Project Zero

News and updates from the Project Zero team at Google

Tuesday, January 12, 2016

Raising the Dead

Posted by James Forshaw, your Friendly Neighbourhood Necromancer.

It's a bit late for Halloween but the ability to resurrect the dead (processes that is) is an interesting type of security issue when dealing with multi-user Windows systems such as Terminal Servers. Specifically this blog is about this issue which I reported to Microsoft and was fixed in bulletin MS15-111. On the surface it looks like at best this would be a local Denial of Service, in fact Microsoft wasn't even sure it was a Elevation of Privilege. By the end of the post I'll show why it was an EoP vulnerability, and how you could exploit it.

Terminal Services Background

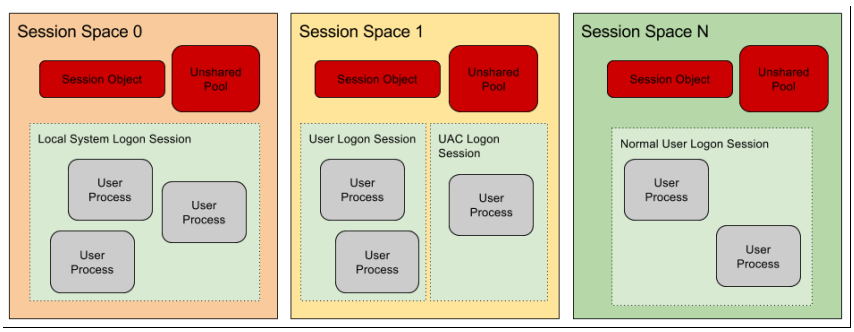
The original versions of Windows NT (from 3.1) were multi-user in principle. NT could have multiple users interacting on a single workstation or server at one time, however only one user could be logged into the physical console and there was no such thing as virtual consoles. This made remote administration of servers tricky (especially with the amount of functionality only exposed through a GUI) as well as making it impossible to share a computer between multiple users without having to log the other user out each time.

The solution to this was the development of Terminal Services. This wasn't even developed by Microsoft originally but Citrix. Unfortunately as the OS was designed expecting only a single GUI session at one time, some clever hacks had to employed to get around the problem. In this case the development of the concept of independent *Sessions*. Since XP, support for multiple Sessions has been included with all released editions; even if you cannot specifically use the OS as a Terminal Server.

The major change introduced to support Sessions is the kernel Session Space, represented in the kernel by the *MM_SESSION_SPACE* structure. This structure maintains all the information about a particular session, including a reference to a kernel Session Object and a list of the processes within that Session. It also includes information about a special area of Pool Memory which is used to store data for the GUI of that session. The pool's virtual address range must be remapped whenever a thread context switch occurs and the current process is in a different Session. This hacky approach is required because the GUI was originally designed for a single terminal and so expects memory to be laid out in the same way all the time. Each Session Space is allocated a 32 bit unique identifier, the Session ID.

To add some extra confusion a Session is usually associated with a Logon Session. Even though they share similar names they're not the same thing. A Logon Session represents a particular logon of a user which is associated with an access token. This is generated by the kernel when creating a new token with a unique Locally Unique Identifier (*LUID)* (which is assigned by *LSASS*). You could have multiple Logon Sessions in a single Session Space (for example this is how *UAC* works, with the Filtered Token and Elevated Token being separate Logon Sessions) or you could have a Logon Session which spans multiple Session Spaces. There's also some hard coded Logon Sessions for the Local System and Service Accounts.

How these all interact I've summarised in the below diagram.



