Tyranid's Lair

Friday, 26 May 2017

Reading Your Way Around UAC (Part 3)

This is the final part in my series on UAC (Part 1 and Part 2 links). In Part 2 we found that if there's any elevated processes running in a split-token admin session on Windows earlier than 10 we could read the primary token from that process as a normal user and impersonate it giving us 99% of admin privileges without a prompt. Of course there's the proviso that there's an elevated process running on the same desktop, but I'd say that's pretty likely for almost any user. At least in most cases you can use silent elevation (through auto elevation, or a scheduled task for instance) to get a process running elevated, you don't care what the process is doing just that it exists.

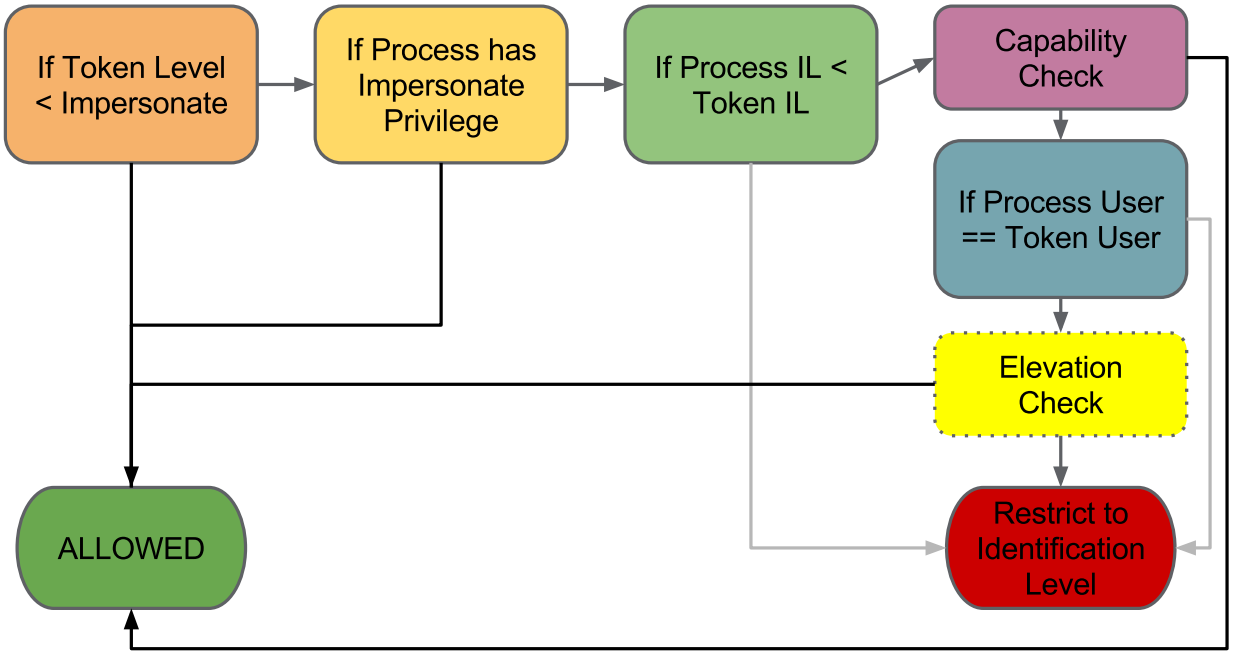
Also it's been pointed out that what I described in Part 2 sounds exactly like the UAC Bypass used in the Stinger Module "released" in the Vault 7 dumps. It could very well be, I'd be surprised if someone didn't already know about this trick. I've not seen the actual module, and I'm not that interested to find out, but if anyone else is then go for it.

Broken Windows

Anyway on to Windows 10. It seems that the impersonation checks fails and we're dumped down to an *Identification* token which is pretty much useless. It seems likely that Microsoft have done something to mitigate this attack, presumably they know about this exploitation route. This needs to be reiterated, just because Microsoft fixes a UAC bypass doesn't mean that they're now treating it as a security boundary.

