






# Bypassing Access Mask Auditing Strategies

 Jonathan Johnson · [Follow](#)  
6 min read · Apr 5, 2022

 --    

## Introduction

This past week I briefly talked about Process Access data within a [talk](#) that [Olaf](#) and I gave at ATT&CKCON 3.0 (YouTube link isn't live yet). During this presentation I talked about the significance of this sub-data source and what it meant to defenders. My portion was heavily focused on how attackers can leverage an attribute that exists in a lot of Process Access based logs known as `Granted Access`. I won't touch on this too much but to summarize, defenders can understand the minimum access rights an attacker needs to perform an operation on an object by tracking down the Win32 API being leveraged ([MiniDumpWriteDump](#), [CreateRemoteThread](#), and more). This allows the defender to leverage bitwise operations within their analytics (depending on if it is supported by your platform) to help track these behaviors without having to look for specific access masks. Due to time I was unable to talk about the potential strategies an attacker could take in hopes to fly under the radar of any access right detections. I would like to touch on that now and some of the research that went along with it.

## Research

### DuplicateHandle

Previously while doing access token research I ran across a Win32 API — [DuplicateTokenEx](#) that could be leveraged to duplicate a handle to a token and specify the updated rights wanted on the duplicated handle. This allowed for higher access to a token then previously requested/granted.

Let me walk through on example of this using [NtObjectManager](#):

```
PS > $lsass = Get-NtToken -ProcessId (Get-Process lsass).Id -
Access Duplicate, Query #Obtaining a handle to LSASS's token with
TOKEN_DUPLCIATE, TOKEN_QUERY rights
PS > $dup = $lsass.DuplicateToken("Primary", "Impersonation",
"Impersonate") #Duplicating the handle so that I have
TOKEN_IMPERSONATE rights to LSASS now.
```

```
PS > $dup.GrantedAccess #Proof
Impersonate
```

Previously I’ve heard of another Win32 API — DuplicateHandle that sounded like it did the same thing and wasn’t object specific. This API seemed to be very flexible. I had a theory that an attacker could use this API to essentially duplicate an already existing handle to a process (only needing process access rights — PROCESS\_DUP\_HANDLE /0x40), specify more access to that duplicated handle and in turn to that target process. This would be nice if an attacker wanted to fly under the radar and make it look like they only got rights `0x40` to LSASS, but in reality they duplicated that handle and specified the correct rights within that duplicated handle allowing them to do some bad afterwards, say dump LSASS.

I ran this idea by James Forshaw and he was generous enough to send me a great blog — Bypassing SACL Auditing on LSASS that leverages this idea, but to bypass SACL auditing....best part it was from 2017.

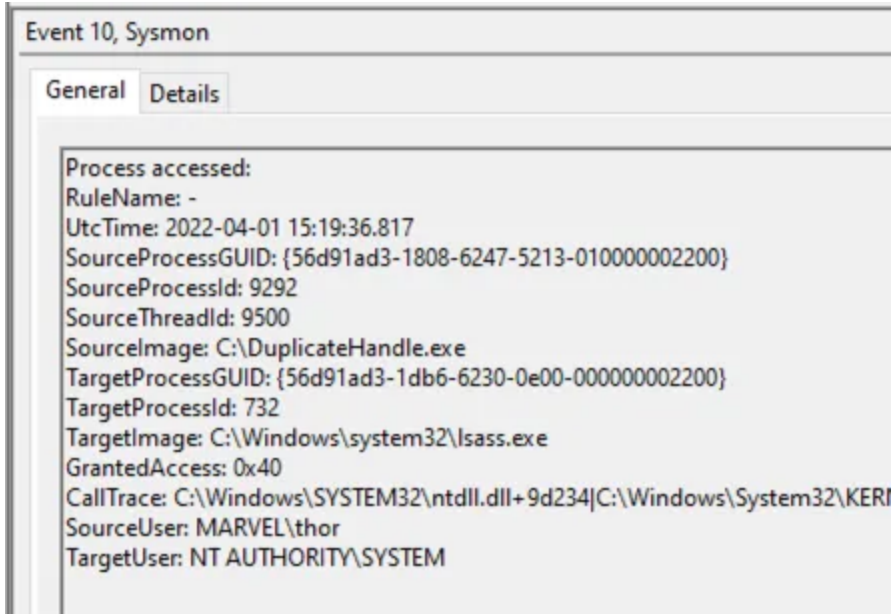
This expanded my understanding on this strategy as it could now be used to:

- Bypass SACL Auditing
- Bypass detections using static access masks and bitmask operations

**Note:** SACL auditing and bypassing detection that are using static access masks or bitmask operations sound similar, because they are functionally the same. However; I see bypassing SACL auditing as being more of a collection (configurational) bypass and the latter being a detection (logical) bypass.

