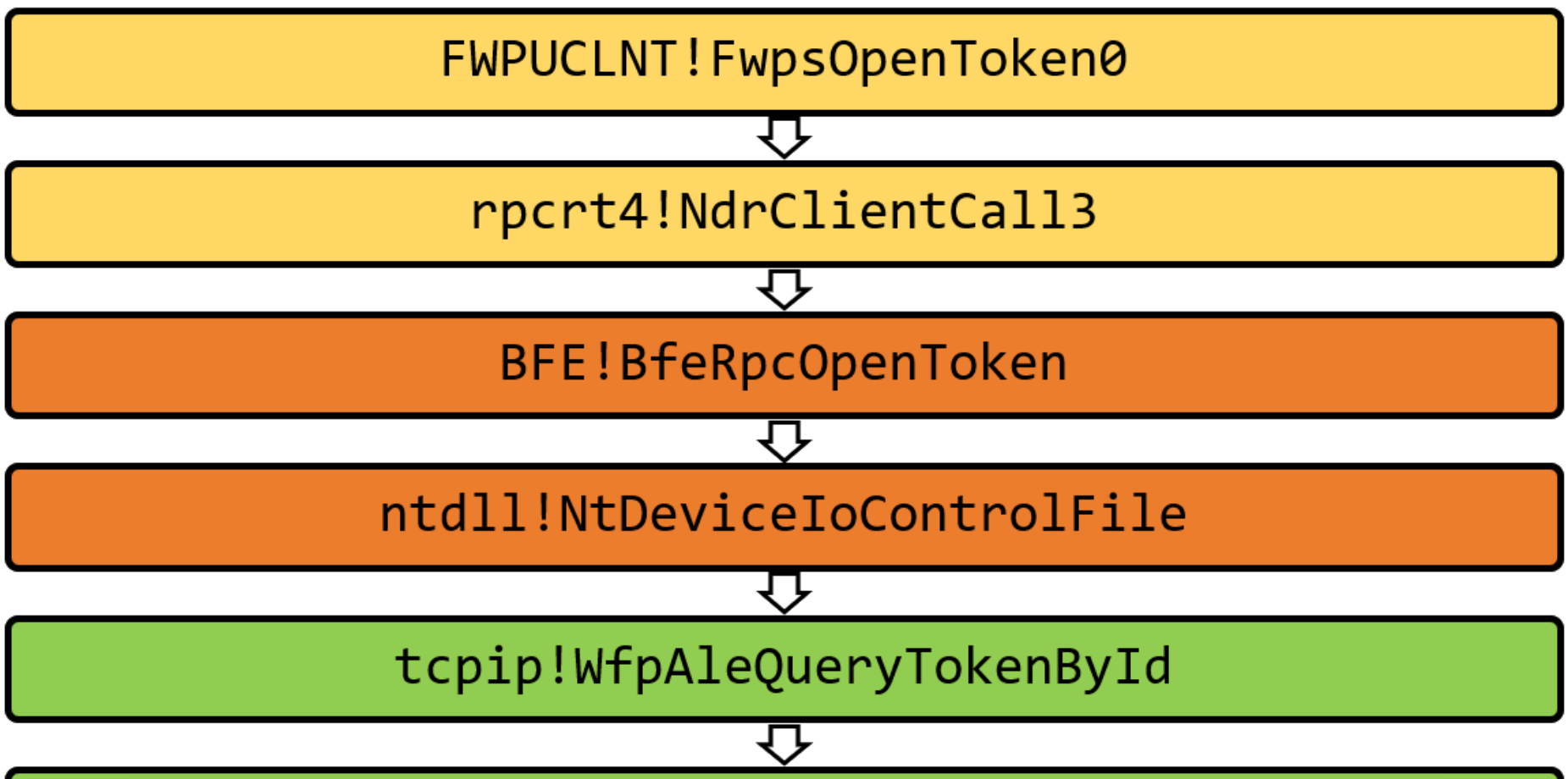
```
int WfpAleQueryTokenById(LUID* pLuid, ACCESS_MASK DesiredAccess, TOKEN_TYPE TokenType,
void** OutputBuffer)
{
    PTOKEN_ENTRY tokenEntry = 0;
    HANDLE NewTokenHandle = 0;
    LUID luid = *pLuid;
    if (!WfpAleAcquireTokenInformation(&luid, &tokenEntry))
    {
        duplicationStatus = ZwDuplicateToken(
            tokenEntry->TokenValue,
            DesiredAccess, // TOKEN_DUPLICATE
            &ObjectAttributes,
            0,
            TokenType,     // TokenPrimary
            &NewTokenHandle);
        if (!duplicationStatus)
            *OutputBuffer = NewTokenHandle;
    }
    return status;
}
```