The final question you might have is how a process is assigned to a session? For that we need to look at the access token. The kernel *TOKEN* structure has a *SessionId* field. This is just a single 32 bit integer which represents the Session ID. When creating a new process the kernel will look up the appropriate Session Space based on this number and assign a reference to it to the *EPROCESS::Session* field (the kernel also adds the process to the Session Space's internal process list). We can modify this *SessionId* field before assigning the token to a new process using the SetTokenInformation API passing the *TokenSessionId* information class. You can only do this if you have *SeTcbPrivilege* which normally means only when running as Local System. It's worth knowing about as you can use this to spawn interactive processes on the current

Search This Blog
Search

Labels
- antivirus
- exploit
- fireeye
- security

Archives
▼ 2016 (1)
▼ January (1)
Raising the Dead
► 2015 (33)
► 2014 (11)

desktop with a Local System token. For example the following code will spawn the command prompt on the current physical desktop, useful to test Local System elevation of privilege vulnerabilities:

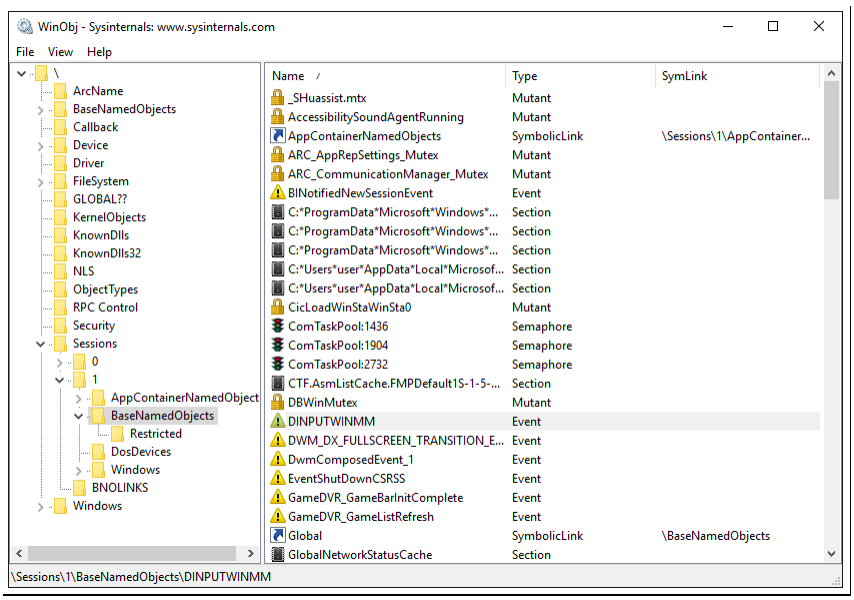
```
void StartSystemProcessInConsole() {
  STARTUPINFO startInfo = { 0 };
  PROCESS_INFORMATION procInfo = { 0 };

  startInfo.cb = sizeof(startInfo);
  HANDLE hToken;
  DWORD sessionId = WTSGetActiveConsoleSessionId();

  WCHAR cmdline[] = L"cmd.exe";

  OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &hToken);
  DuplicateTokenEx(hToken, TOKEN_ALL_ACCESS,
                   nullptr, SecurityAnonymous, TokenPrimary, &hToken);
  SetTokenInformation(hToken, TokenSessionId, &sessionId, sizeof(sessionId));

  startInfo.wShowWindow = SW_SHOW;
  startInfo.lpDesktop = L"WinSta0\\Default";

  if (CreateProcessAsUser(hToken, nullptr, cmdline, nullptr, nullptr, FALSE,
                          NORMAL_PRIORITY_CLASS | CREATE_NEW_CONSOLE,
                          nullptr, nullptr, &startInfo, &procInfo)) {
    CloseHandle(procInfo.hProcess);
    CloseHandle(procInfo.hThread);
  }
}
```

The Vulnerability

Enough about the background, let's get to the bug itself. You can read the writeup on the issue tracker here. The root cause of the vulnerability is the *NtCreateLowBoxToken* system call introduced in Windows 8 to support the creation of locked down tokens for Immersive Applications (formerly known as Metro) as well as IE11's Enhanced Protected Mode.

The LowBox token was an aim to replace the older Restricted token present since Windows 2000 with a more consistent and supported set of features. One of the biggest problems with managing Restricted tokens was allowing access to resources, especially ones which were baked into code the developer had little control over.

