

Forum:[Hack Forums](#)

Thread: [\[Help\] Fixing the Access Violation Exception when Thread Hijacking](#)

LunarManure - November 1, 2020, 8:19 am

I'm fairly new to thread injection and am learning the technique through a book called "Game Hacking: Developing Autonomous Bot for Online Games." Chapter 7 highlights 3 injection techniques (thread injection, thread hijacking, and DLL injection).

Currently, I'm having trouble getting my thread hijacking code to work, because I keep running into an Access Violation Error when running my code. I've tried running as admin and the error still persists. Could anyone sift through my code and give me a reason why this error keeps popping up? I'm using Visual Studios 2019.

Code Exception thrown at 0x00000024 in GameHacking.exe: 0xC0000005: Access violation executing location 0x00000024.

```
Code #include <windows.h>
#include <tlhelp32.h>
#include <string>
```

```
using namespace std;
```

```
// function being referenced by injector
DWORD printStringManyTimes(int times, const char* string)
{
    for (int i = 0; i < times; i++)
        printf(string);
    return 0;
}
```

```
// THIS FUNCTION WORKS JUST FINE
```

```
void injectCodeUsingThreadInjection(HANDLE process, LPVOID func, int times, const char* string)
{
    BYTE codeCave[20] = {
        0xFF, 0x74, 0x24, 0x04, // PUSH DWORD PTR[ESP+0x4]
        0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
        0xB8, 0x00, 0x00, 0x00, 0x00, // MOV EAX, 0x0
        0xFF, 0xD0, // CALL EAX
        0x83, 0xC4, 0x08, // ADD ESP, 0x08
        0xC3 // RETN
    };
```

```
// copy values to the shellcode
memcpy(&codeCave[5], &times, 4);
memcpy(&codeCave[10], &func, 4);
```

```
// allocate memory for the code cave
int stringlen = strlen(string) + 1;
int fullen = stringlen + sizeof(codeCave);
```

```

LPVOID remoteString = VirtualAllocEx(process, NULL, fulllen, MEM_COMMIT,
PAGE_EXECUTE);
LPVOID remoteCave = (LPVOID)((DWORD)remoteString + stringlen);

if (remoteString) {
    // write the code cave
    WriteProcessMemory(process, remoteString, string, stringlen, NULL);
    WriteProcessMemory(process, remoteCave, codeCave, sizeof(codeCave), NULL);

    // run the thread
    HANDLE thread = CreateRemoteThread(process, NULL, NULL,
        (LPTHREAD_START_ROUTINE)remoteCave,
        remoteString, NULL, NULL);
    WaitForSingleObject(thread, INFINITE);
    VirtualFreeEx(process, remoteString, fulllen, MEM_RELEASE);
    CloseHandle(thread);
}
}

```

```

// I know this function is redundant because the PID is already
// found in the main function, but I wanted to first copy what
// the book outlines in order to test how it works
DWORD GetProcessThreadID(HANDLE Process)
{
    THREADENTRY32 entry;
    entry.dwSize = sizeof(THREADENTRY32);
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);

    if (Thread32First(snapshot, &entry) == TRUE)
    {
        DWORD PID = GetProcessId(Process);
        while (Thread32Next(snapshot, &entry) == TRUE)
        {
            if (entry.th32OwnerProcessID == PID)
            {
                CloseHandle(snapshot);
                return entry.th32ThreadID;
            }
        }
    }
    CloseHandle(snapshot);
    return NULL;
}

```

// THIS IS THE FUNCTION GIVING ME PROBLEMS

```

void injectCodeUsingThreadHijacking(HANDLE process, LPVOID func, int times, const char* string)
{
    BYTE codeCave[31] = {
        0x06, //PUSHAD
        0x9C, //PUSHFD
        0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
        0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
    }
}

