# Reading and writing remote process data without using ReadProcessMemory / WriteProcessMemory

web.archive.org/web/20220405165723/https://www.x86matthew.com/view_post

Reading and writing remote process data without using ReadProcessMemory / WriteProcessMemory

I have recently been working on some new methods to read and write remote process data without relying on ReadProcessMemory / WriteProcessMemory (or their ntdll equivalents - NtReadVirtualMemory / NtWriteVirtualMemory).

I will detail one method that I have successfully developed in this post. This method is based around the NtCreateThreadEx function.

My plan was to find some existing function exports in ntdll/kernel32 which could be used as a thread entry-point to manipulate data.

Reading Data

In order to read data, we need to find an exported API function that returns the value of a pointer passed in as a parameter.

I found RtlFirstEntrySList in ntdll.dll which looks like this:

```
77562758 > 8B4424 04 MOV EAX,DWORD PTR SS:[ESP+4]
7756275C 8B00 MOV EAX,DWORD PTR DS:[EAX]
7756275E C2 0400 RETN 4
```

In C, this function would look like this (due to the position of the Next entry in the SLIST_HEADER structure):

```
DWORD __stdcall RtlFirstEntrySList(DWORD *pValue)
{
return *pValue;
}
```

When we compare this with the format of a standard thread entry-point, we can see that this function will be perfect for retrieving data:

```
DWORD __stdcall ThreadProc(LPVOID lpParameter);
```

In summary, we can use the information above to read remote process data using the following steps:

1. Call NtCreateThreadEx to create a thread in the remote process using RtlFirstEntrySList as the entry-point and the parameter as the address that we are reading from.
2. Call WaitForSingleObject on the new thread to wait until the remote RtlFirstEntrySList function call returns.
3. Call GetExitCodeThread to retrieve the return value of RtlFirstEntrySList. This will be the value of the address specified by the thread parameter.

Full reading code below:

```
DWORD ReadMemory_GetByte(HANDLE hProcess, BYTE *pPtr, BYTE *pValue)
{
DWORD (WINAPI *NtCreateThreadEx)(HANDLE *phThreadHandle, DWORD
DesiredAccess, PVOID ObjectAttributes, HANDLE hProcessHandle, PVOID
StartRoutine, PVOID Argument, ULONG CreateFlags, DWORD *pZeroBits, SIZE_T
StackSize, SIZE_T MaximumStackSize, PVOID AttributeList);
HANDLE hThread = NULL;
LPTHREAD_START_ROUTINE pThreadRoutine = NULL;
DWORD dwExitCode = 0;

// find NtCreateThreadEx ptr in ntdll
NtCreateThreadEx = (unsigned long (__stdcall *)(void ** ,unsigned long,void *,void *,void
*,void *,unsigned long,unsigned long *,unsigned long,unsigned long,void
*))GetProcAddress(GetModuleHandle("ntdll.dll"), "NtCreateThreadEx");
if(NtCreateThreadEx == NULL)
{
return 1;
}

// find RtlFirstEntrySList ptr in ntdll
pThreadRoutine =
(LPTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle("ntdll.dll"),
"RtlFirstEntrySList");
if(pThreadRoutine == NULL)
{
return 1;
}

// create remote thread
if(NtCreateThreadEx(&hThread;, 0x001FFFFF, NULL, hProcess, pThreadRoutine,
```

```
(LPVOID)pPtr, 0, NULL, 0, 0, NULL) != 0)
{
return 1;
}

// wait for RtlFirstEntrySList to return
if(WaitForSingleObject(hThread, INFINITE) != WAIT_OBJECT_0)
{
// error
CloseHandle(hThread);

return 1;
}

// get exit code (this contains the RtlFirstEntrySList return value)
if(GetExitCodeThread(hThread, &dwExitCode;) == 0)
{
// error
CloseHandle(hThread);

return 1;
}

// close thread handle
CloseHandle(hThread);

// store output value
*pValue = (BYTE)dwExitCode;

return 0;
}

DWORD ReadMemory(HANDLE hProcess, BYTE *pAddress, BYTE *pData, DWORD
dwDataLength)
{
for(DWORD i = 0; i < dwDataLength; i++)
{
// get current byte
if(ReadMemory_GetByte(hProcess, (BYTE*)(pAddress + i), (BYTE*)(pData + i)) != 0)
{
return 1;
}
}
```

```
return 0;
}
```

Writing Data

Using this concept to write to a remote process is slightly more complex - I will be using the InterlockedIncrement / InterlockedDecrement functions in kernel32.dll.

These functions increase/decrease the specified value by 1, and have the following format:

```
LONG __stdcall InterlockedIncrement(LONG *Addend);
LONG __stdcall InterlockedDecrement(LONG *Addend);
```

This works as follows:

1. Read the original byte value using the ReadMemory_GetByte function above.
2. Check if the current byte value is greater to or less than the target value.
3. Call NtCreateThreadEx to create a thread in the remote process using InterlockedIncrement / InterlockedDecrement as the entry-point, depending on whether the value needs to be increased or decreased.
4. Repeat step #3 until the target byte is correct.