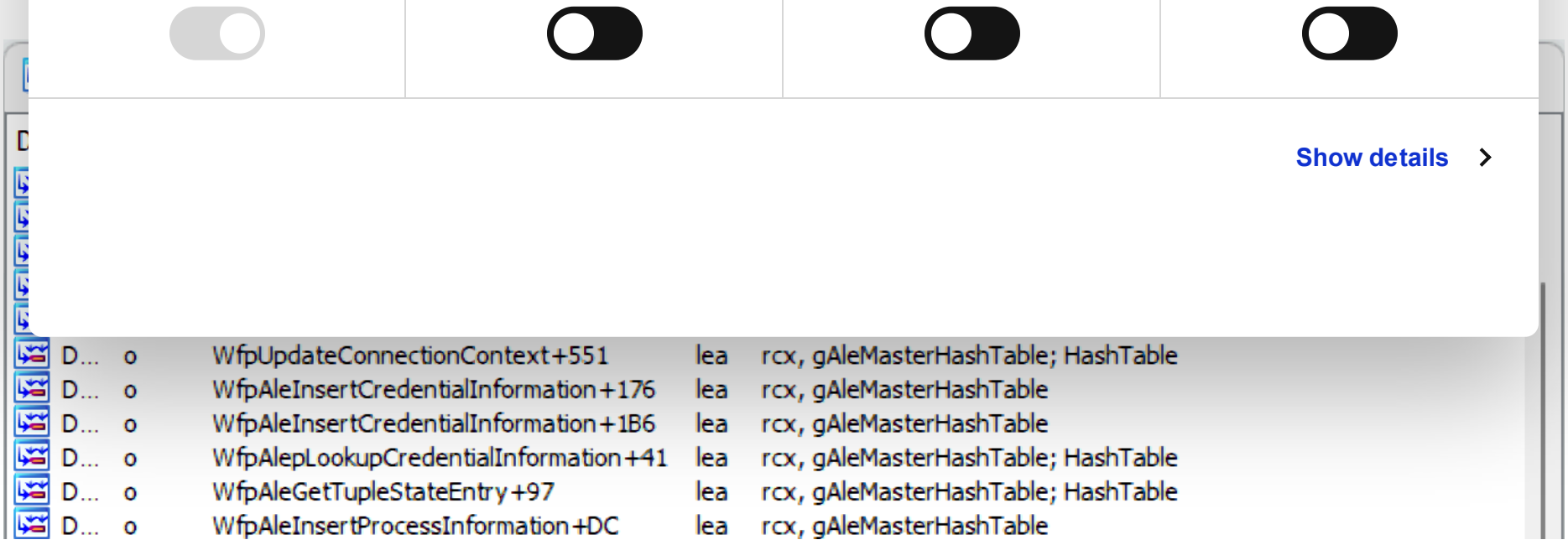
The first step in finding a token entry is calculating a hash that is based on the LUID


```
        i = RtlGetNextEntryHashTable(&gAleMasterHashTable, &Context))
    {
        PTOKEN_ENTRY currentEntry = CONTAINING_RECORD(i, TOKEN_ENTRY, hashTableEntry);
        if (currentEntry->Luid.LowPart == Luid->LowPart &&
            currentEntry->Luid.HighPart == Luid->HighPart)
        {
            *pTokenEntry = currentEntry;
        }
    }
}
```

Token Query Recap

The query starts with the RPC client implemented in FWPUCLNT.DLL, which invokes a method in the BFE service. The service sends a device IO request to the tcpip.sys driver and after several internal functions a hash table is iterated.



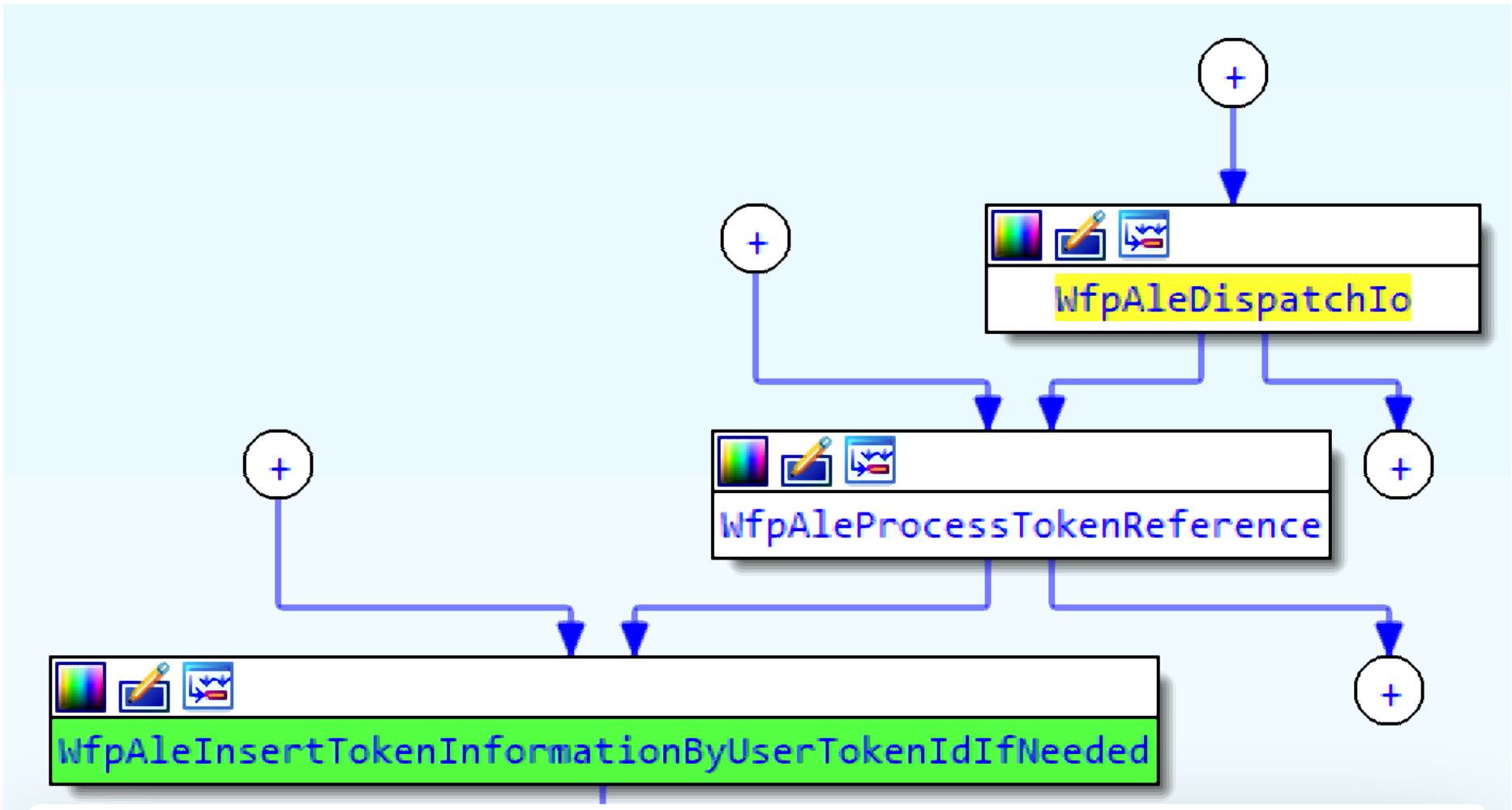


Token entries are added by the function WfpAleInsertTokenInformationByUserTokenIdIfNeeded.

Debugging the boot process of the OS reveals that this function is never called, which means that by default there are no token entries that can be retrieved.

Token Insertion

The insertion function is called by WfpAleProcessTokenReference, which can be invoked by sending a device IO request with the control code 0x128000. The function in the BFE service that sends this specific request is BfeDriverTokenAddRef but it isn't exposed directly by RPC. It is called under certain conditions that aren't simple to create. Triggering this device IO request will insert a token to the table.