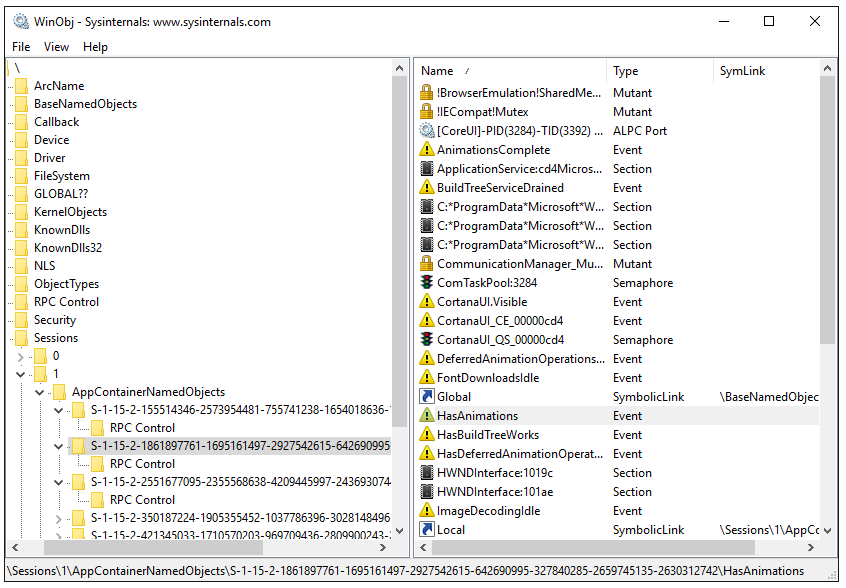
Take for example named kernel objects such as events. You can create a new named event using the CreateEvent API or open an existing named event with OpenEvent. Each API takes the *lpName* parameter which specified the name of the event to create. So where does the actual name get registered? Under the hood these APIs call a function, *BaseGetNamedObjectDirectory* which converts the name to a per-session Object Manager Namespace path of the form \Sessions\X\BaseNamedObjects\EVENTNAME, which is passed to the underlying system call responsible for handling events.



The *BaseNamedObjects* directory is a securable resource, and is intentionally locked down to prevent heavily restricted processes adding or modifying resources within it. So as there's little control over the location of the name (not 100% true, but true enough for this discussion) a low privilege application might fail if it tried to ever create a named resource. If this is in a third party library or component it might cause the restricted process to crash or otherwise misbehave.

When developing the LowBox token model this was still going to be a problem so Microsoft decided to add a transparent workaround. They did this by adding a new directory, *AppContainerNamedObjects* to the per-session directory. When a new AppContainer process is started the creator also adds a few new directories

for supporting named resources for a specific Package SID (which represents the identity of the LowBox token). The directories have their DACL set so only the specific LowBox token can access them. The *BaseGetNamedObjectDirectory* function is changed to return this specific directory when running under a LowBox token. This neatly solves the problem, for the most part, and requires no, or at least very little, changes to existing code.



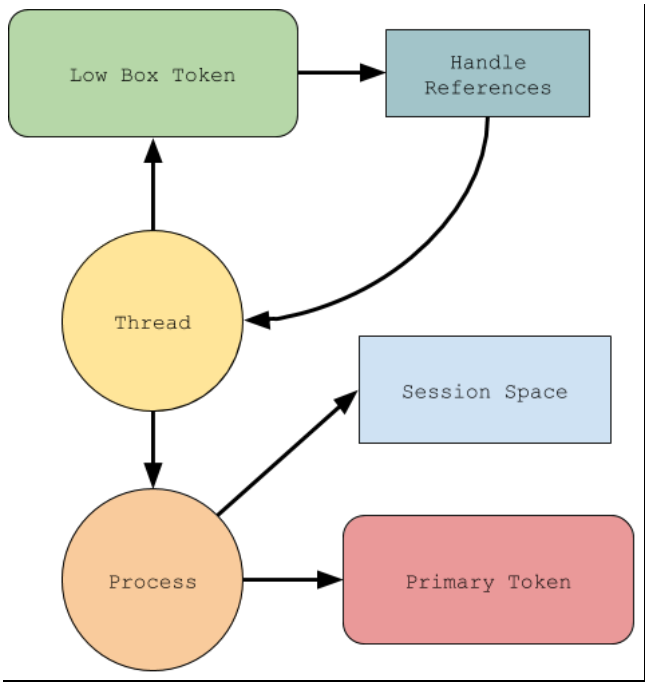
This still leaves one problem, kernel object lifetime. The name of a kernel object only lasts as long as there's a reference to that object either in the kernel or a user mode handle. When the reference count goes to 0 the object is destroyed along with its name. The original way of ensuring a named kernel object continues to exist even if there's no reference to it is to create the object with the OBJ_PERMANENT flag (or call *NtMakePermanentObject*). Unfortunately this requires the *SeCreatePermanentPrivilege* which is typically only granted to Local System, not very useful for a user-mode sandbox.