Full writing code below:

```
DWORD WriteMemory_IncreaseDecreaseValue(HANDLE hProcess, BYTE *pPtr,
DWORD dwDecrease, BYTE *pValue)
{
DWORD (WINAPI *NtCreateThreadEx)(HANDLE *phThreadHandle, DWORD
DesiredAccess, PVOID ObjectAttributes, HANDLE hProcessHandle, PVOID
StartRoutine, PVOID Argument, ULONG CreateFlags, DWORD *pZeroBits, SIZE_T
StackSize, SIZE_T MaximumStackSize, PVOID AttributeList);
HANDLE hThread = NULL;
LPTHREAD_START_ROUTINE pThreadRoutine = NULL;
DWORD dwExitCode = 0;

// find NtCreateThreadEx ptr in ntdll
NtCreateThreadEx = (unsigned long (__stdcall *)(void ** ,unsigned long,void *,void *,void
*,void *,unsigned long,unsigned long *,unsigned long,unsigned long,void
*))GetProcAddress(GetModuleHandle("ntdll.dll"), "NtCreateThreadEx");
if(NtCreateThreadEx == NULL)
{
```

```
return 1;
}

// check if the value should be increased or decreased
if(dwDecrease == 0)
{
// increase by 1 - use InterlockedIncrement
pThreadRoutine =
(LPTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle("kernel32.dll"),
"InterlockedIncrement");
}
else
{
// decrease by 1 - use InterlockedDecrement
pThreadRoutine =
(LPTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle("kernel32.dll"),
"InterlockedDecrement");
}

// ensure InterlockedIncrement / InterlockedDecrement ptr was found
if(pThreadRoutine == NULL)
{
return 1;
}

// create remote thread
if(NtCreateThreadEx(&hThread;, 0x001FFFFF, NULL, hProcess, pThreadRoutine,
(LPVOID)pPtr, 0, NULL, 0, 0, NULL) != 0)
{
return 1;
}

// wait for remote function to return
if(WaitForSingleObject(hThread, INFINITE) != WAIT_OBJECT_0)
{
// error
CloseHandle(hThread);

return 1;
}

// get exit code (this contains the InterlockedIncrement / InterlockedDecrement return
value)
if(GetExitCodeThread(hThread, &dwExitCode;) == 0)
{
```

```
// error
CloseHandle(hThread);

return 1;
}

// close thread handle
CloseHandle(hThread);

// store output value
*pValue = (BYTE)dwExitCode;

return 0;
}

DWORD WriteMemory_UpdateByte(HANDLE hProcess, BYTE *pPtr, BYTE bValue)
{
BYTE bCurrValue = 0;

// get initial value
if(ReadMemory_GetByte(hProcess, pPtr, &bCurrValue;) != 0)
{
return 1;
}

// increase/decrease the current byte until it is correct
for(;;)
{
// check if the value needs to be increased or decreased
if(bCurrValue < bValue)
{
// increase
if(WriteMemory_IncreaseDecreaseValue(hProcess, pPtr, 0, &bCurrValue;) != 0)
{
return 1;
}
}
else if(bCurrValue > bValue)
{
// decrease
if(WriteMemory_IncreaseDecreaseValue(hProcess, pPtr, 1, &bCurrValue;) != 0)
{
return 1;
}
}
```

```
else
{
// finished
break;
}
}

return 0;
}

DWORD WriteMemory(HANDLE hProcess, BYTE *pAddress, BYTE *pData, DWORD
dwDataLength)
{
for(DWORD i = 0; i < dwDataLength; i++)
{
// write current byte
if(WriteMemory_UpdateByte(hProcess, (BYTE*)(pAddress + i), *(BYTE*)(pData + i)) != 0)
{
return 1;
}
}

return 0;
}
```

Example of use:

```
int main()
{
char szTestString[128];
char szReadValue[128];

// set initial value
memset(szTestString, 0, sizeof(szTestString));
strncpy(szTestString, "Original text string", sizeof(szTestString) - 1);

// read value from local process
if(ReadMemory(GetCurrentProcess(), (BYTE*)szTestString, (BYTE*)szReadValue,
sizeof(szReadValue)) != 0)
{
return 1;
}
printf("#1 - '%s'\n", szReadValue);
```

```
// overwrite the word 'Original' with 'Modified'
if(WriteMemory(GetCurrentProcess(), (BYTE*)szTestString, (BYTE*)"Modified",
strlen("Modified")) != 0)
{
return 1;
}
printf("Updated string\n");

// read updated value from local process
if(ReadMemory(GetCurrentProcess(), (BYTE*)szTestString, (BYTE*)szReadValue,
sizeof(szReadValue)) != 0)
{
return 1;
}
printf("#2 - '%s'\n", szReadValue);

return 0;
}
```

This outputs the following:

```
#1 - 'Original text string'
Updated string
#2 - 'Modified text string'
```

Note: CreateRemoteThread could be used in place of NtCreateThreadEx, but this contains a lot of user-mode overhead (allocating a new stack in the remote process, etc). NtCreateThreadEx is a direct syscall which lets the kernel handle everything else.