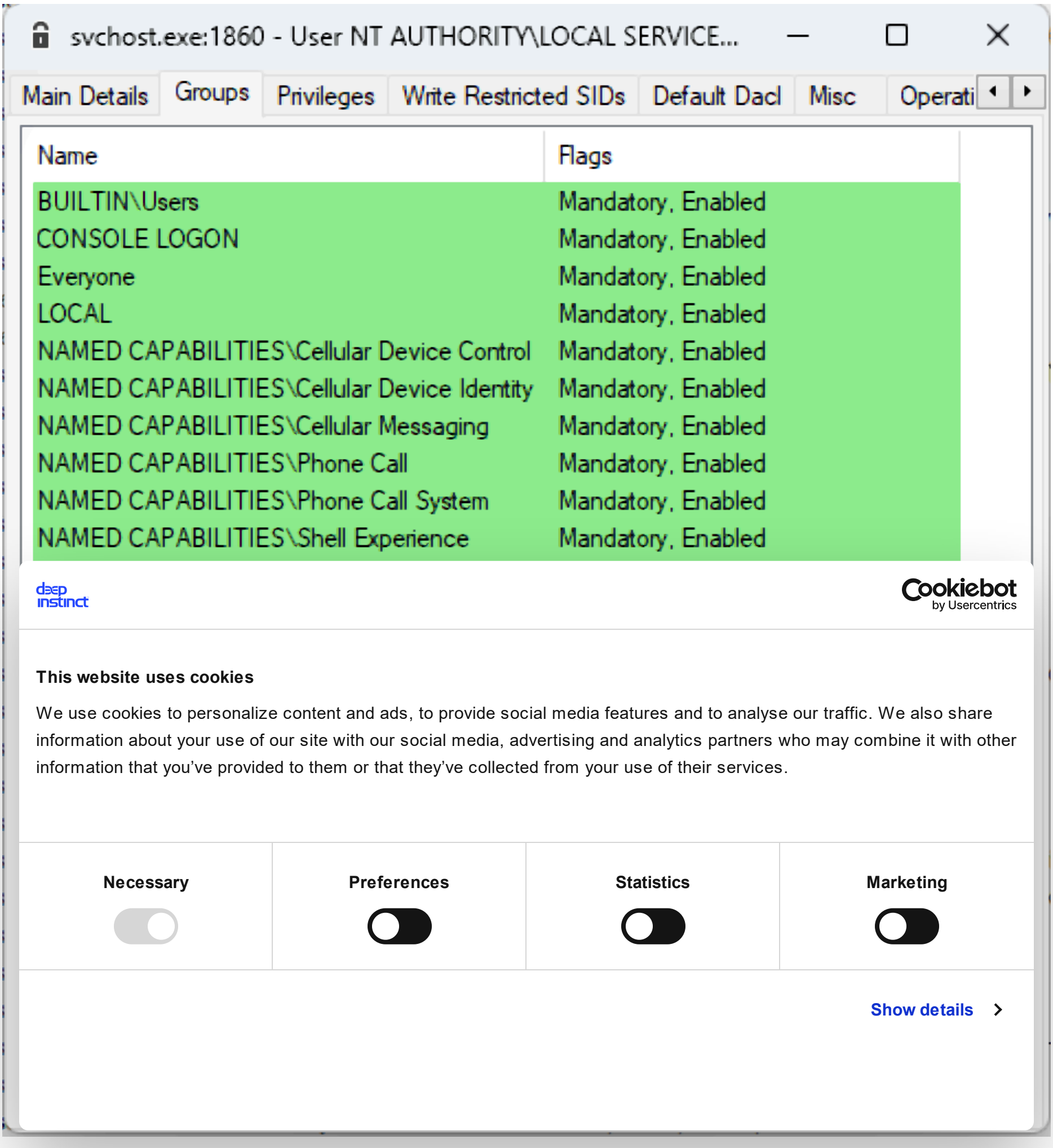

```
        &pEPROCESS, 0);
    KeStackAttachProcess(pEPROCESS, &ApcState);
    ZwDuplicateToken(Token, TOKEN_ALL_ACCESS, &ObjectAttributes, 0, TokenPrimary,
        &newTokenHandle);

    if (!WfpAleCaptureTokenIdByHandle(newTokenHandle, luid))
    {
        if (!WfpAleAcquireTokenInformationFromToken(0, newTokenHandle, luid,
            &newTokenEntry, v13))
        {
            if (!WfpAleInsertTokenInformationByUserTokenIdIfNeeded(newTokenEntry))
                *OutputBuffer = newTokenEntry->Luid;
        }
    }
}
```

Accessing WfpAle

There is no RPC call to the BFE service that will insert a token into the hash table. Sending the device IO request directly to the tcpip driver will solve this problem, but the device WfpAle is created with a security descriptor that blocks any process from gaining a handle to it, except for the BFE service.

```
int WfpAleRegisterDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT* ppDeviceObject)
{
    UNICODE_STRING DeviceName;
    UNICODE_STRING DestinationString;
    RtlInitUnicodeString(&DeviceName, L"\\Device\\WfpAle");
    NTSTATUS error = IoCreateDevice(DriverObject, 0, &DeviceName, FILE_DEVICE_NETWORK,
        FILE_DEVICE_SECURE_OPEN, 0, ppDeviceObject);
    if (!error)
    {
        PDEVICE_OBJECT pDeviceObject = *ppDeviceObject;
        deviceCreated = TRUE;
        if (!WfpAllowBfeGenericAll(pDeviceObject))
        {
            RtlInitUnicodeString(&DestinationString, L"\\DosDevices\\WfpAle");
            if (!IoCreateSymbolicLink(&DestinationString, &DeviceName))
            {
                symbolsLinkCreated = TRUE;
            }
        }
    }
    return error;
}
```

Th
se



The BFE service has an open handle to the device and this handle can be duplicated for another process. That will give any process access to the device WfpAle. This is possible because the security descriptor doesn't block the duplication of the handle, only creating a new one.

