x86matthew - SharedMemUtils - A simple tool to automatically find vulnerabilities in shared memory objects

SharedMemUtils - A simple tool to automatically find vulnerabilities in shared memory objects

20/10/2022

Following on from my previous post regarding exploit development, this post describes another common weakness in services that can often lead to successful exploitation.

Shared memory is a very common method of inter-process communication (IPC) on Windows - these are often combined with event objects as follows:

(Service initialisation)

1. Process #1 (service) creates a named section using CreateFileMapping.
2. Process #1 (service) creates a named event using CreateEvent.
3. Process #1 (service) waits for the event to be triggered using WaitforSingleObject.

(Communication with service via a user process)

4. Process #2 (user) wants to transfer some data to Process #1 (service) - the user process opens the named section using OpenFileMapping.
5. Process #2 (user) maps the shared section into memory using MapViewOfFile.
6. Process #2 (user) copies the relevant data into the memory region.

(Notify the service that the memory has been updated)

7. Process #2 (user) opens the named event using OpenEvent.
8. Process #2 (user) triggers the event using SetEvent.

(Service receives data)

9. Process #1 (service) receives notification of new data from previous WaitforSingleObject call.
10. Process #1 (service) maps the shared section into memory using MapViewOfFile.
11. Process #1 (service) reads data.

Due to the nature of these shared memory objects being used for IPC purposes, it is usually possible to open and write to them as a low-privileged user. This opens up the possibility of exploitation if a high-privileged service performs weak validation checks against the input data.

To experiment with this theory, I created a brute-force tool to enumerate all of the shared memory objects on the system and overwrite their contents with a (0x61) characters. As mentioned above, each shared memory object used in this way generally has a corresponding event object to notify the receiving process of any changes. As there is no way to pair these objects automatically, this tool will trigger all named event objects on the system after updating the shared memory sections.

A brief summary of how this tool works:

1. Call NtQuerySystemInformation with the SystemProcessInformation class type to retrieve a list of all handles in the system.
2. Enumerate all handles in the list and identify all objects with an ObjectTypeIndex value of Section.

3. For the first section object found, call OpenProcess to open the process that owns this handle (UniqueProcessId).
4. Call DuplicateHandle to copy the handle (HandleValue) from the remote process into the local process.
5. Call NtQueryObject with the ObjectNameInformation class to retrieve the full path of this section object.
6. Check if this object exists within BaseNamedObjects - objects within sandboxed containers should be ignored.
7. Call MapViewOfFile to map this section into the virtual memory of the local process.
8. Call VirtualQuery to retrieve the total size of the shared memory region.
9. Overwrite all bytes in the section with a characters.
10. Call UnmapViewOfFile to close the memory mapping.
11. Return to step #3 for the next section handle. Continue until all sections have been processed.
12. Identify all handles with an ObjectTypeIndex value of Event using the same steps as above.
13. Call SetEvent to trigger each event object.

In practice, the tool works in a slightly different way than described above. For best results, it should be split into two parts:

1. Enumerate all shared memory sections and write the names to a file. This step should be performed with elevated permissions (Administrator) with SeDebugPrivilege enabled.
2. Attempt to open all of the section names listed in the file created above and overwrite them with a characters. This step should be performed as a normal user.

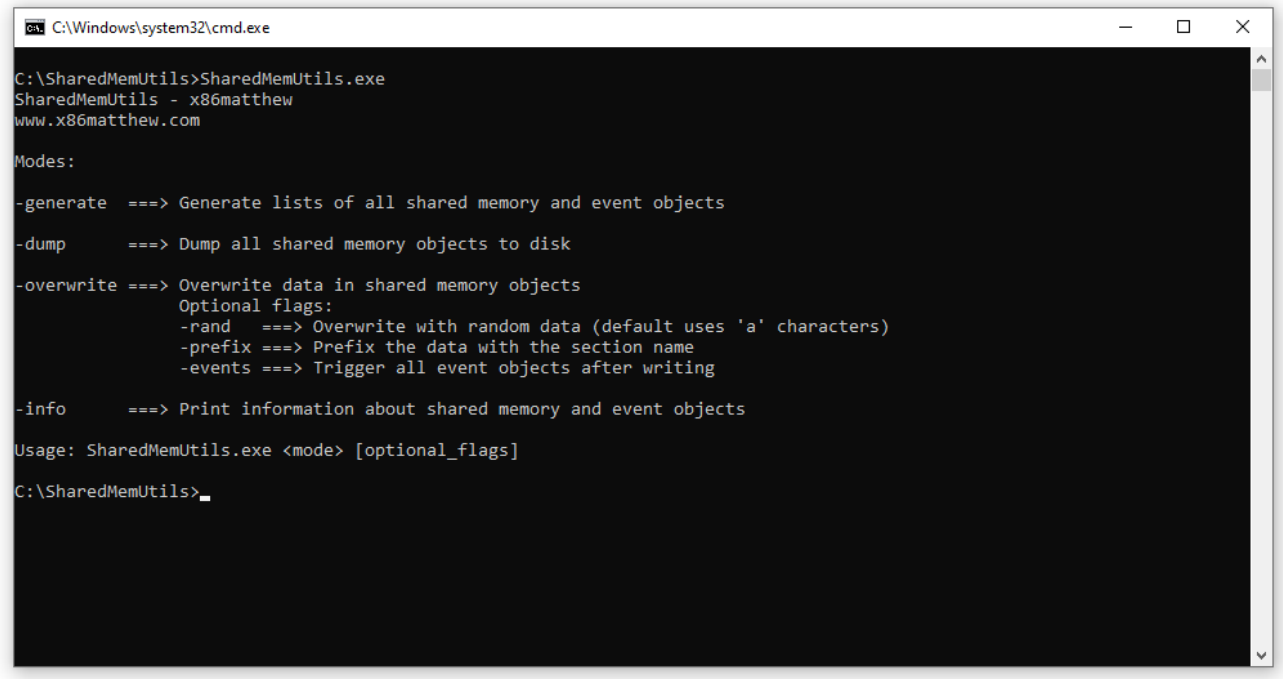
The reason for splitting it into two steps is because a normal user will be unable to duplicate section handles from high-privilege processes - for best results, we should generate the list of section names while running as Administrator. However, we don't want to attempt write to these sections while running as Administrator - we should attempt this as a normal user instead. If Administrator rights are necessary to open an shared memory object, it should be ignored.