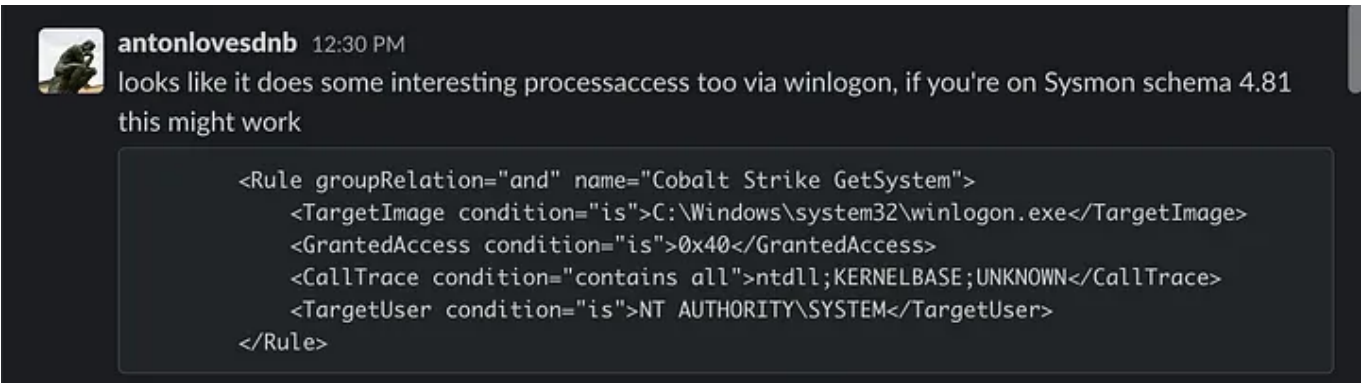
I decided to take it a step further and write a POC to test my assumptions. I won’t release this, as there is plenty of code out there and NtObjectManager makes it easy to test this as well. However; here is a screenshot of my code and of a Sysmon EID 10 to show the access requested.

```
C:\Users\thor\Desktop>DuplicateHandle.exe 732
[*] Handle: 000000000000009C obtained with rights PROCESS_DUP_HANDLE.
[*] Handle duplicated: 000000000000004C. Handle now has with rights PROCESS_ALL_ACCESS.
```



As seen above I am obtaining an initial handle to LSASS with **PROCESS\_DUP\_HANDLE** rights, but then duplicating the handle to get **PROCESS\_ALL\_ACCESS**.

Around the time I finished writing a POC on this [Anton](#) posted in the #detectionlife channel within the BloodHound slack asking why when he ran `getsystem` that beacon was opening a handle to `lsass` and `winlogon` with granted access rights `0x40`.



Now there are a lot of handle requests from beacon (which Anton also mentions), but I will dive into that later. My theory was that beacon was leveraging James’ duplicate handle strategy to fly under the radar for SACL auditing. Let’s test this.

## Beacon

The first thing I did before testing beacon was to track down any other functions that DuplicateHandle calls under the hood. I did this because a lot of code now a days seem to call either syscalls directly or some undocumented function (ntdll.dll) to avoid detection.

I know now that if I were to see any level of these calls within beacon, they would relate to each other.

Note: I am looking at this as if I am already holding High IL and running the default `getsystem` within Cobalt Strike.

My process was as follows:

- Launch beacon with WinDbg attached in user-mode.
- Broke on **kernelbase!OpenProcess** where the process id was equal to my winlogon process (have to do this because beacon requests handles to System, csrss, smss, wininit, etc).
- Broke on **advapi32!DuplicateHandle**
- Broke on **advapi32!OpenProcessToken**
- Broke on **advapi32!ImpersonateLoggedOnUser**

Here is an image of the process flow:

As seen above, we can confirm that Cobalt Strike is leveraging this bypass technique.

This was a pretty targeted approach. I approached it this way because I am comfortable with the function calls used for impersonation. There are other ways one could go about doing something similar if they were unaware of the API calls used, like using Time Travel Debugging or API Monitor.

I wanted to show this process to show that there are tools in the wild that are leveraging this bypass technique. Cobalt Strike isn't the only one, but being that it seems to be so popular I wanted to use it as a use-case.

## Detection Tips

1. There are A LOT more requests from beacon than there should be in my opinion. I would test to see how often the rights **PROCESS\_DUP\_HANDLE** (0x40) is getting granted to processes within your organization. I tested how often processes were accessing LSASS/Winlogon with 0x40 rights within my test organization and only got back my test activity (outside of svchost.exe — this seems to be quite loud). Be sure to test before implementing this strategy.
2. If the above seems to be too noisy consider doing a basic detection for a process requesting 0x40 rights 5 times within a min should be fishy.

## Conclusion

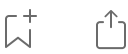
After explaining the importance of Process Access data and how defenders could use it within my talk with Olaf, I wanted to explain how attackers might attempt to avoid this type of detection. It happened that while going through this process I learned a lot along the way:

- About James' blog
- Cobalt Strike is leveraging this strategy

It is good to note, not only Cobalt Strike is leveraging this strategy but others might be as well. Attackers have to access objects when performing their techniques, my hope is to bring awareness of how defenders can look into protecting and auditing these objects better.

## References

- Bypassing SACL Auditing on LSASS by James Forshaw
- Conversations with James Forshaw
- Thank you to Anton for bringing this conversation up within the Bloodhound slack.



Written by Jonathan Johnson

Follow



870 Followers

Principal Security Engineer @Prelude | Windows Internals