Duplicating the handle to the device requires debug privileges and a handle to the BFE service with the permissions PROCESS_DUP_HANDLE and PROCESS_QUERY_INFORMATION. These requirements shouldn't trigger security products since they aren't suspicious. Tools like Process Hacker open these types of handles to show the handle table of processes and the RPC client implemented in FWPUCLNT.DLL also duplicates handles from the BFE service to other processes.

Sending the device IO request directly also helps avoid detection by not performing suspicious calls to DuplicateToken and DuplicateHandle. Security products might be triggered if these APIs return a handle to a token that belongs to "NT AUTHORITY\SYSTEM" to a process that has lower privileges.

When using the RPC client, the handle to the token needs to be duplicated from the BFE service to the current process by calling DuplicateHandle. The only permission this token will have is TOKEN_DUPLICATE, which isn't sufficient to launch a new process, so calling DuplicateToken is necessary to get a token with enough permissions.

By sending the device IO request directly, the token will be sent to the current process instead of the BFE service, and those API calls won't be necessary.

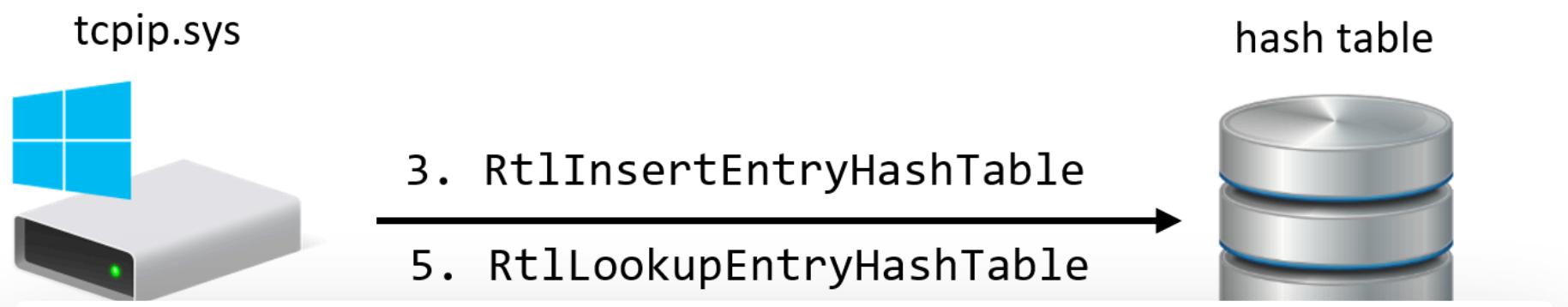
Attack #1 - Duplicating tokens via WFP

The handle table of another process can be retrieved by calling NtQueryInformationProcess. This table lists the tokens held by the process. The handles to those tokens can be duplicated for another process to escalate to SYSTEM.

This technique can be modified to perform the duplication in the kernel instead of calling DuplicateHandle from user mode.

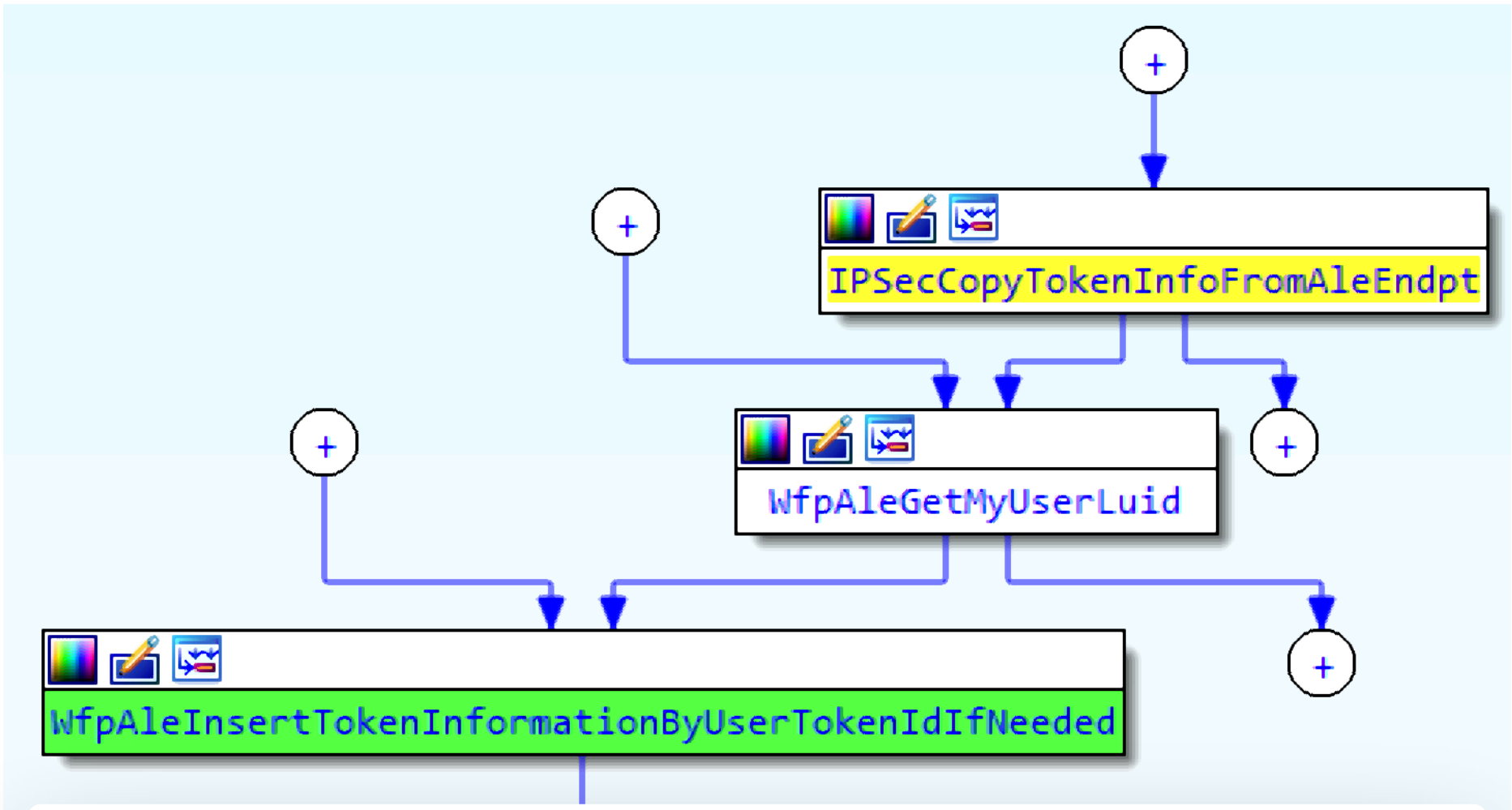
Device IO request is sent to call WfpAleProcessTokenReference. It will attach to the address space of the service, duplicate the token of the service that belongs to SYSTEM, and will store it in the hash table.