Note: In the steps described above, we need to check if the ObjectTypeIndex has a value of Section or Event. Unfortunately, these values differ by OS version. Most tools hardcode these values, but there is a simple way to reliably calculate the object type values. Pseudo-code below:

```
DWORD GetEventObjectTypeIndex()
{
HANDLE hTempEvent = NULL;
DWORD dwObjectTypeIndex = 0;

// create a temporary event object
hTempEvent = CreateEvent();

// get a list of handles on this system
GetHandleList();

// enumerate all handles
for(DWORD i = 0; i < dwHandleCount; i++)
{
// check if the current handle belongs to this process
if(HandleList[i].UniqueProcessId == GetCurrentProcess())
{
// check if the current handle is our temporary event object
if(HandleList[i].HandleValue == hTempEvent)
{
```

```
// store object type index
dwObjectTypeIndex = HandleList[i].ObjectTypeIndex;
break;
}
}
}

// delete temporary event object
CloseHandle(hTempEvent);

return dwObjectTypeIndex;
}
```

Testing the tool



In most cases, this tool should be used as follows:
1. Execute SharedMemUtils.exe -generate as Administrator. This will generate two files - shared_memory_list.txt and event_list.txt.
2. Execute SharedMemUtils.exe -overwrite -prefix -events as a normal user. This will overwrite all of the shared memory sections listed in shared_memory_list.txt, and trigger all events listed in event_list.txt.

The -prefix flag will write the section name to the beginning of all overwritten regions. This makes it easy to identify which section caused the crash.

The first test of this tool immediately revealed several interesting crashes - some of which appeared to allow arbitrary memory writes under the correct circumstances, which could easily become a precursor to exploitation.
Other than one program which I have successfully exploited using this method, I haven't attempted to take these examples any further - this is left as an exercise for the reader.

Another feature of this tool (-dump parameter) allows us to dump the contents of all shared memory objects to disk - this may be useful when targeting specific applications.