```

```

    0xB8, 0x00, 0x00, 0x00, 0x00, // MOV EAX, 0x0
    0xFF, 0xD0, // CALL EAX
    0x83, 0xC4, 0x08, // ADD ESP, 0x08
    0x9D, // POPFD
    0x61, // POPAD
    0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
    0xC3 // RETN
};

// allocate memory for the code cave
int stringlen = strlen(string) + 1;
int fulllen = stringlen + sizeof(codeCave);
LPVOID remoteString = VirtualAllocEx(process, NULL, fulllen, MEM_COMMIT,
PAGE_EXECUTE);
LPVOID remoteCave = (LPVOID)((DWORD)remoteString + stringlen);

// suspend the thread and query its control context
DWORD threadID = GetProcessThreadID(process);
HANDLE thread = OpenThread(THREAD_ALL_ACCESS, false, threadID);
//(THREAD_GET_CONTEXT | THREAD_SUSPEND_RESUME | THREAD_SET_CONTEXT)
SuspendThread(thread);

CONTEXT threadContext;
threadContext.ContextFlags = CONTEXT_CONTROL;
GetThreadContext(thread, &threadContext);

// copy values to the shellcode (happens late because we need values from allocation)
memcpy(&codeCave[3], &remoteString, 4);
memcpy(&codeCave[8], &times, 4);
memcpy(&codeCave[13], &func, 4);
memcpy(&codeCave[25], &threadContext.Eip, 4);

// write the code cave
WriteProcessMemory(process, remoteString, string, stringlen, NULL);
WriteProcessMemory(process, remoteCave, codeCave, sizeof(codeCave), NULL);

//hijack the thread
threadContext.Eip = (DWORD)remoteCave;
threadContext.ContextFlags = CONTEXT_CONTROL;
SetThreadContext(thread, &threadContext);
ResumeThread(thread);

//clean
CloseHandle(thread);
}

DWORD WINAPI hijackThread(LPVOID lpParam)
{
    injectCodeUsingThreadHijacking((HANDLE)lpParam, &printStringManyTimes, 2, "hijacked\n");
    return 1;
}

```

```

int main() {
    string myTitle;

    cout << "What is the process you are looking for? ";
    cin >> myTitle;

    DWORD PID = -1;
    PROCESSENTRY32 entry;
    entry.dwSize = sizeof(PROCESSENTRY32);
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);

    if (Process32First(snapshot, &entry) == true) {
        while (Process32Next(snapshot, &entry) == true) {
            cout << entry.szExeFile << endl;
            string binPath = entry.szExeFile;
            if (binPath.find(myTitle) != string::npos) {
                cout << endl << myTitle << " has the pid " << entry.th32ProcessID << endl;
                PID = entry.th32ProcessID;
                break;
            }
        }
    }
    if (PID == -1) {
        printf("\n%s was not found in the process list.\n", myTitle);
        return -1;
    }
    CloseHandle(snapshot);

    HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID);
    if (process == INVALID_HANDLE_VALUE) {
        printf("Failed to open PID %d, error code %lu", PID, GetLastError());
    }

    // Inject code into self using thread injection
    injectCodeUsingThreadInjection(process, &printStringManyTimes, 2, "injected\n");

    // inject code into self using thread hijacking
    // we need to do it from a secondary thread or else
    // the hijacking code would hijack itself.. which
    // doesn't work
    CreateThread(NULL, 0, hijackThread, process, 0, NULL);

    while (true) {
        Sleep(100);
    }

    CloseHandle(process);
    return 0;
}

```

saltcracker - November 1, 2020, 8:28 am

Pro tip..

Never and I mean never run a hacking app with administration rights.

LunarManure - November 1, 2020, 8:36 am

(Nov 1, 2020 09:28 AM)saltycracker Wrote: [\[CITE\]](#) Pro tip..

Never and I mean never run a hacking app with administration rights.

I simply ran Visual Studios 2019 as admin, so I could rule out any reasons as to why the exception was thrown. I understand that hacker apps should work without admin rights if you want them to truly be reliable and effective. Not to mention the damage that could be done by any malicious code within the app.

I wrote this code myself with the help of the book, so I'm not worried about me hacking myself lmao...

saltycracker - November 1, 2020, 8:40 am

(Nov 1, 2020 09:36 AM)LunarManure Wrote: [\[CITE\]](#) I simply ran Visual Studios 2019 as admin, so I could rule out any reasons as to why the exception was thrown. I understand that hacker apps should work without admin rights if you want them to truly be reliable and effective. Not to mention the damage that could be done by any malicious code within the app.

I wrote this code myself with the help of the book, so I'm not worried about me hacking myself lmao...