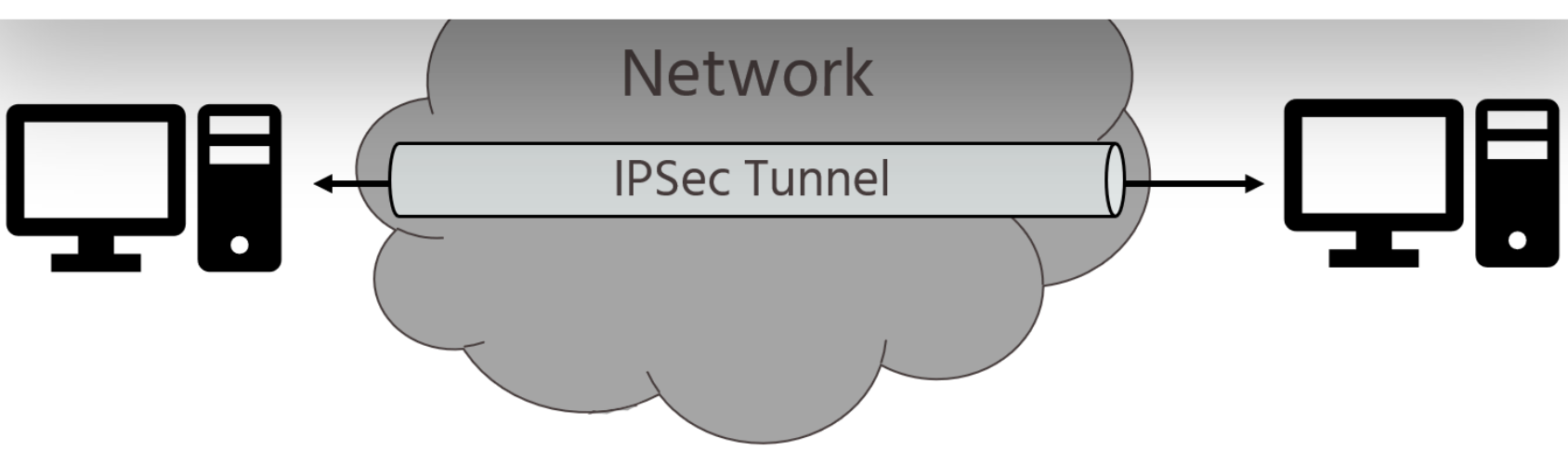
The LUID of the new token will be returned to the caller, and then WfpAleQueryTokenById will be called with the LUID. Handle to a SYSTEM token will be returned to the caller. The access of the handle is hard coded to be TOKEN_DUPLICATE, but it can be duplicated to gain TOKEN_ALL_ACCESS permissions.



Furthermore, the action of duplicating a handle to a SYSTEM token from a service into a process running with lower permissions is suspicious and might be detected, even by underperforming EDR solutions. This technique will evade this detection by avoiding the call to DuplicateHandle.

Additional cross-references to the token insertion function reveal relation to IPSec. Maybe using IPSec will insert a token?





Microsoft provides an example for programmatically configuring an IPSec policy on the machine that uses a pre-shared key for authentication. While the policy is installed, the token of each process that creates a connection that matches the policy is inserted into the hash table stored in tcpip.sys.

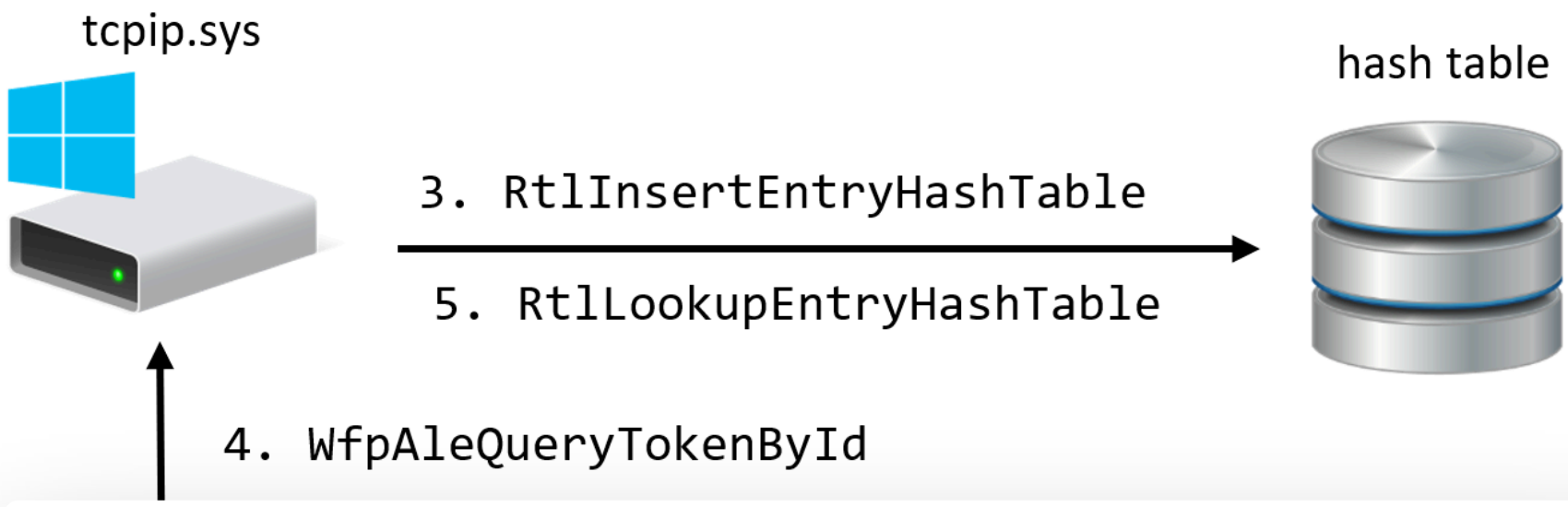
According to the IPSec documentation, the reason this is done is because IPsec impersonates the security context under which the socket is created.