Full code below:

```
#include <stdio.h>
#include <windows.h>

#define ObjectNameInformation 1
#define SystemProcessInformation 5
#define SystemExtendedHandleInformation 64
#define STATUS_INFO_LENGTH_MISMATCH 0xC0000004

#define MODE_GENERATE 1
#define MODE_DUMP 2
#define MODE_OVERWRITE 3
#define MODE_INFO 4

#define SHARED_MEMORY_LIST_FILENAME "shared_memory_list.txt"
#define EVENT_LIST_FILENAME "event_list.txt"

struct SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX
{
PVOID Object;
PVOID UniqueProcessId;
PVOID HandleValue;
ULONG GrantedAccess;
USHORT CreatorBackTraceIndex;
USHORT ObjectTypeIndex;
ULONG HandleAttributes;
ULONG Reserved;
};

struct SYSTEM_HANDLE_INFORMATION_EX
{
PVOID NumberOfHandles;
PVOID Reserved;
SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX HandleList[1];
};

struct UNICODE_STRING
{
USHORT Length;
USHORT MaximumLength;
PWSTR Buffer;
};

struct SYSTEM_PROCESS_INFORMATION
{
ULONG NextEntryOffset;
ULONG NumberOfThreads;
LARGE_INTEGER WorkingSetPrivateSize;
ULONG HardFaultCount;
ULONG NumberOfThreadsHighWatermark;
BYTE CycleTime[8];
LARGE_INTEGER CreateTime;
LARGE_INTEGER UserTime;
LARGE_INTEGER KernelTime;
```

```
UNICODE_STRING ImageName;
DWORD BasePriority;
HANDLE UniqueProcessId;
HANDLE InheritedFromUniqueProcessId;
ULONG HandleCount;
ULONG SessionId;
DWORD *UniqueProcessKey;
SIZE_T PeakVirtualSize;
SIZE_T VirtualSize;
ULONG PageFaultCount;
SIZE_T PeakWorkingSetSize;
SIZE_T WorkingSetSize;
SIZE_T QuotaPeakPagedPoolUsage;
SIZE_T QuotaPagedPoolUsage;
SIZE_T QuotaPeakNonPagedPoolUsage;
SIZE_T QuotaNonPagedPoolUsage;
SIZE_T PagefileUsage;
SIZE_T PeakPagefileUsage;
SIZE_T PrivatePageCount;
LARGE_INTEGER Reserved7[6];
};

DWORD (WINAPI *pNtQueryObject)(HANDLE Handle, DWORD ObjectInformationClass, PVOID
ObjectInformation, DWORD ObjectInformationLength, DWORD *ReturnLength) = NULL;
DWORD (WINAPI *pNtQuerySystemInformation)(DWORD SystemInformationClass, PVOID
SystemInformation, ULONG SystemInformationLength, PULONG ReturnLength) = NULL;

SYSTEM_HANDLE_INFORMATION_EX *pGlobal_SystemHandleInfo = NULL;
SYSTEM_PROCESS_INFORMATION *pGlobal_SystemProcessInfo = NULL;

DWORD dwGlobal_EventObjectCount = 0;
char **pGlobal_EventObjectNameList = NULL;

DWORD dwGlobal_SectionObjectCount = 0;
char **pGlobal_SectionObjectNameList = NULL;

DWORD dwGlobal_EventObjectType = 0;
DWORD dwGlobal_SectionObjectType = 0;

BYTE *GetSystemInformationBlock(DWORD dwSystemInformationClass)
{
DWORD dwAllocSize = 0;
DWORD dwStatus = 0;
DWORD dwLength = 0;
BYTE *pSystemInfoBuffer = NULL;

// get system process list
dwAllocSize = 0;
for(;;)
{
if(pSystemInfoBuffer != NULL)
{
// free previous inadequately sized buffer
```

```
free(pSystemInfoBuffer);
pSystemInfoBuffer = NULL;
}

if(dwAllocSize != 0)
{
// allocate new buffer
pSystemInfoBuffer = (BYTE*)malloc(dwAllocSize);
if(pSystemInfoBuffer == NULL)
{
return NULL;
}
}

// query system info block
dwStatus = pNtQuerySystemInformation(dwSystemInformationClass, (void*)pSystemInfoBuffer,
dwAllocSize, &dwLength);
if(dwStatus == 0)
{
// success
break;
}
else if(dwStatus == STATUS_INFO_LENGTH_MISMATCH)
{
// not enough space - allocate a larger buffer and try again (also add an extra 1kb to allow for additional
data between checks)
dwAllocSize = (dwLength + 1024);
}
else
{
// other error
free(pSystemInfoBuffer);
return NULL;
}
}

return pSystemInfoBuffer;
}

DWORD GetSystemProcessInformation()
{
// free previous process info list
if(pGlobal_SystemProcessInfo != NULL)
{
free(pGlobal_SystemProcessInfo);
}

// get system process list
pGlobal_SystemProcessInfo =
(SYSTEM_PROCESS_INFORMATION*)GetSystemInformationBlock(SystemProcessInformation);
if(pGlobal_SystemProcessInfo == NULL)
```

```
{
return 1;
}

return 0;
}

DWORD GetSystemHandleList()
{
// free previous handle info list
if(pGlobal_SystemHandleInfo != NULL)
{
free(pGlobal_SystemHandleInfo);
}

// get system handle list
pGlobal_SystemHandleInfo =
(SYSTEM_HANDLE_INFORMATION_EX*)GetSystemInformationBlock(SystemExtendedHandleInformation);
if(pGlobal_SystemHandleInfo == NULL)
{
return 1;
}

return 0;
}