Instead of requiring a system service or weakening the existing permanent object support the *NtCreateLowBoxToken* API supports capturing handle references. The system call takes an array of user handles. During token creation the system call references each handle and stores them in a reference counted structure within the TOKEN structure. When a LowBox token is created normally this array is filled with the AppContainer specific object directories and supporting kernel objects. The references are only released when the token kernel object is deleted (or any of its duplicates), which effectively means they last as long as the last process exists with that token. This seems a practical solution, effectively the lifetime of the kernel objects necessary for supporting normal operation are self managed. When there's no longer any process which needs them they'll be automatically deleted. Of course this functionality is what I abused in issue 483.

The vulnerability is the kernel didn't verify what types of kernel objects it was referencing, it just captured the handles and that was that. I realized that this might be something which you could exploit to generate a reference cycle which would have some interesting consequences. Specifically we can use this prevent a process from truly be deleted even when the user logged out of the system. So let's look at how we might create the reference cycle and then what we could do with it.

Voodoo Magic

The prerequisites for creating a reference cycle is we must be able to get a handle to a kernel object before calling *NtCreateLowBoxToken*, this kernel object must then be able to take a reference to the lowbox token. There's two kernel object types which fit this definition, Processes and Threads. For our purposes Threads make the most sense. A thread can have an impersonation token set after the thread is created. The thread maintains a reference to the token object so if we assign a LowBox token as the thread's impersonation token which has a reference to the thread itself we get our cycle. Handily the thread also maintains a reference to its process, which in turn maintains a reference to both the Session Space object and its primary token. The following diagram show these relationships.



At this point you might think, “Big deal when the user logs out the process will be terminated so you can’t do anything useful” and you’d be right the system will terminate the process when you log out. That doesn’t mean the process goes away, all it means is the kernel stops the process running, unschedules the threads and cleans up the process’s virtual memory.

Many kernel resources are lazily dereferenced; for example until the thread’s kernel object is deleted (by its reference count going to zero) it won’t release its reference to the token object. In this case the token object holds a reference to the thread, which prevents it being deleted which keeps the token alive. Of course the thread also keeps the process around, however from the systems perspective the process is dead and buried. The process cannot execute any more code, which is a good thing, but its undead status can still be abused.

An interesting aspect is the process doesn’t show up in applications like Task Manager or Process Explorer, but it’s still there, at least the remnants of its kernel object. Even if it was visible it’s not possible to terminate, as it’s already terminated. If you know the process ID of the terminated process you can reopen it and fix the reference cycle, but it’s not possible to even know it exists. It should be reasonably obvious how this could be abused for a local denial of service. Open handles to important resources, such as system files and capture them in the reference cycle. The user can then log out and the only thing the system administrator could do about it would be to reboot the server as only a custom tool is able to break the reference cycle and free the resources. This is pretty boring though, is there a way of taking this bug and using it to elevate privileges? Of course, but it’ll start to get complicated.

Exploiting for Elevation of Privilege

Before I can start explaining how you can use this vulnerability to elevate privileges I first need to define what I mean by elevation of privilege in this context. Elevating your privilege typically means gaining objectively higher privileges such as the level of Administrator. But it can equally refer to gaining access to another user account. With this in mind the goal I’ll set for this exploit is to get arbitrary code running as another user on the server.