The likely candidate for additional checks is in the *SeTokenCanImpersonate* function in the kernel, and if we look we find it's been changed up a bit. Compare the following diagram to the similar one in Part 2 and you'll

notice a couple of differences:



I've highlighted the important additional step, the kernel now does an elevation check on the impersonation token to determine if the caller is allowed to impersonate it. A simplified version is as follows:

```
TOKEN* process_token = ...;
TOKEN* imp_token = ...;
#define LIMITED_LOGON_SESSION 0x4

if (SeTokenIsElevated(imp_token)) {
  if (!SeTokenIsElevated(process_token) &&
      (process_token->LogonSession->Flags & LIMITED_LOGON_SESSION)) {
    return STATUS_PRIVILEGE_NOT_HELD;
  }
}
```

The additional check first determines if the impersonation token is elevated (we'll go into what this means in a bit). If it's not elevated then the function carries on with its other checks as prior to Windows 10. However if it

is elevated (which is the case from our PoC in Part 2) it then does an elevation check on the process token. If the process token is not elevated the function will check if a specific flag is set for the Token's Logon Session. If the flag is set then an error is returned. What this means is that there's now only two scenarios where we can impersonate an elevated token, if the process doing the impersonation is already elevated or the process token's logon session has this flag set. Our PoC from Part 2 will clearly fail the first scenario and presumably fails the second. We can check this using a kernel debugger by running an elevated and a non-elevated copy of *cmd.exe* and checking the flags.

```
PROCESS ffffb50dd0d0a7c0
    Image: cmd.exe
    Token                             ffff980d0ab5c060
    * SNIP *

kd> dx -r1 ((nt!_TOKEN*)0xffff980d0ab5c060)->LogonSession->Flags
0xc [Type: unsigned long]
```

This process token is non-elevated, the flags are set to 0xC which contains the value 4 which is the *LIMITED_LOGON_SESSION* flag.

```
PROCESS ffffb50dd0cc1080
    Image: cmd.exe
    Token                             ffff980d0a2478e0
    * SNIP *

kd> dx -r1 ((nt!_TOKEN*)0xffff980d0a2478e0)->LogonSession->Flags
0xa [Type: unsigned long]
```

And now the elevated process token, flags are 0xA which doesn't contain the *LIMITED_LOGON_SESSION* flag. So we are getting caught by this second check. Why is this check even there at all? As far as I can tell it's for compatibility, possibly with Chrome *ahem*. The additional check was added prior to final release of Windows 10 10586 (in the insider previews this additional logon session flag check didn't exist, and in 10240 the whole elevation check was present but wasn't on by default). So assuming for the moment we can't get a process token without that flag set, what about the *SeTokenIsElevated* function, is that exploitable in anyway? The code of *SeTokenIsElevated* looks something like the following:

```
BOOLEAN SeTokenIsElevated(_TOKEN *token) {
  DWORD* elevated;
```

```
  SeQueryInformationToken(token, TokenElevation, &elevated);
  return *elevated;
}
```

The function queries a token information property, *TokenElevation*, which returns a non-zero value if the token is elevated. The *SeQueryInformationToken* API is the kernel equivalent to *NtQueryInformationToken* from user mode (mostly anyway), so we should also be able to query the elevation state using PS. Let's change a script we had in Part 2 to print the elevation state instead of integrity level as we proved last time that *IL* doesn't mean a token is really privileged.

```
function Write-ProcessTokenInfo {
  Param([NtApiDotNet.NtProcess]$Process)
  Use-NtObject($token = Get-NtToken -Primary -Process $Process) {
    $token | Format-List -Property User, TokenType, Elevated
  }
}

Use-NtObject($ps = Get-NtProcess -Name mmc.exe) {
  Write-ProcessTokenInfo $ps[0]
}

Write-ProcessTokenInfo $(Get-NtProcess -Current)
```

This outputs:

```
User      : domain\user
TokenType : Primary
Elevated  : True

User      : domain\user
TokenType : Primary
Elevated  : False
```

So what does the kernel use to make the determination, clearly it's not the *IL* as we've already changed that and it still failed. If you dig into the implementation of *SeQueryInformationToken* the kernel checks two things, firstly whether the token has any *GOD* privileges (it just so happens the list matches the ones we couldn't enable in Part 2) and whether the Token's groups have any "elevated" SIDs.

The list of *GOD* privileges that I know of are as follows:

- SeCreateTokenPrivilege
- SeTcbPrivilege
- SeTakeOwnershipPrivilege
- SeLoadDriverPrivilege
- SeBackupPrivilege
- SeRestorePrivilege
- SeDebugPrivilege
- SeImpersonatePrivilege
- SeRelabelPrivilege
- SeDelegateSessionUserImpersonatePrivilege

As an aside, isn't it odd that SeAssignPrimaryTokenPrivilege isn't in that list? Not that it matters, Administrators don't get that privilege by default, so perhaps that's why.