DWORD GetHandleObjectType(HANDLE hHandle, DWORD *pdwObjectType)
{
DWORD dwFoundObjectType = 0;
DWORD dwObjectType = 0;

// get system handle list
if(GetSystemHandleList() != 0)
{
return 1;
}

// find the specified handle in the list
for(DWORD i = 0; i < (DWORD)pGlobal_SystemHandleInfo->NumberOfHandles; i++)
{
// check if this handle is for the current process
if((DWORD)pGlobal_SystemHandleInfo->HandleList[i].UniqueProcessId != GetCurrentProcessId())
{
continue;
}

// check if the handle index matches
if(pGlobal_SystemHandleInfo->HandleList[i].HandleValue == hHandle)
{
dwFoundObjectType = 1;
dwObjectType = pGlobal_SystemHandleInfo->HandleList[i].ObjectTypeIndex;
}
}
```

```
// ensure the handle type was found
if(dwFoundObjectType == 0)
{
return 1;
}

// store debug object type
*pdwObjectType = dwObjectType;

return 0;
}

DWORD GetEventObjectType(DWORD *pdwObjectType)
{
HANDLE hTempHandle = NULL;

// create temporary event handle
hTempHandle = CreateEvent(NULL, 0, 0, NULL);
if(hTempHandle == NULL)
{
return 1;
}

// get object type
if(GetHandleObjectType(hTempHandle, pdwObjectType) != 0)
{
// error
CloseHandle(hTempHandle);

return 1;
}

// close temporary handle
CloseHandle(hTempHandle);

return 0;
}

DWORD GetSectionObjectType(DWORD *pdwObjectType)
{
HANDLE hTempHandle = NULL;

// create temporary section handle
hTempHandle = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE,
0, 1024, "temp_section_name");
if(hTempHandle == NULL)
{
return 1;
}

// get object type
if(GetHandleObjectType(hTempHandle, pdwObjectType) != 0)
{
// error
CloseHandle(hTempHandle);
```

```
return 1;
}

// close temporary handle
CloseHandle(hTempHandle);

return 0;
}

DWORD EnumNamedHandles(DWORD dwObjectType, DWORD (*pCallback)(DWORD dwOwnerPID,
char *pName, BYTE *pParam), BYTE *pCallbackParam)
{
DWORD dwOwnerPID = 0;
HANDLE hRemoteHandle = NULL;
HANDLE hRemoteProcess = NULL;
HANDLE hClonedHandle = NULL;
DWORD dwObjectTypeIndex = 0;
BYTE bObjectNameBuffer[4096];
DWORD dwObjectNameLength = 0;
DWORD dwStatus = 0;
UNICODE_STRING *pObjectNameUnicodeString = NULL;
char *pNamedObjectLabel = NULL;
char szFullObjectPath[1024];

// get system handle list
if(GetSystemHandleList() != 0)
{
return 1;
}

// check for debug handles
for(DWORD i = 0; i < (DWORD)pGlobal_SystemHandleInfo->NumberOfHandles; i++)
{
// get values
dwOwnerPID = (DWORD)pGlobal_SystemHandleInfo->HandleList[i].UniqueProcessId;
hRemoteHandle = (HANDLE)pGlobal_SystemHandleInfo->HandleList[i].HandleValue;
dwObjectTypeIndex = pGlobal_SystemHandleInfo->HandleList[i].ObjectTypeIndex;

// ignore handles within the current process
if(dwOwnerPID == GetCurrentProcessId())
{
continue;
}

// check object type
if(dwObjectTypeIndex != dwObjectType)
{
continue;
}

// open remote process
hRemoteProcess = OpenProcess(PROCESS_DUP_HANDLE, 0, dwOwnerPID);
if(hRemoteProcess == NULL)
{
```

```
// failed to open process handle
continue;
}

// clone remote handle to local process
if(DuplicateHandle(hRemoteProcess, hRemoteHandle, GetCurrentProcess(), &hClonedHandle, 0, 0,
DUPLICATE_SAME_ACCESS) == 0)
{
// failed to duplicate handle
CloseHandle(hRemoteProcess);
continue;
}

// close remote process handle
CloseHandle(hRemoteProcess);

// get object name
memset(bObjectNameBuffer, 0, sizeof(bObjectNameBuffer));
dwStatus = pNtQueryObject(hClonedHandle, ObjectNameInformation, bObjectNameBuffer,
sizeof(bObjectNameBuffer), &dwObjectNameLength);
if(dwStatus != 0)
{
// failed to get object name
CloseHandle(hClonedHandle);
continue;
}

// close handle
CloseHandle(hClonedHandle);

// check if a valid name was returned
pObjectNameUnicodeString = (UNICODE_STRING*)bObjectNameBuffer;
if(pObjectNameUnicodeString->Length == 0)
{
// blank name
continue;
}

// convert path to ansi string
memset(szFullObjectPath, 0, sizeof(szFullObjectPath));
wcstombs(szFullObjectPath, pObjectNameUnicodeString->Buffer, sizeof(szFullObjectPath) - 1);

// check if the object name exists under BaseNamedObjects (ignore app containers)
pNamedObjectLabel = strstr(szFullObjectPath, "\\BaseNamedObjects\\");
if(pNamedObjectLabel == NULL)
{
// ignore
continue;