Of course as I’ve hinted there has to be another restriction here, that we need other users to log in to the same machine while we maintain some level of control. This is only really possible on something like a shared Terminal Server. So in this case we’ll exploit the bug on a Windows Server 2012 machine with Remote Desktop Services enabled. As Windows Server 2012 is based on the Windows 8 code base it contains the vulnerable version of *NtCreateLowBoxToken*. We can create the reference cycle using the following code:

```
HANDLE CreateToken(HANDLE hProcessToken) {
  BOOL bRet = FALSE;
  HANDLE hRetToken = nullptr;
  HANDLE hThread = CreateThread(nullptr, 0, DummyFunc, nullptr, 0, nullptr);
  HANDLE hLowBoxToken;
  PSID psid;
  ConvertStringSidToSid(L"S-1-15-2-1-1-1-1-1-1-1", &psid);
  // Create the reference cycle with the thread handle.
  NtCreateLowBoxToken(&hLowBoxToken, hProcessToken, ..., psid,
                      0, nullptr, &hThread, 1);
  HANDLE hImpToken;
  DuplicateToken(hLowBoxToken, SecurityImpersonation,
                 TokenImpersonation, &hImpToken);
  SetThreadToken(&hThread, hImpToken);

  return hLowBoxToken;
```

```
}
```

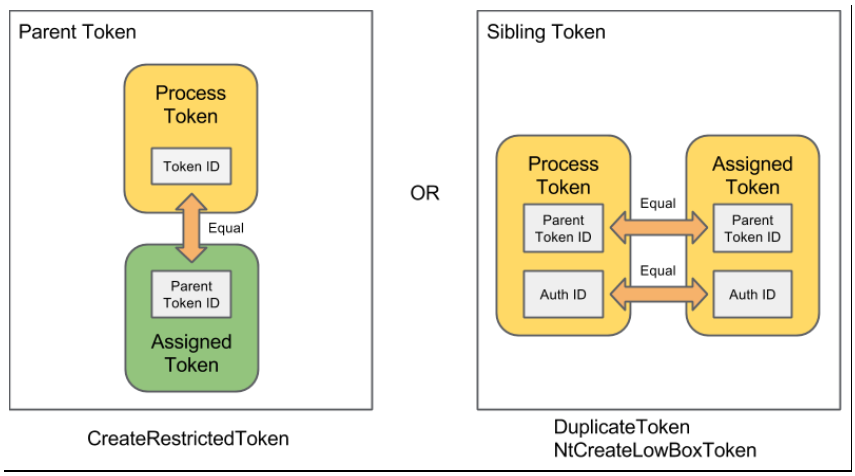
What can we do with this vulnerability which might allow us to achieve the stated goal? The first observation is that based on the reference cycle diagram shown earlier access tokens do not hold a reference to the Session Space which the token's Session ID field refers. If we set the reference cycle, log out then back in again we can reopen the dead process. As the process's' primary token is lazily destroyed we can open the process and take a copy of its token. With this token we can create a new process in a Session Space with the original ID. As I've already mentioned changing the Session ID of an existing token requires TCB privilege, which only Local System gets by default. But of course in this case we've already got the system to assign the original ID, we're just repurposing it. You could think of this like a Session ID Use-After-Free, the original Session Space is gone but we still have a reference to a token with that ID. To avoid having to save the old process ID somewhere we'll use the *NtGetNextProcess* system call to walk all processes looking for a process we can open but isn't in our current session.

```
HANDLE GetOldProcessToken() {
  HANDLE hCurr = nullptr;
  DWORD dwCurrSession;
  ProcessIdToSessionId(GetCurrentProcessId(), &dwCurrSession);

  while (NtGetNextProcess(hCurr, MAXIMUM_ALLOWED, 0, 0, &hCurr) == 0) {
    DWORD dwPid = GetProcessId(hCurr);
    DWORD dwSession = 0;
    ProcessIdToSessionId(dwPid, &dwSession);
    if (dwSession == dwCurrSession)
      continue;
    HANDLE hToken;
    if (OpenProcessToken(hCurr, MAXIMUM_ALLOWED, &hToken)) {
      CloseHandle(hCurr);
      return hToken;
    }
  }
  return nullptr;
}
```