Well from the info you provided, you didn't supply anything that indicated that it was a administration problem.

How do you know you aren't trying to write to read only memory or writing to executable memory?

Does this
Code 0x00000024 look like user memory?

LunarManure - November 1, 2020, 9:00 am

(Nov 1, 2020 09:40 AM)saltycracker Wrote: [\[CITE\]](#) Well from the info you provided, you didn't supply anything that indicated that it was a administration problem.

How do you know you aren't trying to write to read only memory or writing to executable memory?

Does this
Code 0x00000024 look like user memory?

It doesn't look like user memory... I understand that much, but I'm not sure why my code would be trying to access that memory space if the thread I created would be adjacent to the program "GameHacking.exe" I'm trying to hijack. At least, that's how I understand thread creation works.

I'm assuming the Code DWORD GetProcessThreadID(HANDLE Process) function is grabbing the correct thread ID...

My program prints out "hijacked" twice like it should, but it's only after this that the program throws the Access Violation Exception. Is it the clean up that's allowing the problem to persist? The EIP is being set back to the original position in the code cave, so I'm all turned around as to where the problem could be...

cghhhhasd - November 1, 2020, 11:08 am

A few remarks, not all are issues, however.

1) You are allocating a page just with execution protection and then you attempt to write data to it. Allocate it with `PAGE_READWRITE` then use the `VirtualProtect` API to set it to `PAGE_EXECUTE` right before you redirect your thread to it.

2) You are passing the address to the `Esp` member of `threadContext` in your last `memcpy` call instead of passing just the `Esp` value.

3) The `ContextFlags` parameter is not reset, so you don't need to set it to `CONTEXT_CONTROL` twice.

4) Make sure you call the `FlushInstructionCache` API after you write everything to the memory allocated for the shellcode. Some CPUs might cache all of the code that is in that region of memory right after you allocate it and they see it is executable and will not pay attention to whatever you write to it afterward and will try to execute whatever random data was there before.

5) Make the function you are trying to redirect your thread to an `__stdcall`, not a `__cdecl`, so you do not have to bother with cleaning up the stack.

6) Your shellcode is pretty weird and you should re-write it, you do not need all that stuff you added there.

7) You should pass the address of a value to your shellcode that gets set to 1 once it has returned from your function and wait for it to become 1 in your main function so you can do some clean-up like erasing your memory and such.

8) Some time ago I made some code that resets a specific thread's stack in case of an overflow. I have come up with two ways:

A) Using shellcode, like you are attempting to do, in order to redirect it to another function

B) Erasing the stack and placing the address of my function at the top and let the stack unwind itself to my function (The way this is written, you cannot receive any parameters through this way with my code, however)

I have uploaded it on Github as perhaps you will find it useful:

<https://github.com/ghzcrlvct2/Thread-Stack-Overflow-Recovery>

(Nov 1, 2020 09:28 AM)saltycracker Wrote: [\[CITE\]](#) Pro tip..

Never and I mean never run a hacking app with administration rights.
Nothing will happen, son.

LunarManure - November 1, 2020, 12:35 pm

(Nov 1, 2020 12:08 PM)cghhhhasd Wrote: [\[CITE\]](#) A few remarks, not all are issues, however.

1) You are allocating a page just with execution protection and then you attempt to write data to it. Allocate it with `PAGE_READWRITE` then use the `VirtualProtect` API to set it to `PAGE_EXECUTE` right before you redirect your thread to it.

2) You are passing the address to the `Esp` member of `threadContext` in your last `memcpy` call instead of passing just the `Esp` value.

3) The `ContextFlags` parameter is not reset, so you don't need to set it to `CONTEXT_CONTROL` twice.

4) Make sure you call the FlushInstructionCache API after you write everything to the memory allocated for the shellcode. Some CPUs might cache all of the code that is in that region of memory right after you allocate it and they see it is executable and will not pay attention to whatever you write to it afterward and will try to execute whatever random data was there before.

5) Make the function you are trying to redirect your thread to an `__stdcall`, not a `__cdecl`, so you do not have to bother with cleaning up the stack.

6) Your shellcode is pretty weird and you should re-write it, you do not need all that stuff you added there.