}

// find start of object label
pNamedObjectLabel += strlen("\\BaseNamedObjects\\");
```

```
// execute callback
if(pCallback(dwOwnerPID, pNamedObjectLabel, pCallbackParam) != 0)
{
// callback error
return 1;
}
}

return 0;
}

DWORD AddEntryToList(char *pName, char ***pEntryList, DWORD *pdwEntryCount)
{
DWORD dwFound = 0;
char *pStoredName = NULL;
DWORD dwAllocSize = 0;
DWORD dwTempEntryCount = 0;
char **pTempEntryList = NULL;

// store initial values
dwTempEntryCount = *pdwEntryCount;
pTempEntryList = *pEntryList;

// check if this name already exists in the list
for(DWORD i = 0; i < dwTempEntryCount; i++)
{
// check current entry
if(stricmp(pTempEntryList[i], pName) == 0)
{
// found matching entry
dwFound = 1;
break;
}
}

// add the name to the list if it doesn't already exist
if(dwFound == 0)
{
// allocate memory for name string
dwAllocSize = (DWORD)strlen(pName) + 1;
pStoredName = (char*)malloc(dwAllocSize);
if(pStoredName == NULL)
{
return 1;
}

// copy name string
memset(pStoredName, 0, dwAllocSize);
memcpy(pStoredName, pName, dwAllocSize - 1);

// re-allocate the list, append a new entry
pTempEntryList = (char**)realloc(pTempEntryList, sizeof(char*) * (dwTempEntryCount + 1));
if(pTempEntryList == NULL)
{
```

```
free(pStoredName);
return 1;
}

// store name ptr
pTempEntryList[dwTempEntryCount] = pStoredName;

// increase entry count
dwTempEntryCount++;
}

// update values
*pdwEntryCount = dwTempEntryCount;
*pEntryList = pTempEntryList;

return 0;
}

DWORD WriteListToFile(char *pFileName, char **pEntryList, DWORD dwEntryCount)
{
FILE *pFile = NULL;

// create file
pFile = fopen(pFileName, "wb");
if(pFile == NULL)
{
return 1;
}

// write all list entries to file
for(DWORD i = 0; i < dwEntryCount; i++)
{
fprintf(pFile, "%s\r\n", pEntryList[i]);
}

// close file handle
fclose(pFile);

return 0;
}

DWORD ImportListFromFile(char *pFileName, char ***pEntryList, DWORD *pdwEntryCount)
{
FILE *pFile = NULL;
char *pEndOfLine = NULL;
char szLine[1024];

// create file
pFile = fopen(pFileName, "r");
if(pFile == NULL)
{
printf("Failed to open file: %s. Use \"-generate\" first.\n", pFileName);
return 1;
}
```

```
// read all entries from file
for(;;)
{
// read next line
memset(szLine, 0, sizeof(szLine));
if(fgets(szLine, sizeof(szLine) - 1, pFile) == NULL)
{
break;
}

// remove carriage return
pEndOfLine = strstr(szLine, "\r");
if(pEndOfLine != NULL)
{
*pEndOfLine = '\0';
}

// remove carriage return
pEndOfLine = strstr(szLine, "\n");
if(pEndOfLine != NULL)
{
*pEndOfLine = '\0';
}

// check if the line is blank
if(strlen(szLine) == 0)
{
continue;
}

// add entry to list
if(AddEntryToList(szLine, pEntryList, pdwEntryCount) != 0)
{
fclose(pFile);
return 1;
}
}

// close file handle
fclose(pFile);

return 0;
}

DWORD EnumEventsCallback_Generate(DWORD dwOwnerPID, char *pName, BYTE *pParam)
{
// add event object to list
if(AddEntryToList(pName, &pGlobal_EventObjectNameList, &dwGlobal_EventObjectCount) != 0)
{
return 1;
}

return 0;
}
```

```
DWORD EnumSectionsCallback_Generate(DWORD dwOwnerPID, char *pName, BYTE *pParam)
{
// add section object to list
if(AddEntryToList(pName, &pGlobal_SectionObjectNameList, &dwGlobal_SectionObjectCount) != 0)
{
return 1;
}

return 0;
}

DWORD GetProcessNameFromPID(DWORD dwPID, char *pProcessName, DWORD dwMaxLength)
{
SYSTEM_PROCESS_INFORMATION *pCurrProcessInfo = NULL;
SYSTEM_PROCESS_INFORMATION *pNextProcessInfo = NULL;
SYSTEM_PROCESS_INFORMATION *pTargetProcessInfo = NULL;

// get system process list
if(GetSystemProcessInformation() != 0)
{
return 1;
}

// find the target process in the list
pCurrProcessInfo = pGlobal_SystemProcessInfo;
for(;;)
{
// check if this is the target PID
if((DWORD)pCurrProcessInfo->UniqueProcessId == dwPID)
{
// found target process
pTargetProcessInfo = pCurrProcessInfo;
break;
}

// check if this is the end of the list
if(pCurrProcessInfo->NextEntryOffset == 0)
{
// end of list
break;
}
else
{
// get next process ptr
pNextProcessInfo = (SYSTEM_PROCESS_INFORMATION*)((BYTE*)pCurrProcessInfo +
pCurrProcessInfo->NextEntryOffset);
}

// go to next process