Now at this point we can delete the original Session Space by clearing the impersonation token on the stuck thread. This breaks the reference cycle which causes all the resources to be closed. This leaves the Session ID ready to be reused. So at this point the malicious user just lays in wait for another user to log in to the server resulting in the system allocating a new Session Space with the old Session ID. We can use the Terminal Services APIs to enumerate current active sessions until the Session ID we want is reused.

```
void WaitForSession(DWORD dwSessionId)
{
  BOOL bSessionFound = FALSE;
  while (!bSessionFound) {
    PWTS_SESSION_INFO pSessions;
    DWORD pSessionCount;
    WTSEnumerateSessions(WTS_CURRENT_SERVER_HANDLE,
                         0, 1, &pSessions, &pSessionCount));
    for (DWORD i = 0; i < pSessionCount; ++i) {
      if ((pSessions[i].SessionId == dwSessionId)
          && (pSessions[i].State == WTSActive)) {
        bSessionFound = TRUE;
      }
      WTSFreeMemory(pSessions);
      Sleep(1000);
   }
}
```

At this point we can now create a new process in the newly created Session Space. Unfortunately we'll immediately hit a snag. As a normal user we don't have *SeAssignPrimaryTokenPrivilege*. This means the kernel will do a check on the token being assigned relative to the current process token. This check is done in the *SeIsTokenAssignableToProcess* kernel function; it will succeed if either the assigned token is a child of the current process token or it's a sibling token. These relationships are based on the Logon Session LUID as shown in the following diagram.



We can't satisfy the parent/child relationship as the current process token is totally unrelated to the captured token. We also cannot satisfy the sibling token requirement either as the Auth ID (which is actually the Logon Session LUID) is different to the current ID. Does this mean we're stuck? Well of course not, all we need is a system service which will create a new process with our specified token. As the system service will have *SeAssignPrimaryTokenPrivilege* it should succeed. What we're looking for is something similar to the following code.

```
void IdealProcessCreator(LPWSTR CommandLine) {
  HANDLE hToken;
  ImpersonateCaller();
  OpenThreadToken(GetCurrentThread(), ..., &hToken);
  CreateProcessAsUser(hToken, CommandLine, ...);
}
```

This ideal code impersonates the caller, for example via RPC/DCOM or Named Pipes, opens the thread token and then creates the new process with that token. We can impersonate the captured token because the rules for what tokens you can impersonate differ from process creation. This of course isn't a security issue in normal circumstances as you'll only ever create a process with the original callers privilege, but in this case it's a problem because the thread token refers to a different Session ID. The next question is does such code exist on the system? Turns out there's plenty of places this kind of code is used, but the simplest one to exploit is the Create method on the WMI Win32_Process class. The following code is a simplified example of how to use this to create an arbitrary process. Note that the *EOAC_STATIC_CLOAKING* flag must be set on the COM proxy otherwise the process token ends up being used for impersonation instead, which is of course massively confusing when the new process doesn't have the expected Session ID.

```
void StartWmi(HANDLE hProcessToken)
{
  HANDLE hImpToken;
  DuplicateToken(hProcessToken, SecurityImpersonation, &hImpToken)
  ImpersonateLoggedOnUser(hImpToken))

  IWbemLocatorPtr pLoc;
  CoCreateInstance(
    CLSID_WbemLocator,
    0,
    CLSCTX_INPROC_SERVER,
    IID_PPV_ARGS(&pLoc));

  IWbemServicesPtr pSvc;
  pLoc->ConnectServer(L"ROOT\\CIMV2", ..., &pSvc);
  CoSetProxyBlanket(pSvc, ...,  EOAC_STATIC_CLOAKING);

  IWbemClassObjectPtr pClass;
  pSvc->GetObject(L"Win32_Process", 0, nullptr, &pClass, nullptr);

  IWbemClassObjectPtr pInParamsDefinition;
  hres = pClass->GetMethod(L"Create", 0,
    &pInParamsDefinition, nullptr);

  IWbemClassObjectPtr pClassInstance = nullptr;
  pInParamsDefinition->SpawnInstance(0, &pClassInstance);
  pClassInstance->Put(L"CommandLine", 0, L"C:\\temp\\ExploitProcess.exe", 0);

  IWbemClassObjectPtr pOutParams;
  hres = pSvc->ExecMethod(ClassName, MethodName, 0,
    nullptr, pClassInstance, &pOutParams, nullptr);
}
```