The LUID of the token is unknown to the attacker. This value is used for the modifiedId parameter for FwpsOpenTokeno and later as the index into the hash table. This value ranges between 1 and 0x1000, so it can be brute forced.

Attack #2 – Trigger IPSec Connection

Forcing a service to initiate a connection that matches the policy will result in inserting a SYSTEM token into the table. In this case, the Print Spooler service will be abused to achieve this. It has a documented IDL file, which makes it easier to find RPC methods that will make the service connect to a socket.

One such function is RpcOpenPrinter, which retrieves a handle for a printer by name. When setting the name to "\\127.0.0.1" the service will connect to the localhost. After this RPC call, multiple device IO requests to WfpAleQueryTokenById can be made until it returns a SYSTEM token.



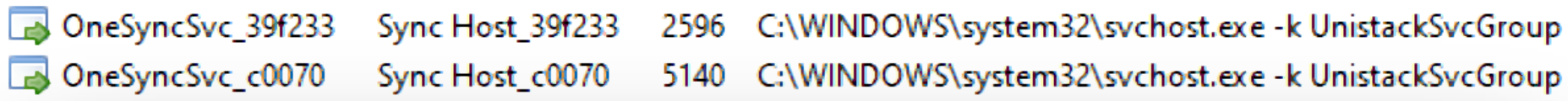
Attack #3 – Manipulate User Service

Gaining the token of another user logged on to the machine can lead to lateral movement in the domain. If the token can be added to the hash table a process can be launched with this user's permissions.

RPC servers running as logged-on users (and not as "NT ATHORITY\SYSTEM") were searched for. The following script looks for processes running as the domain admin then checking if they expose an RPC interface. This led to SyncController.dll.

```
PS C:\WINDOWS\system32> $DomainAdminProcs = Get-Process -IncludeUserName |
where {$_.UserName -eq 'TEST\Administrator'}
$RpcServers = New-Object -TypeName System.Collections.ArrayList
foreach ($proc in $DomainAdminProcs) {
    foreach ($interface in Get-RpcServer -ProcessId $proc.id) {
        [void]$RpcServers.Add($interface.Name) } }
$RpcServers | sort -unique
aphostservice.dll
d3d10warp.dll
modernexecserver.dll
PlaySndSrv.dll
SyncController.dll
...
```

Once multiple sessions are active on the machine, every session will launch the OneSyncSvc service with the user's permissions. This service loads SyncController.dll, which registers the RPC interface 923c9623-db7f-4b34-9e6d-e86580f8ca2a.



5. Call the RPC method. It will cause the service to connect to the local host with port 443. While the connection is active, the token of the user from the other session is stored inside the table.
6. Bruteforce the LUID of the new token while OneSyncSvc is connected to the socket.
7. Launch process with the new token.

OneSyncSvc and SyncController.dll were never abused by an offensive tool, and the RPC call should not trigger security solutions.

Detection

These attacks were developed to be as stealthy as possible, but they can still be detected by looking for the following events on the machine:

- Configuring new IPSec policies that don't match the known network configuration.
- RPC calls to Spooler / OneSyncSvc while an IPSec policy is active.
- Brute force the LUID of a token via multiple calls to WfpAleQueryTokenById.
- Device IO request to the device WfpAle by processes other than the BFE service.

The Windows Filtering Platform generates logs for events. Most logs are about packets drops or failures during the key exchange process. It is possible to generate logs about packets that were allowed to be sent. This must be set explicitly — and it is not recommended — as it will generate a lot of noise. Even when generating those logs it will be difficult to detect the attacks. The following log is about a connection made during the attack:

```
FilterId        : 67348
LayerId         : 48
ReauthReason    : 0
OriginalProfile : None
CurrentProfile  : None
MsFwpDirection  : 0
IsLoopback      : False
Type            : ClassifyAllow
Flags           : IpProtocolSet, LocalAddrSet, RemoteAddrSet,
LocalPortSet, RemotePortSet, AppIdSet, UserIdSet,
                  IpVersionSet, PackageIdSet
Timestamp       : 3/23/2023 10:39:59 AM
IPProtocol      : Tcp
LocalEndpoint   : 127.0.0.1:49841
RemoteEndpoint  : 127.0.0.1:135
```



```
FieldKeyName          MatchType   Value
------------          ---------   -----
FWPM_CONDITION_FLAGS FlagsAllSet IsLoopback
```