pCurrProcessInfo = pNextProcessInfo;
}
```

```
// ensure the target process was found in the list
if(pTargetProcessInfo == NULL)
{
return 1;
}

// store process name
wcstombs(pProcessName, pTargetProcessInfo->ImageName.Buffer, dwMaxLength);

return 0;
}

DWORD EnumEventsCallback_Info(DWORD dwOwnerPID, char *pName, BYTE *pParam)
{
char szProcessName[512];

// get process name from PID
memset(szProcessName, 0, sizeof(szProcessName));
if(GetProcessNameFromPID(dwOwnerPID, szProcessName, sizeof(szProcessName) - 1) != 0)
{
// error
strncpy(szProcessName, "<unknown>", sizeof(szProcessName) - 1);
}

// print current entry details
printf("Event Object: %s (PID: %u) ===> %s\n", szProcessName, dwOwnerPID, pName);

return 0;
}

DWORD EnumSectionsCallback_Info(DWORD dwOwnerPID, char *pName, BYTE *pParam)
{
char szProcessName[512];

// get process name from PID
memset(szProcessName, 0, sizeof(szProcessName));
if(GetProcessNameFromPID(dwOwnerPID, szProcessName, sizeof(szProcessName) - 1) != 0)
{
// error
strncpy(szProcessName, "<unknown>", sizeof(szProcessName) - 1);
}

// print current entry details
printf("Section Object: %s (PID: %u) ===> %s\n", szProcessName, dwOwnerPID, pName);

return 0;
}

DWORD EnableDebugPrivilege()
{
HANDLE hToken = NULL;
LUID DebugLuid;
TOKEN_PRIVILEGES NewTokenPrivileges;
```

```
// open process token
if(OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, &hToken) == 0)
{
return 1;
}

// find SE_DEBUG_NAME luid
memset((void*)&DebugLuid, 0, sizeof(DebugLuid));
if(LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &DebugLuid) == 0)
{
return 1;
}

// initialise NewTokenPrivileges
memset((void*)&NewTokenPrivileges, 0, sizeof(NewTokenPrivileges));
NewTokenPrivileges.PrivilegeCount = 1;
NewTokenPrivileges.Privileges[0].Luid = DebugLuid;
NewTokenPrivileges.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

// enable debug privilege
if(AdjustTokenPrivileges(hToken, 0, &NewTokenPrivileges, sizeof(NewTokenPrivileges), NULL, NULL) ==
0)
{
return 1;
}

// confirm success - AdjustTokenPrivileges can fail even when returning a successful return value
if(GetLastError() == ERROR_NOT_ALL_ASSIGNED)
{
return 1;
}

return 0;
}

DWORD TriggerEvent(char *pEventName)
{
char szGlobalPath[1024];
HANDLE hEvent = NULL;

// open existing event - try global namespace first
memset(szGlobalPath, 0, sizeof(szGlobalPath));
_snprintf(szGlobalPath, sizeof(szGlobalPath) - 1, "Global\\%s", pEventName);
hEvent = OpenEvent(EVENT_ALL_ACCESS, 0, szGlobalPath);
if(hEvent == NULL)
{
// failed - try local namespace
hEvent = OpenEvent(EVENT_ALL_ACCESS, 0, pEventName);
if(hEvent == NULL)
{
return NULL;
}
}
```

```
// trigger event
SetEvent(hEvent);

return 0;
}

BYTE *MapSectionToMemory(char *pSectionName, DWORD *pdwSize)
{
BYTE *pMappedData = NULL;
char szGlobalPath[1024];
HANDLE hMapping = NULL;
MEMORY_BASIC_INFORMATION MemoryBasicInformation;

// open existing section - try global namespace first
memset(szGlobalPath, 0, sizeof(szGlobalPath));
_snprintf(szGlobalPath, sizeof(szGlobalPath) - 1, "Global\\%s", pSectionName);
hMapping = OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE, 0, szGlobalPath);
if(hMapping == NULL)
{
// failed - try local namespace
hMapping = OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE, 0, pSectionName);
if(hMapping == NULL)
{
return NULL;
}
}

// map section to memory
pMappedData = (BYTE*)MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);
if(pMappedData == NULL)
{
CloseHandle(hMapping);
return NULL;
}

// close section handle (the memory mapped pointer will remain valid until UnmapViewOfFile is called)
CloseHandle(hMapping);

// get memory region size
memset((void*)&MemoryBasicInformation, 0, sizeof(MemoryBasicInformation));
if(VirtualQuery(pMappedData, &MemoryBasicInformation, sizeof(MemoryBasicInformation)) == 0)
{
UnmapViewOfFile(pMappedData);
return NULL;
}

// store size
*pdwSize = MemoryBasicInformation.RegionSize;

return pMappedData;
}

DWORD DumpSection(char *pSectionName)
{
BYTE *pMappedData = NULL;
```

```
DWORD dwSectionSize = 0;
char szDumpFileName[1024];