This will create the new process in the new Session Space, but there's still a few problems. First when the new process tries to access per-session object directories (such as *BaseNamedObjects*) it will fail to initialize as these directories are giving permissions only allowing the Session User or the Login Session LUID access. Fortunately this is pretty easy to solve, the lifetime of these directories are maintained by reference counting and we already have a way of keeping kernel objects alive through referencing them in a LowBox

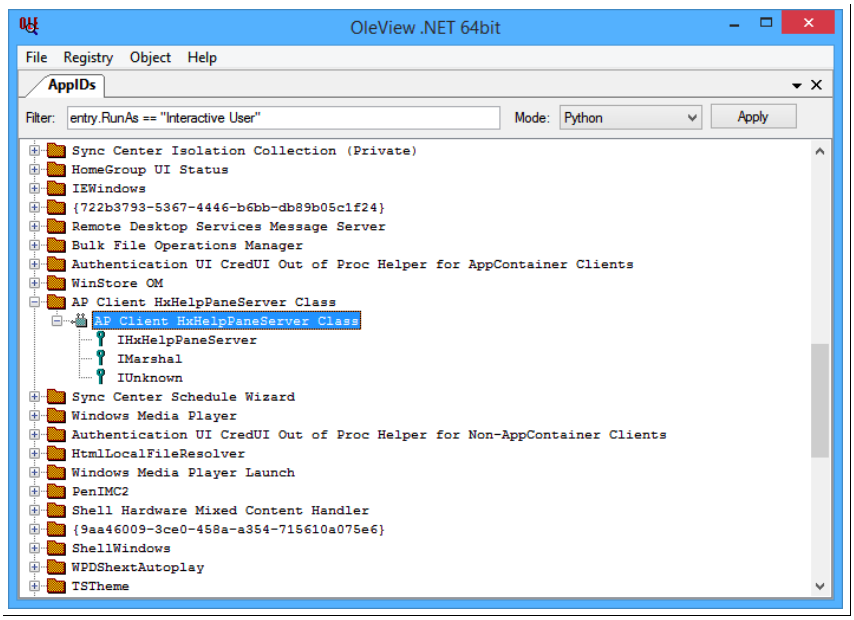
token. At minimum we need to capture *\Sessions\X* and *\Sessions\X\BaseNamedObjects*. When a new user is logged in CSRSS creates these directories, fortunately it doesn't care if they already exist. CSRSS will reset their permissions to only allow the new user access, but we can open a handle to them before the new user logs in with *WRITE_DAC* permission and then reset the DACL to restore access. This in itself is a way of interacting with processes in the new Session Space (by access their named kernel resources) but we'll use a different technique to get a process spawned as the new user.

The second problem we encounter is a new Windows Station and Desktop is created when the user logs in. Unfortunately our new process doesn't have access to either the Windows Station or the Desktop which means that we can't interact with the Windowing system (we can't even load USER32). If we could access the Desktop we'd be able to send Window messages to the explorer shell to spawn an arbitrary process. We need another technique to spawn our arbitrary process. This is where the confusion between the Session Space and the Logon Session works to our advantage.