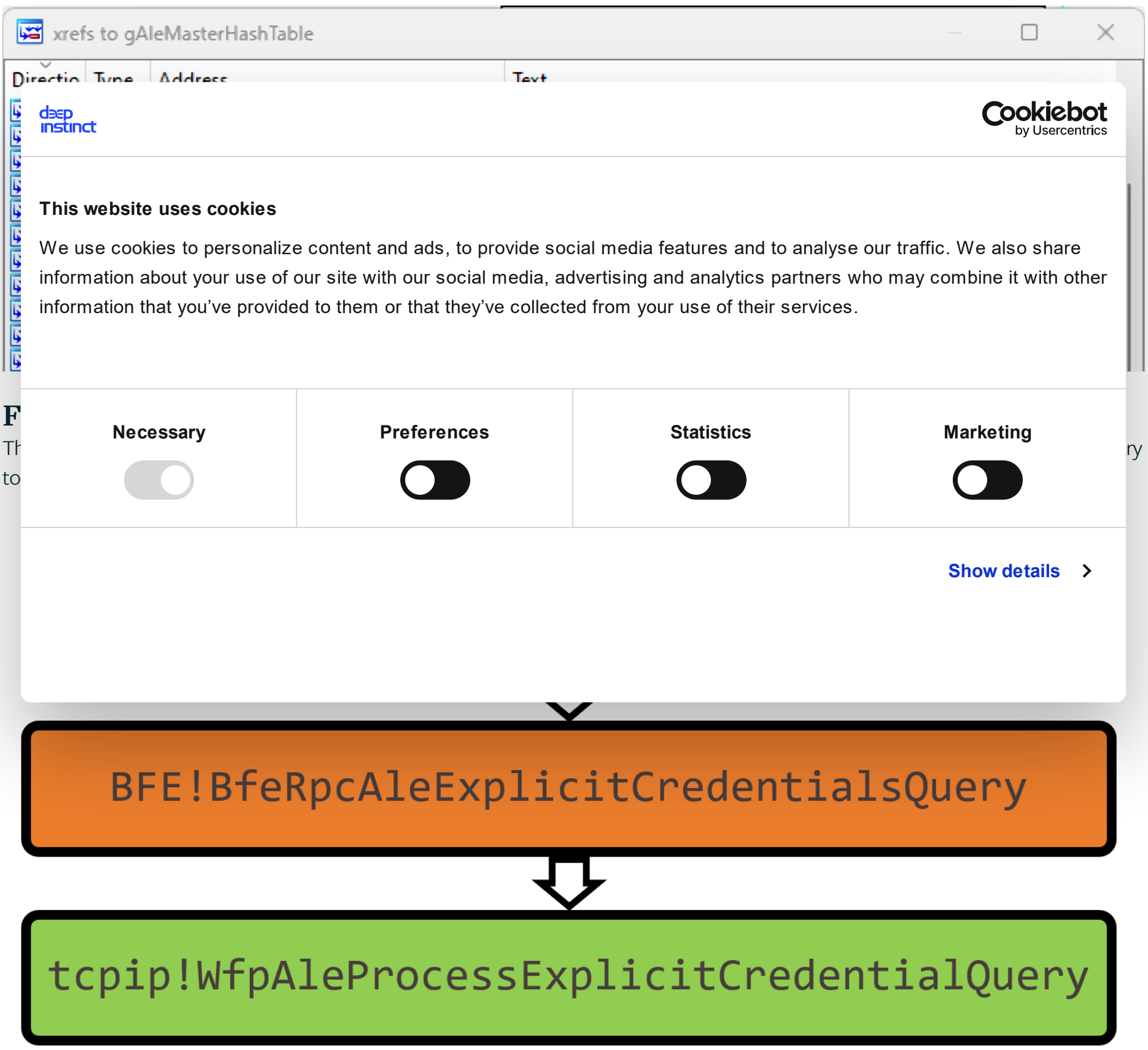
Further Research – tcpip.sys

The driver tcpip.sys creates several devices that expose several functionalities. Sending device IO requests to them can uncover new attack surfaces. Some of the functions are listed in the following table:

Device	Control Code	Tcpip Function
IPSECDOSP	0x124004	IdpProcessQueryStatsIoctl
	0x124002	IdpProcessEnumStateIoctl
NXTIPSEC	0x128028	IPsecSetS2STunnelInterfaceHndlr

	0x12801C	IPSecNotifyStatusHndlr
	0x128018	IPSecUpdateSaInfoHndlr
WFP	0x12803C	IoctlKfdResetState
	0x124050	IoctlKfdSetBfeEngineSd
	0x128004	IoctlKfdAddIndex

Cross-references to the hash table in the tcpip driver reveal a lot about the various operations it is used for and the data it manages in addition to tokens. Some of the data can be valuable for attackers, for example: data labeled as “Process Explicit Credentials.”



The BFE service performs an access check on the client when BfeRpcAleExplicitCredentialsQuery is called. Processes running with admin privileges receive an ERROR_ACCESS_DENIED. If the same call is sent from a process running as SYSTEM, the BFE service allows it and sends a device IO request to the tcpip driver.

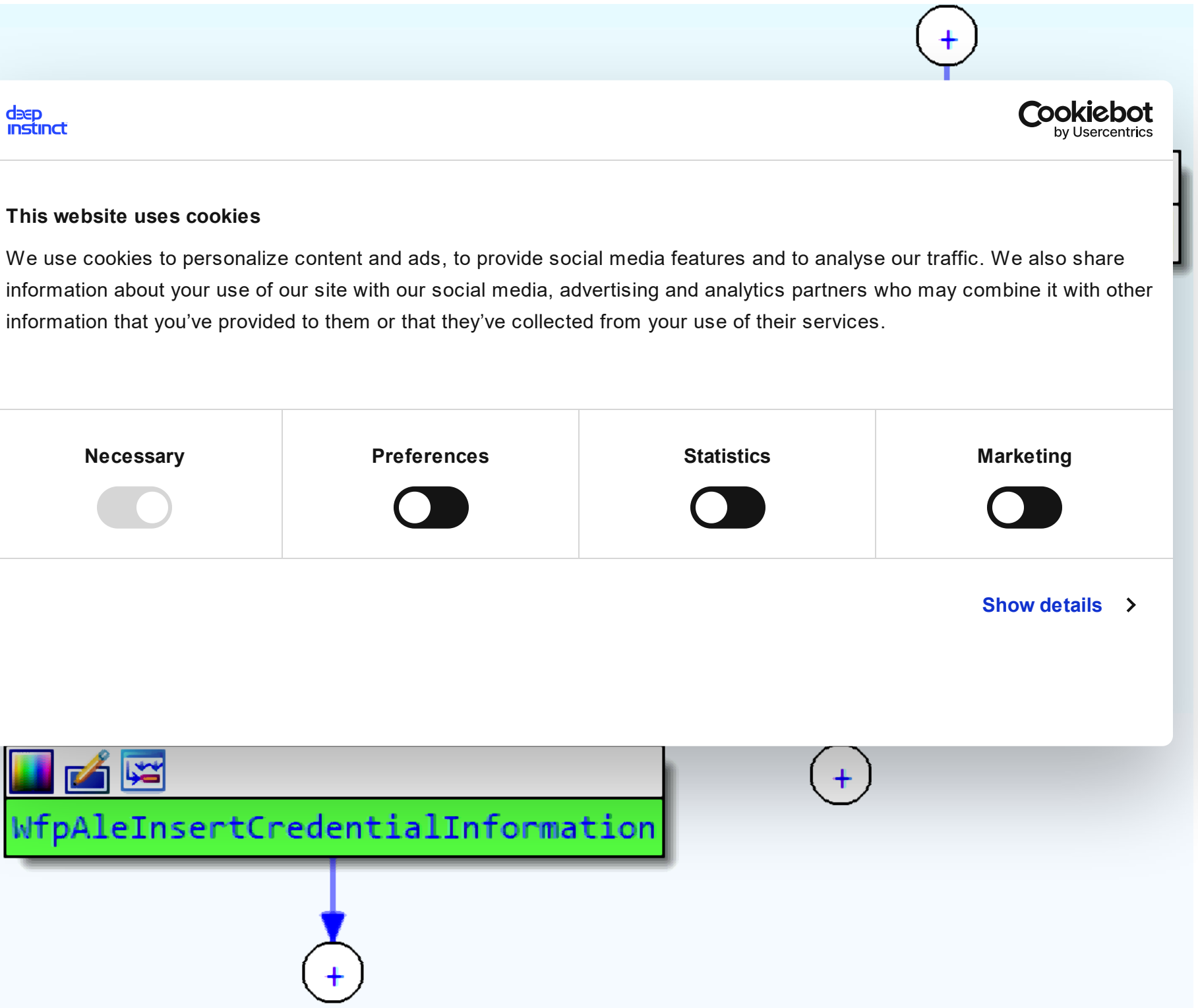
This is another security mechanism that can be bypassed by duplicating the handle to the device WfpAle. Sending the device IO request directly will skip the access check in the BFE service.

```
int BfeDriverQueryExplicitCredentials(int ValueToQuery, void* ExplicitCredentials)
{
    int value = ValueToQuery;
    return BfeDeviceIoControl(WfpAleHandle, 0x128010, 8i64, &value, 0x107FE,
        ExplicitCredentials, 0, 0);
}
```

