x86matthew - WriteProcessMemoryAPC - Write memory to a remote process using APC calls

x86matthew.com/view_post

WriteProcessMemoryAPC - Write memory to a remote process using APC calls

09/09/2022

This small project demonstrates another technique that I discovered to write memory to a remote process - this can be used to replace the WriteProcessMemory API.

This method uses a sequence of APC calls scheduled against a newly-created thread, and takes advantage of various pre-existing functions within Windows.

The standard Windows API to schedule APC calls (QueueUserAPC) only allows us to call functions with a single parameter. Without injecting additional code, it is difficult to make use of this to write specific data to a specific address. Internally, QueueUserAPC calls an undocumented ntdll.dll function called NtQueueApcThread - this function allows us to specify 3 parameters to the callback function. This is because QueueUserAPC initially routes all APC calls to an internal redirector function called RtlDispatchAPC before being calling the target function. Now that we are able to specify a callback function with 3 custom parameters, this opens up a lot more opportunities to overwrite memory.

My initial idea was to use a function similar to memset. For example, scheduling a call to memset(0x11223344, 'a', 1) would write a single 'a' (0x61) character to 0x11223344. Unfortunately, the memset function uses the cdecl calling convention whereas the NtQueueApcThread function expects a callback with a stdcall calling convention. This means memset would cause stack corruption on a 32-bit process. I was aware that ntdll.dll exports some Rtl* functions that match the common C-runtime functions - I had a

look and found a function called RtlFillMemory. This function performs the same functionality as memset, but crucially uses the stdcall calling convention, making it suitable to use with NtQueueApcThread.

This proof-of-concept takes the following steps:

1. Locate the addresses of NtCreateThreadEx, NtQueueApcThread, RtlFillMemory within ntdll.dll.
2. Call NtCreateThreadEx to create a suspended thread with an entry-point of ExitThread within the target process.
3. For each byte to write, call NtQueueApcThread to schedule an APC call to RtlFillMemory within the newly created thread.
4. Call ResumeThread to begin execution of the target thread. This will execute all of the scheduled RtlFillMemory calls before calling ExitThread(0).

Full code below:

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

#define NT_CREATE_THREAD_EX_SUSPENDED 1
#define NT_CREATE_THREAD_EX_ALL_ACCESS 0x001FFFFF

DWORD WriteProcessMemoryAPC(HANDLE hProcess, BYTE *pAddress, BYTE *pData,
DWORD dwLength)
{
HANDLE hThread = NULL;
DWORD (WINAPI *pNtQueueApcThread)(HANDLE ThreadHandle, PVOID pApcRoutine,
PVOID pParam1, PVOID pParam2, PVOID pParam3) = NULL;
DWORD (WINAPI *pNtCreateThreadEx)(HANDLE *phThreadHandle, DWORD
DesiredAccess, PVOID ObjectAttributes, HANDLE hProcessHandle, PVOID
StartRoutine, PVOID Argument, ULONG CreateFlags, DWORD *pZeroBits, SIZE_T
StackSize, SIZE_T MaximumStackSize, PVOID AttributeList) = NULL;
void *pRtlFillMemory = NULL;

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

// find NtCreateThreadEx function
pNtCreateThreadEx = (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(pNtCreateThreadEx == NULL)
{
return 1;
}

// find RtlFillMemory function
pRtlFillMemory = (void*)GetProcAddress(GetModuleHandle("kernel32.dll"),
"RtlFillMemory");
if(pRtlFillMemory == NULL)
{
return 1;
}

// create suspended thread (ExitThread)
if(pNtCreateThreadEx(&hThread, NT_CREATE_THREAD_EX_ALL_ACCESS, NULL,
hProcess, (LPVOID)ExitThread, (LPVOID)0, NT_CREATE_THREAD_EX_SUSPENDED,
NULL, 0, 0, NULL) != 0)
{
return 1;
}

// write memory
for(DWORD i = 0; i < dwLength; i++)
{
// schedule a call to RtlFillMemory to update the current byte
if(pNtQueueApcThread(hThread, pRtlFillMemory, (void*)((BYTE*)pAddress + i), (void*)1,
(void*)*(BYTE*)(pData + i)) != 0)
{
// error
TerminateThread(hThread, 0);
CloseHandle(hThread);
return 1;
}
}

// resume thread to execute queued APC calls
ResumeThread(hThread);

// wait for thread to exit
WaitForSingleObject(hThread, INFINITE);

// close thread handle
CloseHandle(hThread);

return 0;
}
```

```
int main()
{
char szTestString[1024];
char szOverwriteValue[64];

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

// set original string value
memset(szTestString, 0, sizeof(szTestString));
_snprintf(szTestString, sizeof(szTestString) - 1, "Original Value");

// print the original string value
printf("Value: '%s'\n", szTestString);

printf("Overwriting data...\n");

// overwrite the string value using APC - use the current process for demonstration
purposes
memset(szOverwriteValue, 0, sizeof(szOverwriteValue));
_snprintf(szOverwriteValue, sizeof(szOverwriteValue) - 1, "** Overwritten Value **");
if(WriteProcessMemoryAPC(GetCurrentProcess(), (BYTE*)szTestString,
(BYTE*)szOverwriteValue, strlen(szOverwriteValue) + 1) != 0)
{
return 1;
}

// print the new string value
printf("Value: '%s'\n", szTestString);

return 0;
}
```