The "elevated" SIDs don't seem to have an explicit (full) blacklist, instead the kernel calls the function *RtlIsElevatedRid* with each group and uses that to determine if the SID is an elevated SID. The only check is on the last relative identifier in the SID not the whole SID and looks something like this:

```
BOOLEAN RtlIsElevatedRid(SID_AND_ATTRIBUTES *sid_and_attr) {
  if ( sid_and_attr->Attributes &
       (SE_GROUP_USE_FOR_DENY_ONLY | SE_GROUP_INTEGRITY)) {
    return FALSE;
  }
  PSID sid = sid_and_attr->Sid;
  BYTE auth_count = *RtlSubAuthorityCountSid(sid);
  DWORD last_rid = *RtlSubAuthoritySid(sid, auth_count-1);
  DWORD check_rids[] = { 0x200, 0x204, 0x209, 0x1F2,
                         0x205, 0x206, 0x207, 0x208,
                         0x220, 0x223, 0x224, 0x225,
                         0x226, 0x227, 0x229, 0x22A,
                         0x22C, 0x239, 0x72 };

  for(int i = 0; i < countof(check_rids); ++i) {
    if (check_rids[i] == last_rid) {
```

```
      return TRUE;
    }
  }
  return FALSE;
}
```

There's currently 19 banned RIDs. To pick an example, 0x220 is 544 in decimal. The string SID for the *BUILTIN\Administrators* group is S-1-5-32-544 so that's clearly one banned SID. Anyway as we've got *Duplicate* access we can make a non-elevated Token using CreateRestrictedToken to set some groups to Deny Only and remove *GOD* privileges. That way we should be able to impersonate a token with some funky privileges such as *SeMountVolumePrivilege* which are still allowed, but that's not very exciting. The thought occurs, can we somehow create a process we control which doesn't have the logon session flag and therefore bypass the impersonation check?

Getting Full Admin Privileges

So we're now committed, we want to get back that which Microsoft have taken away. The first thought would be can we just use the elevated token to create a new process? As I described in Part 2 due the various checks (and the fact we don't have *SeAssignPrimaryTokenPrivilege*), we can't do so directly. But what about indirectly? There's a number of system services where the following pattern can be observed:

```
void CreateProcessWithCallerToken(string path) {
  RpcImpersonateClient(nullptr);
  HANDLE Token = OpenThreadToken();
  HANDLE PrimaryToken = DuplicateToken(Token, TokenPrimary);

  CreateProcessAsUser(PrimaryToken, path, ...);
}
```

This code creates a new process based on the caller's token, be that over RPC, COM or Named Pipes. This in itself isn't necessarily a security risk, the new process would only have the permission that the caller already had during impersonation. Except that there's numerous places in the kernel, including our impersonation check, that explicitly check the process token and not the current impersonation token. Therefore being able to impersonate a token doesn't necessarily mean that the resulting process isn't slightly more privileged in some ways. In this case that's exactly what we get, if we can convince a system service to create a process using the non-elevated copy of an elevated token the logon session flags won't have the

LIMITED_LOGON_SESSION flag set as the logon session is shared between all Token object instances. We can therefore do the following to get back full admin privileges:

1. Capture the admin token and create a restricted version which is no longer elevated.
2. Impersonate the token and get a system service to create a new process using that token. This results in a low-privilege new process which happens to be in the non-limited logon session.
3. Capture the original full privileged admin token in the new process and impersonate, as our logon session doesn't have the *LIMITED_LOGON_SESSION* flag the impersonation check passes and we've got full privilege again.

A good example of this process creation pattern is the *WMI* Win32_Process::Create call. It's a pretty simple function and doesn't do a lot of checking. It will just create the new process based on the caller. It sounds ideal and PS has good support for WMI. Sadly COM's weird security and cloaking rules makes this a pain to do in .NET, let alone PS. I do have C++ version but it's not simple or pretty, but it works from Vista through Windows 10 Creators Update. I've not checked the latest insider preview builds (currently running RS3) to see if this is fixed yet, perhaps if it isn't yet it will be soon. I might release the C++ version if there's enough interest.