If you look at the Terminal Services API you'll find a method called WTSQueryUserToken. This API takes the Session ID and returns a token for the user associated with the Session Space, which is the first user who was logged into the Session. While we can't call it from a low privileged user we might be able to find a system service which will do it for us. Some ideal code would be something where the caller's token (or process, it doesn't really matter which) is used to determine the current session but the process is created with the token returned from *WTSQueryUserToken*.

```
void IdealSessionProcessCreator(LPWSTR CommandLine) {
  HANDLE hImpToken;
  DWORD dwSessionId;
  ImpersonateCaller();
  OpenThreadToken(GetCurrentProcess(), ..., &hImpToken);
  QueryTokenInformation(hImpToken, TokenSessionId, &dwSessionId);

  HANDLE hToken;
  WTSQueryUserToken(dwSessionId, &hToken);

  CreateProcessAsUser(hToken, CommandLine, ...);
}
```

There seems to be fewer of these sort of services compared to ones which create based on the calling user, but one service which will work is the DCOM activator. An out-of-process COM object can be registered with an Application ID (AppID) which specifies through the RunAs setting which user account to the run the executable under. There's a special value "Interactive User" which turns out doesn't mean "Run as the caller" but "Run as the Session User". Unfortunately we can't register our own COM object with one of these AppIDs for security reasons, but we can reuse an existing registration and try and use that to spawn an arbitrary process. First we need to find what COM objects are registered to run as "Interactive User", for that I'll use my OleViewDotNet tool with an appropriate filter.



After a bit of searching I found the ideal candidate, the undocumented *HxHelpPaneServer* class. This class is accessible by any user and has an *Execute* method on the *IHxHelpPaneServer* interface which for all intents and purposes just passes a string to *ShellExecute*. As this server is running as the Session user and not the caller any new process will be created as the Session user, and there ends the quest for an elevation of privilege.

```
struct __declspec(uuid("8cec592c-07a1-11d9-b15e-000d56bfe6ee"))
  IHxHelpPaneServer : public IUnknown {
  virtual HRESULT __stdcall DisplayTask(wchar_t*) = 0;
  virtual HRESULT __stdcall DisplayContents(wchar_t*) = 0;
  virtual HRESULT __stdcall DisplaySearchResults(wchar_t*) = 0;
  virtual HRESULT __stdcall Execute(const wchar_t*) = 0;
};
```

```
void CreateExploitProcess() {
  CoInitializeEx(NULL, COINIT_MULTITHREADED);
  CLSID clsid;
  CLSIDFromString(L"{8cec58ae-07a1-11d9-b15e-000d56bfe6ee}", &clsid);
  IHxHelpPaneServer* pServer;
  CoCreateInstance(clsid, nullptr, CLSCTX_LOCAL_SERVER,
                   IID_PPV_ARGS(&pServer));
  pServer->Execute(L"file:///c:/temp/ExploitProcess.exe");
  CoUninitialize();
}
```

So in summary these are the steps to getting a process running as a user who logs into the same terminal server as the malicious user.

1. Log in to the terminal server as the malicious user
2. Setup a reference cycle to keep a process object from being deleted as well as the session and named object directories
3. Logout then log back in again as the same user
4. Reopen the stuck process, open a handle to the process token with the old session ID as well as handles to the user's session and named object directories
5. Unset the stuck thread's token, this will cause the reference cycle to break which will delete the process and Session Space objects
6. Wait for another user to log into the terminal server and reuse the session ID in the captured process token. Reset the security on the named object directory which was set by CSRSS
7. Spawn a new process with the captured token using *Win32_Process* WMI class
8. Use *HxHelpPaneServer* COM object to execute an arbitrary executable as the Session's User

Conclusions

This blog post was about a very odd vulnerability, one where it wasn't even clear from the outset that it was usefully exploitable to elevate privileges. Hopefully I've demonstrated one chain which could be used to exploit it, but no doubt there are more. It's interesting that there's systemic problems with the nomenclature of Sessions and whether they refer to a Session Space or a Logon Session and the consequences of mistaking one for the other. I'm sure there's probably other vulnerabilities which would provide a similar primitive waiting to be found.