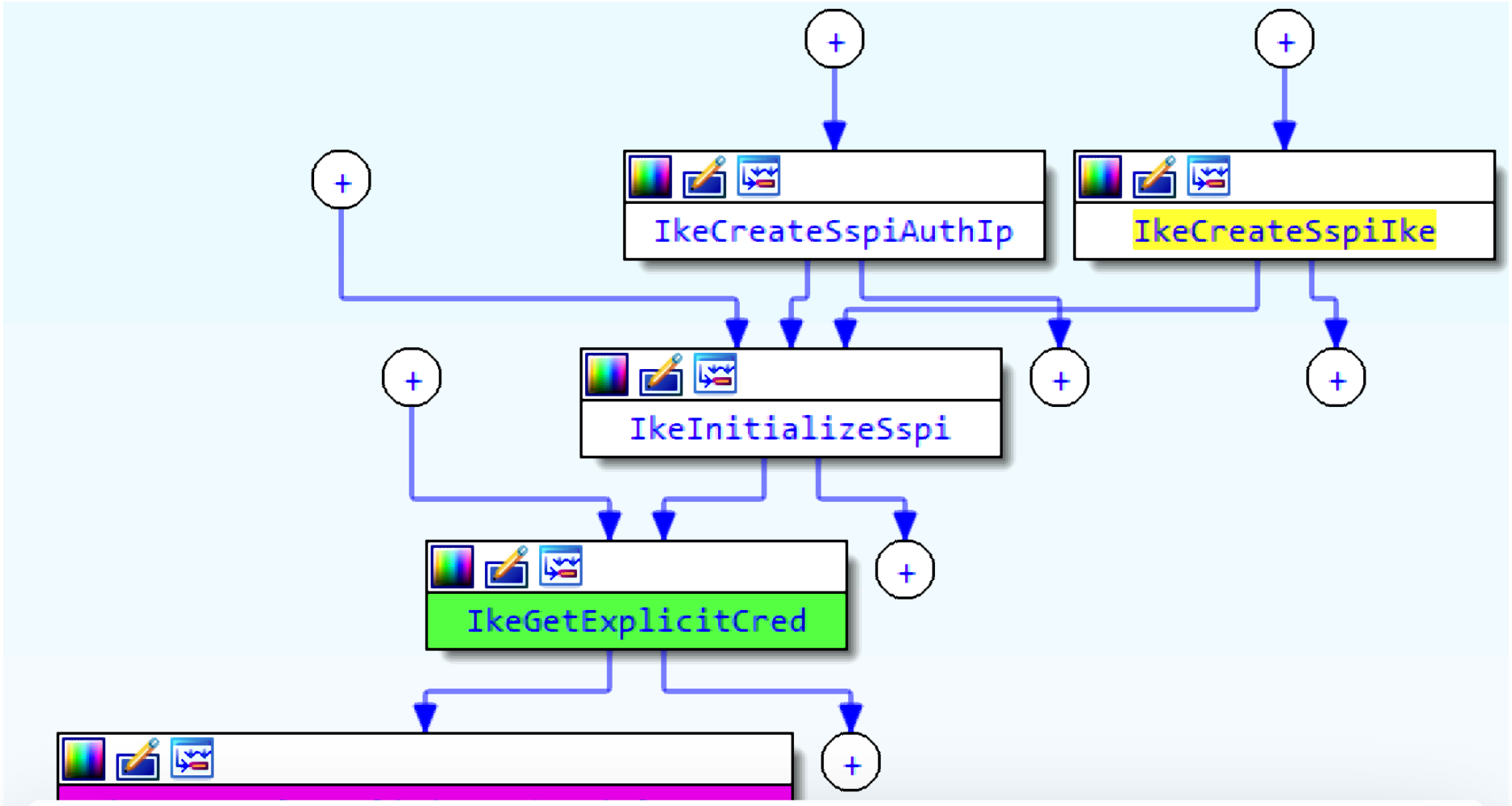
The function that inserts explicit credentials is WfpAleInsertCredentialInformation, but much like tokens, they aren't inserted into the hash table in tcpip.sys without a special configuration. This configuration has not been found yet.

The purpose of the data labeled as "process explicit credentials" is unclear at this point, but cross references to the functions related to it might shed some light.

The function that inserts credentials to the table is called by functions named WfpAleSetSecurity and WfpAleProcessSocketOption. Based on those names, maybe WSASetSocketSecurity is somehow related?



FwpsAleExplicitCredentialsQueryo is called by the IKE service in the function IkeGetExplicitCred. Based on cross-references to this function, credentials might be inserted when using a Security Support Provider Interface (SSPI).



This research was reported to Microsoft Security Response Center. According to Microsoft this behavior is by design.

GitHub repo: https://github.com/deepinstinct/NoFilter

← BACK TO BLOG