7) You should pass the address of a value to your shellcode that gets set to 1 once it has returned from your function and wait for it to become 1 in your main function so you can do some clean-up like erasing your memory and such.

8) Some time ago I made some code that resets a specific thread's stack in case of an overflow. I have come up with two ways:

A) Using shellcode, like you are attempting to do, in order to redirect it to another function

B) Erasing the stack and placing the address of my function at the top and let the stack unwind itself to my function (The way this is written, you cannot receive any parameters through this way with my code, however)

I have uploaded it on Github as perhaps you will find it useful:
github.com/ghzcrlvct2/Thread-Stack-Overflow-Recovery

Nothing will happen, son.

1) From my understanding, allocating with `PAGE_EXECUTE` and then `WriteProcessMemory()` actually works since all `VirtualAllocEx()` allocates with all 0x00's to begin with and would never execute 0x00's without allowing a write to that memory, otherwise `PAGE_EXECUTE` would be a useless parameter. Even when I pass `PAGE_EXECUTE_READWRITE` to `VirtualAllocEx()` I'm still getting an Access Violation AFTER the "hijacked" function prints. Please correct me if I'm mistaken.

2) I'm passing the address of EIP not ESP... but either way the code cave works since the "hijacked" string is being printed before the Access Violation.

3) Ok, I've removed the second instance of `threadContext.ContextFlags = CONTEXT_CONTROL;`

4) Again, the "hijacked" string was printed so I don't think this would be an issue... correct me if I'm mistaken.

5) The whole point of the exercise is to understand how to manipulate the stack and cover your tracks, so I'd like to keep the function as `__cdecl` for the time being

6) What about the shellcode seems superfluous?

7) That will be the next step once I can get this portion of the code to work, but I don't want to get ahead of myself...

8) I appreciate the example, although it doesn't really point me towards a solution to my problem given my current code...

(Nov 1, 2020 01:35 PM)LunarManure Wrote: [CITE](#) 1) From my understanding, allocating with PAGE_EXECUTE and then WriteProcessMemory() actually works since all VirtualAllocEx() allocates with all 0x00's to begin with and would never execute 0x00's without allowing a write to that memory, otherwise PAGE_EXECUTE would be a useless parameter. Even when I pass PAGE_EXECUTE_READWRITE to VirtualAllocEx() I'm still getting an Access Violation AFTER the "hijacked" function prints. Please correct me if I'm mistaken.

2) I'm passing the address of EIP not ESP... but either way the code cave works since the "hijacked" string is being printed before the Access Violation.

3) Ok, I've removed the second instance of threadContext.ContextFlags = CONTEXT_CONTROL;

4) Again, the "hijacked" string was printed so I don't think this would be an issue... correct me if I'm mistaken.

5) The whole point of the exercise is to understand how to manipulate the stack and cover your tracks, so I'd like to keep the function as _cdecl for the time being

6) What about the shellcode seems superfluous?

7) That will be the next step once I can get this portion of the code to work, but I don't want to get ahead of myself...

8) I appreciate the example, although it doesn't really point me towards a solution to my problem given my current code...

```
Code #include <Windows.h>
#include <TIHelp32.h>
#include <iostream>
```

```
DWORD printStringManyTimes(int times, const char* string)
{
    for (int i = 0; i < times; i++)
        printf(string);
    return 0;
}
```

```
void __stdcall X_DummyThread(
    void * Parameter
)
{
    for (;;)
    {
        YieldProcessor();
        Sleep(1);
    }
}
```

```
bool X_GetProcessId(
    const wchar_t * ProcessName,
    unsigned int * ProcessId
)
{
    bool Status = false;
```

```

void          * Snapshot      = nullptr;
PROCESSENTRY32W Pe           = { 0 };
Pe.dwSize     = sizeof(Pe);
bool          Success        = false;

if (!ProcessName || !ProcessName[0] || !ProcessId)
{
    goto Exit;
}

*ProcessId    = 0;

Snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,
                                     0);
if (!Snapshot)
{
    return false;
}

Success = Process32FirstW(Snapshot,
                          &Pe);
while (Success)
{
    if (!_wcsicmp(ProcessName,
                  Pe.szExeFile))
    {
        *ProcessId = Pe.th32ProcessID;
        Status     = true;

        goto Cleanup;
    }

    Success = Process32NextW(Snapshot,
                             &Pe);
}