DWORD dwDumpFileNameLength = 0;
FILE *pFile = NULL;

printf("\nDumping section: %s...\n", pSectionName);

// open section and map into memory
pMappedData = MapSectionToMemory(pSectionName, &dwSectionSize);
if(pMappedData == NULL)
{
return 1;
}

// generate output filename
memset(szDumpFileName, 0, sizeof(szDumpFileName));
_snprintf(szDumpFileName, sizeof(szDumpFileName) - 1, "dump_%s.bin", pSectionName);

// filter characters
dwDumpFileNameLength = strlen(szDumpFileName);
for(DWORD i = 0; i < dwDumpFileNameLength; i++)
{
if(szDumpFileName[i] >= 'a' && szDumpFileName[i] <= 'z')
{
// a-z
continue;
}
else if(szDumpFileName[i] >= 'A' && szDumpFileName[i] <= 'Z')
{
// A-Z
continue;
}
else if(szDumpFileName[i] >= '0' && szDumpFileName[i] <= '9')
{
// 0-9
continue;
}
else if(szDumpFileName[i] == '_' || szDumpFileName[i] == '-' || szDumpFileName[i] == '.')
{
// special characters
continue;
}

// change character to '_' in filename
szDumpFileName[i] = '_';
}

// create output file
pFile = fopen(szDumpFileName, "wb");
if(pFile == NULL)
{
UnmapViewOfFile(pMappedData);
return 1;
}
```

```
// write section data to file
fwrite(pMappedData, dwSectionSize, 1, pFile);

// close file
fclose(pFile);

// unmap memory
UnmapViewOfFile(pMappedData);

printf("Dumped successfully, written %u bytes\n", dwSectionSize);

return 0;
}

DWORD OverwriteSection(char *pSectionName, DWORD dwRand, DWORD dwPrefix)
{
BYTE *pMappedData = NULL;
DWORD dwSectionSize = 0;
DWORD dwSectionNameLength = 0;

printf("\nOverwriting section: %s...\n", pSectionName);

// open section and map into memory
pMappedData = MapSectionToMemory(pSectionName, &dwSectionSize);
if(pMappedData == NULL)
{
return 1;
}

// calculate section name length
dwSectionNameLength = strlen(pSectionName);

// overwrite all bytes in shared memory block
for(DWORD i = 0; i < dwSectionSize; i++)
{
// check if the "-prefix" flag was specified
if(dwPrefix != 0)
{
// write the section name to the beginning of the region
if(i < dwSectionNameLength)
{
*(BYTE*)((BYTE*)pMappedData + i) = *(BYTE*)((BYTE*)pSectionName + i);
continue;
}
}

// check if the "-rand" flag was specified
if(dwRand == 0)
{
// write an 'a' character
*(BYTE*)((BYTE*)pMappedData + i) = 'a';
}
else
{
// write a random character
```

```
*(BYTE*)((BYTE*)pMappedData + i) = (BYTE)rand();
}
}

// unmap section from memory
UnmapViewOfFile(pMappedData);

return 0;
}

DWORD InitialiseSharedMemUtils(DWORD dwMode)
{
// get pNtQuerySystemInformation function ptr
pNtQuerySystemInformation = (unsigned long (__stdcall *)(unsigned long,void *,unsigned long,unsigned
long *))GetProcAddress(GetModuleHandle("ntdll.dll"), "NtQuerySystemInformation");
if(pNtQuerySystemInformation == NULL)
{
return 1;
}

// get pNtQueryObject function ptr
pNtQueryObject = (unsigned long (__stdcall *)(void *,unsigned long,void *,unsigned long,unsigned long
*))GetProcAddress(GetModuleHandle("ntdll.dll"), "NtQueryObject");
if(pNtQueryObject == NULL)
{
return 1;
}

// find the event handle object type
if(GetEventObjectType(&dwGlobal_EventObjectType) != 0)
{
return 1;
}

// find the section handle object type
if(GetSectionObjectType(&dwGlobal_SectionObjectType) != 0)
{
return 1;
}

// enabling debug privileges will improve results when enumerating system handles
if(dwMode == MODE_GENERATE || dwMode == MODE_INFO)
{
// try to enable debug privileges
if(EnableDebugPrivilege() != 0)
{
printf("Warning - Failed to enable debug privileges, run this command elevated for better results.\n\n");
}
}

return 0;
}
```

```
int main(int argc, char *argv[])
{
DWORD dwMode = 0;
DWORD dwOverwriteFlags_Rand = 0;
DWORD dwOverwriteFlags_Prefix = 0;
DWORD dwOverwriteFlags_Events = 0;

printf("SharedMemUtils - x86matthew\n");
printf("www.x86matthew.com\n");
printf("\n");

if(argc < 2)
{
// print usage
printf("Modes:\n");
printf("\n");
printf("-generate ===> Generate lists of all shared memory and event objects\n");
printf("\n");
printf("-dump ===> Dump all shared memory objects to disk\n");
printf("\n");
printf("-overwrite ===> Overwrite data in shared memory objects\n");
printf(" Optional flags:\n");