Still it would be nice if I could give a simple script for use in PS for the hell of it. One interesting observation I made when playing with this is that impersonating the restricted version of the elevated token while calling the LogonUser API with the *LOGON32_LOGON_NEW_CREDENTIALS* logon type returns you back the elevated token again (even with a *High IL*), run the following script to see the result (*$token* needs to be a reference to the elevated token).

```
# Filter elevated token down to a non-elevated token
$lua_token = Get-NtFilteredToken -Token $token -Flags LuaToken
$lua_token | Format-List -Property User, Elevated, IntegrityLevel

# Impersonate non-elevated token and change credentials
Use-NtObject($lua_token.Impersonate()) {
    Get-NtToken -Logon -User ABC -LogonType NewCredentials
} | Format-List -Property User, Elevated, IntegrityLevel
```

This is interesting behavior, but it still doesn't seem immediately useful. Normally the result of *LogonUser* can be used to create a new process. However as the elevated token is still in a separate logon session it won't work. There is however one place I know of that you can abuse this "feature", the Secondary Logon service and specifically the exposed CreateProcessWithLogon API. This API allows you to create a new process by first calling *LogonUser* (well really *LsaLogonUser* but anyway) and takes

a *LOGON_NETCREDENTIALS_ONLY* flag which means we don't need permissions or need to know the password.

As the Secondary Logon service is privileged it can happily create a new process with the newly minted elevated token, so all we need to do is call CreateProcessWithLogon while impersonating the non-elevated token and we get an arbitrary process running as full administrator (barring some privileges we had to remove) and even a *High IL*. The only problem is we've changed our password in the session to something invalid, but it doesn't matter for local access. As it's still pretty long I've uploaded the full script here, but the core is these few lines:

```
Use-NtObject($lua_token.Impersonate()) {
 [SandboxAnalysisUtils.Win32Process]::CreateProcessWithLogin(
     "Badger", "Badger", "Badger",
     "NetCredentialsOnly", "cmd.exe", "cmd.exe",
     0, "WinSta0\Default")
}
```

Detection and Mitigation

Is there a good way of detecting this UAC bypass in use? Prior to Windows 10 it can be done pretty silently, a thread will magically become more privileged. So I suppose you might be able to detect that taking place, namely a elevated token being impersonated by a non-elevated process. For Windows 10 is should be easier as you need to do one or more dances with processes, at least as I've implemented it. I'm not much into the latest and greatest in DFIR, so perhaps someone else is better placed to look at this :-)

On the mitigation side it's simple:

DON'T USE SPLIT-TOKEN ADMINISTRATOR ACCOUNTS FOR ANYTHING YOU CARE ABOUT.

Or just don't get malware on your machine in the first place ;-) About the safest way of using Windows is to run as a normal user and use Fast User Switching to login to a new session with a separate administrator account. The price of Fast User Switching is the friction of hitting CTRL+ALT-DEL, then selecting Switch User, then typing in a password. Perhaps though that friction has additional benefits.

What about Over-The-Shoulder elevation, where you need to supply a username and password of a different user, does that suffer from the same problem? Due to the design of UAC those "Other User" processes also have the same Logon Session SID access rights so a normal, non-admin user can access the elevated token

in the same way. Admittedly just having the token isn't necessarily exploitable, but attacks only get better, would you be willing to take the bet that it's not exploitable?

Wrapping Up

The design behind UAC had all the hallmarks of trying to be secure and it turning out to be impossible to do so without severely compromising usability. So presumably it was ret-conned into something else entirely. Perhaps it's finally time for Microsoft to take UAC out the back and give it a proper sending off. I wonder if with modern versions of Windows the restrictions on compatibility can be dropped as UAC has served its purpose of acting as a forcing function to try and make applications behave better. Then again MS do seem to be trying to plug the leaks, which is surprising considering their general stance of it not being a security boundary, so I don't really know what to think.

Anyway, unless Microsoft change things substantially you should consider UAC to be entirely broken by design, in more fundamental ways than people perhaps realized (except perhaps the CIA). You don't need to worry about shared resources, bad environment variables, auto-elevating applications and the like. If malware is running in your split-token account you've given it Administrator access. In the worst case all it takes is patience, waiting for you to elevate once for any reason. Once you've done that you're screwed.

Posted by tiraniddo at 14:35
Labels: Exploit, UAC, Windows

Newer Post Home Older Post