```

Cleanup:

```

    CloseHandle(Snapshot);

```

Exit:

```

    return Status;
}

```

```

bool X_AttachToProcess(
    unsigned int ProcessId,
    void        ** ProcessHandle
)
{
    bool Status = false;

```

```

    if (!ProcessId || !ProcessHandle)

```

```

{
    goto Exit;
}

*ProcessHandle = OpenProcess(PROCESS_ALL_ACCESS,
                             false,
                             ProcessId);
if (!*ProcessHandle)
{
    Status = true;
}

Exit:

    return Status;
}

void X_HijackThread(
    void      * ProcessHandle,
    void      * ThreadHandle,
    void      * Function,
    int       Times,
    const char * String
)
{
    bool      Status      = false;
    unsigned char Payload[] =
    {
        0x55,                // PUSH EBP
        0x89, 0xE5,          // MOV EBP, ESP
        0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0x00000000
        0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0x00000000
        0xE8, 0x00, 0x00, 0x00, 0x00, // CALL 0x00000000
        0x5D,                // POP EBP
        0xC3                 // RET
    };
    void      * PayloadMemory = nullptr;
    unsigned char * StringMemory = nullptr,
               * ShellcodeMemory = nullptr;
    unsigned int StringSize = 0;
    CONTEXT    ThreadContext = {};
    DWORD      OldProtection = 0;

    SuspendThread(ThreadHandle);

    StringSize = strlen(String) + 1;

    PayloadMemory = VirtualAllocEx(ProcessHandle,
                                   nullptr,
                                   sizeof(Payload) + StringSize,
                                   MEM_COMMIT | MEM_RESERVE,
                                   PAGE_READWRITE);
    StringMemory = reinterpret_cast<unsigned char *>(PayloadMemory);

```

```
ShellcodeMemory    = StringMemory + StringSize;
```

```
WriteProcessMemory(ProcessHandle,  
    StringMemory,  
    String,  
    StringSize,  
    nullptr);
```

```
ThreadContext.ContextFlags = CONTEXT_FULL;  
GetThreadContext(ThreadHandle,  
    &ThreadContext);
```

```
*reinterpret_cast<unsigned int*>(Payload + 4)    = reinterpret_cast<unsigned  
int>(StringMemory);  
*reinterpret_cast<unsigned int*>(Payload + 9)    = Times;  
*reinterpret_cast<unsigned int*>(Payload + 14)    = reinterpret_cast<unsigned int>(Function)  
    - reinterpret_cast<unsigned int>((ShellcodeMemory + 18));
```

```
ThreadContext.Eip                = reinterpret_cast<unsigned int>(ShellcodeMemory);
```

```
WriteProcessMemory(ProcessHandle,  
    ShellcodeMemory,  
    Payload,  
    sizeof(Payload),  
    nullptr);
```

```
VirtualProtectEx(ProcessHandle,  
    PayloadMemory,  
    sizeof(Payload),  
    PAGE_EXECUTE,  
    &OldProtection);
```

```
FlushInstructionCache(ProcessHandle,  
    PayloadMemory,  
    sizeof(Payload));
```

```
SetThreadContext(ThreadHandle,  
    &ThreadContext);
```

```
ResumeThread(ThreadHandle);
```

```
}
```

```
int main(  
    void  
)  
{  
    unsigned int    ProcessId        = 0;  
    void            * ProcessHandle    = nullptr;  
    void            * ThreadHandle    = nullptr;
```

```
ProcessId          = GetCurrentProcessId();  
X_AttachToProcess(ProcessId,  
    &ProcessHandle);
```

```

ThreadHandle = CreateThread(nullptr,
    0,
    reinterpret_cast<LPTHREAD_START_ROUTINE>(X_DummyThread),
    nullptr,
    0,
    nullptr);

X_HijackThread(ProcessHandle,
    ThreadHandle,
    printStringManyTimes,
    200,
    "Test!");

for (;;)
{
    YieldProcessor();
    Sleep(1);
}

return 0;
}