printf(" -rand ===> Overwrite with random data (default uses 'a' characters)\n");
printf(" -prefix ===> Prefix the data with the section name\n");
printf(" -events ===> Trigger all event objects after writing\n");
printf("\n");
printf("-info ===> Print information about shared memory and event objects\n");
printf("\n");
printf("Recommended usage:\n");
printf("\n");
printf("Usage: %s <mode> [optional_flags]\n", argv[0]);

return 1;
}

// set random seed
srand(GetTickCount());

// read parameters
for(DWORD i = 0; i < (DWORD)argc; i++)
{
if(i == 0)
{
// ignore argv[0] (path)
continue;
}
else if(i == 1)
{
// get mode
if(stricmp(argv[i], "-generate") == 0)
{
dwMode = MODE_GENERATE;
}
```

```
else if(stricmp(argv[i], "-dump") == 0)
{
dwMode = MODE_DUMP;
}
else if(stricmp(argv[i], "-overwrite") == 0)
{
dwMode = MODE_OVERWRITE;
}
else if(stricmp(argv[i], "-info") == 0)
{
dwMode = MODE_INFO;
}
else
{
printf("Error - invalid mode\n");
return 1;
}
}
else
{
// optional flags
if(stricmp(argv[i], "-rand") == 0)
{
dwOverwriteFlags_Rand = 1;
}
else if(stricmp(argv[i], "-prefix") == 0)
{
dwOverwriteFlags_Prefix = 1;
}
else if(stricmp(argv[i], "-events") == 0)
{
dwOverwriteFlags_Events = 1;
}
else
{
printf("Error - unknown parameter\n");
return 1;
}
}
}

// initialise
if(InitialiseSharedMemUtils(dwMode) != 0)
{
printf("Error - failed to initialise\n");
return 1;
}

if(dwMode == MODE_GENERATE)
{
printf("Generating list of shared memory objects...\n");
```

```
// generate list of named section objects
if(EnumNamedHandles(dwGlobal_SectionObjectType, EnumSectionsCallback_Generate, NULL) != 0)
{
return 1;
}

printf("Writing to %s...\n", SHARED_MEMORY_LIST_FILENAME);

// write list to file
if(WriteListToFile(SHARED_MEMORY_LIST_FILENAME, pGlobal_SectionObjectNameList,
dwGlobal_SectionObjectCount) != 0)
{
return 1;
}

printf("\nGenerating list of event objects...\n");

// generate list of named event objects
if(EnumNamedHandles(dwGlobal_EventObjectType, EnumEventsCallback_Generate, NULL) != 0)
{
return 1;
}

printf("Writing to %s...\n", EVENT_LIST_FILENAME);

// write list to file
if(WriteListToFile(EVENT_LIST_FILENAME, pGlobal_EventObjectNameList,
dwGlobal_EventObjectCount) != 0)
{
return 1;
}
}
else if(dwMode == MODE_DUMP)
{
printf("Dumping all shared memory objects...\n");

// import section list from file
if(ImportListFromFile(SHARED_MEMORY_LIST_FILENAME, &pGlobal_SectionObjectNameList,
&dwGlobal_SectionObjectCount) != 0)
{
return 1;
}

// dump all sections
for(DWORD i = 0; i < dwGlobal_SectionObjectCount; i++)
{
// dump current section data
if(DumpSection(pGlobal_SectionObjectNameList[i]) != 0)
{
printf("Failed to dump section\n");
}
}
}
```

```
else if(dwMode == MODE_OVERWRITE)
{
printf("Overwriting all shared memory objects...\n");

// import section list from file
if(ImportListFromFile(SHARED_MEMORY_LIST_FILENAME, &pGlobal_SectionObjectNameList,
&dwGlobal_SectionObjectCount) != 0)
{
return 1;
}

// overwrite all sections
for(DWORD i = 0; i < dwGlobal_SectionObjectCount; i++)
{
// overwrite current section data
if(OverwriteSection(pGlobal_SectionObjectNameList[i], dwOverwriteFlags_Rand,
dwOverwriteFlags_Prefix) != 0)
{
printf("Failed to overwrite section\n");
}
}

// check if the "-events" flag was specified
if(dwOverwriteFlags_Events != 0)
{
printf("Triggering all event objects...\n");

// import event list from file
if(ImportListFromFile(EVENT_LIST_FILENAME, &pGlobal_EventObjectNameList,
&dwGlobal_EventObjectCount) != 0)
{
return 1;
}

// trigger all events
for(DWORD i = 0; i < dwGlobal_EventObjectCount; i++)
{
// trigger current event object
if(TriggerEvent(pGlobal_EventObjectNameList[i]) != 0)
{
printf("Failed to trigger event\n");
}
}
}
}
else if(dwMode == MODE_INFO)
{
// display all named section objects
if(EnumNamedHandles(dwGlobal_SectionObjectType, EnumSectionsCallback_Info, NULL) != 0)
{
return 1;
}
```

```
// display all named event objects
if(EnumNamedHandles(dwGlobal_SectionObjectType, EnumEventsCallback_Info, NULL) != 0)
{
return 1;
}
}
else
{
// error
return 1;
}

printf("\nFinished\n\n");

return 0;
}
```