```

LunarManure - November 2, 2020, 2:47 pm

(Nov 2, 2020 05:46 AM)cghhhhasd Wrote: [\[CITE\]](#) Code #include <Windows.h>

#include <TIHelp32.h>

#include <iostream>

```

DWORD printStringManyTimes(int times, const char* string)
{
    for (int i = 0; i < times; i++)
        printf(string);
    return 0;
}

```

```

void __stdcall X_DummyThread(
    void * Parameter
)
{
    for (;;)
    {
        YieldProcessor();
        Sleep(1);
    }
}

```

```

bool X_GetProcessId(
    const wchar_t * ProcessName,
    unsigned int * ProcessId
)
{
    bool Status = false;
    void * Snapshot = nullptr;
    PROCESSENTRY32W Pe = { 0 };
}

```

```
Pe.dwSize          = sizeof(Pe);  
bool      Success  = false;
```

```
if (!ProcessName || !ProcessName[0] || !ProcessId)  
{  
    goto Exit;  
}
```

```
*ProcessId          = 0;
```

```
Snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,  
                                     0);
```

```
if (!Snapshot)  
{  
    return false;  
}
```

```
Success = Process32FirstW(Snapshot,  
                           &Pe);
```

```
while (Success)  
{  
    if (!_wcsicmp(ProcessName,  
                  Pe.szExeFile))  
    {  
        *ProcessId = Pe.th32ProcessID;  
        Status     = true;  
  
        goto Cleanup;  
    }
```

```
    Success = Process32NextW(Snapshot,  
                              &Pe);  
}
```

Cleanup:

```
    CloseHandle(Snapshot);
```

Exit:

```
    return Status;  
}
```

```
bool X_AttachToProcess(  
    unsigned int  ProcessId,  
    void          ** ProcessHandle  
)  
{  
    bool Status = false;
```

```
    if (!ProcessId || !ProcessHandle)  
    {  
        goto Exit;
```

```

}

*ProcessHandle = OpenProcess(PROCESS_ALL_ACCESS,
                             false,
                             ProcessId);
if (!*ProcessHandle)
{
    Status = true;
}

```

Exit:

```

    return Status;
}

```

```

void X_HijackThread(
    void      * ProcessHandle,
    void      * ThreadHandle,
    void      * Function,
    int       Times,
    const char * String
)
{
    bool      Status      = false;
    unsigned char Payload[] =
    {
        0x55, // PUSH EBP
        0x89, 0xE5, // MOV EBP, ESP
        0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0x00000000
        0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0x00000000
        0xE8, 0x00, 0x00, 0x00, 0x00, // CALL 0x00000000
        0x5D, // POP EBP
        0xC3 // RET
    };
    void      * PayloadMemory = nullptr;
    unsigned char * StringMemory = nullptr,
                * ShellcodeMemory = nullptr;
    unsigned int StringSize = 0;
    CONTEXT    ThreadContext = {};
    DWORD      OldProtection = 0;

    SuspendThread(ThreadHandle);

    StringSize = strlen(String) + 1;

    PayloadMemory = VirtualAllocEx(ProcessHandle,
                                   nullptr,
                                   sizeof(Payload) + StringSize,
                                   MEM_COMMIT | MEM_RESERVE,
                                   PAGE_READWRITE);
    StringMemory = reinterpret_cast<unsigned char *>(PayloadMemory);
    ShellcodeMemory = StringMemory + StringSize;
}

```

```

WriteProcessMemory(ProcessHandle,
    StringMemory,
    String,
    StringSize,
    nullptr);

ThreadContext.ContextFlags = CONTEXT_FULL;
GetThreadContext(ThreadHandle,
    &ThreadContext);

*reinterpret_cast<unsigned int *>(Payload + 4) = reinterpret_cast<unsigned int>(StringMemory);
*reinterpret_cast<unsigned int *>(Payload + 9) = Times;
*reinterpret_cast<unsigned int *>(Payload + 14) = reinterpret_cast<unsigned int>(Function
    - reinterpret_cast<unsigned int>((ShellcodeMemory + 18)));

ThreadContext.Eip = reinterpret_cast<unsigned int>(ShellcodeMemory);

WriteProcessMemory(ProcessHandle,
    ShellcodeMemory,
    Payload,
    sizeof(Payload),
    nullptr);

VirtualProtectEx(ProcessHandle,
    PayloadMemory,
    sizeof(Payload),
    PAGE_EXECUTE,
    &OldProtection);

FlushInstructionCache(ProcessHandle,
    PayloadMemory,
    sizeof(Payload));

SetThreadContext(ThreadHandle,
    &ThreadContext);

ResumeThread(ThreadHandle);
}

int main(
    void
)
{
    unsigned int ProcessId = 0;
    void * ProcessHandle = nullptr;
    void * ThreadHandle = nullptr;

    ProcessId = GetCurrentProcessId();
    X_AttachToProcess(ProcessId,
        &ProcessHandle);
    ThreadHandle = CreateThread(nullptr,
        0,
        reinterpret_cast<LPTHREAD_START_ROUTINE>(X_DummyThread),

```

```

        nullptr,
        0,
        nullptr);

X_HijackThread(ProcessHandle,
                ThreadHandle,
                printStringManyTimes,
                200,
                "Test!");

for (;;)
{
    YieldProcessor();
    Sleep(1);
}

return 0;
}

```

Your code still gives me an Access Violation...

cghhhhasd - November 2, 2020, 10:35 pm

(Nov 2, 2020 03:47 PM)LunarManure Wrote: [\[CITE\]](#) Your code still gives me an Access Violation... I didn't test it but it's supposed to work. I'm probably a bit confused so ill check it again once I get home.

LunarManure - November 16, 2020, 4:25 pm

(Nov 2, 2020 11:35 PM)cghhhhasd Wrote: [\[CITE\]](#) I didn't test it but it's supposed to work. I'm probably a bit confused so ill check it again once I get home.
Did you ever get around to testing your code?

cghhhhasd - November 17, 2020, 12:46 am

(Nov 16, 2020 05:25 PM)LunarManure Wrote: [\[CITE\]](#) Did you ever get around to testing your code? I totally forgot about your post, man, not going to lie.

Donkey Death Jam - November 19, 2020, 11:09 am

Hopefully this still helps you:

There was an error with the second shell code:

```

Code    BYTE codeCave[31] = {
        0x06, //PUSHAD - 0x06 is not the opcode for PUSHAD
    };

```

This ultimately messes up the 'ret' of the suspended main thread when it finishes with your shell code, thus resulting in an execution access error as the EIP shifts to a shitty location in memory.

Here is the fixed version:

```

Code    BYTE codeCave[31] = {
        0x60, //PUSHAD - 0x60 is correct
        0x9C, //PUSHFD
        0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
        0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
        0xB8, 0x00, 0x00, 0x00, 0x00, // MOV EAX, 0x0
    };

```

```

0xFF, 0xD0, // CALL EAX
0x83, 0xC4, 0x08, // ADD ESP, 0x08
0x9D, //POPF
0x61, //POPAD
0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
0xC3 // RETN
};

```

Utilising inline assembly within a function and reading the bytes of said function might be a good idea for the future.

LunarManure - December 10, 2020, 4:21 pm

(Nov 19, 2020 12:09 PM)Donkey Death Jam Wrote: [\[CITE\]](#) Hopefully this still helps you:

There was an error with the second shell code:

```

Code    BYTE codeCave[31] = {
    0x06, //PUSHAD - 0x06 is not the opcode for PUSHAD
};

```

This ultimately messes up the 'ret' of the suspended main thread when it finishes with your shell code, thus resulting in an execution access error as the EIP shifts to a shitty location in memory.

Here is the fixed version:

```

Code    BYTE codeCave[31] = {
    0x60, //PUSHAD - 0x60 is correct
    0x9C, //PUSHFD
    0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
    0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
    0xB8, 0x00, 0x00, 0x00, 0x00, // MOV EAX, 0x0
    0xFF, 0xD0, // CALL EAX
    0x83, 0xC4, 0x08, // ADD ESP, 0x08
    0x9D, //POPF
    0x61, //POPAD
    0x68, 0x00, 0x00, 0x00, 0x00, // PUSH 0
    0xC3 // RETN
};

```

Utilising inline assembly within a function and reading the bytes of said function might be a good idea for the future.

Wow I feel a bit embarrassed by not catching such an obvious typo...

Thanks for taking the time to shuffle through the code, though.

You're the real MVP. ?