← Windows Process Injection : Windows Notification Facility

Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL →

Shellcode: In-Memory Execution of DLL

Posted on June 24, 2019

Introduction

In March 2002, the infamous group 29A published their sixth e-zine. One of the articles titled In-Memory PE EXE Execution by Z0MBiE demonstrated how to manually load and run a Portable Executable entirely from memory. The InMem client provided as a PoC downloads a PE from a remote TFTP server into memory and after some basic preparation executes the entrypoint. Of course, running console and GUI applications from memory

Search

Recent Posts

- MiniDumpWriteDump via COM+ Services DLL
- Windows Process Injection: Asynchronous Procedure Call (APC)
- Windows Process Injection: KnownDlls Cache Poisoning
- Windows Process Injection: Tooltip or Common Controls
- Windows Process Injection: Breaking BaDDEr
- Windows Process Injection: DNS Client API

isn’t that straightforward because Microsoft Windows consists of subsystems. Try manually executing a console application from inside a GUI subsystem without using NtCreateProcess and it will probably cause an unhandled exception crashing the host process. Unless designed for a specific subsystem, running a DLL from memory is relatively error-free and simple to implement, so this post illustrates just that with C and x86 assembly.

Proof of Concept

ZOMBiE didn’t seem to perform any other research beyond a PoC, however, YOda did write a tool called InConEx that was published in 29A#7 ca. 2004. Since then, various other implementations have been published, but they all seem to be derived in one form or another from the original PoC and use the following steps.

1. Allocate RWX memory for size of image. (VirtualAlloc)
2. Copy each section to RWX memory.
3. Initialize the import table. (LoadLibrary/GetProcAddress)
4. Apply relocations.
5. Execute entry point.

Today, some basic loaders will also handle resources and TLS callbacks. The following is example in C based on ZOMBiE’s article.

```
typedef struct _IMAGE_RELOC {
    WORD offset :12;
    WORD type   :4;
} IMAGE_RELOC, *PIMAGE_RELOC;
```

- Windows Process Injection: Multiple Provider Router (MPR) DLL and Shell Notifications
- Windows Process Injection: Winsock Helper Functions (WSHX)
- Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL
- Shellcode: In-Memory Execution of DLL
- Windows Process Injection : Windows Notification Facility
- How Red Teams Bypass AMSI and WLDP for .NET Dynamic Code
- Windows Process Injection: KernelCallbackTable used by FinFisher / FinSpy
- Windows Process Injection: CLIPBRDWNDCLASS
- Shellcode: Using the Exception Directory to find GetProcAddress
- Shellcode: Loading .NET Assemblies From Memory
- Windows Process Injection: WordWarping, Hyphentension, AutoCourgette, Streamception, Oleum, ListPlanting, Treepoline
- Shellcode: A reverse shell for Linux in C with support for TLS/SSL
- Windows Process Injection: Print Spooler
- How the L0pht (probably) optimized attack against the LanMan hash.

```
typedef BOOL (WINAPI *DllMain_t)(HINSTANCE hinstDLL, DWORD fdwReason,
typedef VOID (WINAPI *entry_exe)(VOID);

VOID load_dllx(LPVOID base);

VOID load_dll(LPVOID base) {
    PIMAGE_DOS_HEADER        dos;
    PIMAGE_NT_HEADERS        nt;
    PIMAGE_SECTION_HEADER    sh;
    PIMAGE_THUNK_DATA        oft, ft;
    PIMAGE_IMPORT_BY_NAME    ibn;
    PIMAGE_IMPORT_DESCRIPTOR imp;
    PIMAGE_RELOC             list;
    PIMAGE_BASE_RELOCATION   ibr;
    DWORD                    rva;
    PBYTE                    ofs;
    PCHAR                    name;
    HMODULE                  dll;
    ULONG_PTR                ptr;
    DllMain_t                DllMain;
    LPVOID                   cs;
    DWORD                    i, cnt;

    dos = (PIMAGE_DOS_HEADER)base;
    nt  = RVA2VA(PIMAGE_NT_HEADERS, base, dos->e_lfanew);

    // 1. Allocate RWX memory for file
    cs  = VirtualAlloc(
      NULL, nt->OptionalHeader.SizeOfImage,
      MEM_COMMIT | MEM_RESERVE,
      PAGE_EXECUTE_READWRITE);
```

- A Guide to ARM64 / AArch64 Assembly on Linux with Shellcodes and Cryptography
- Windows Process Injection: ConsoleWindowClass
- Windows Process Injection: Service Control Handler
- Windows Process Injection: Extra Window Bytes
- Windows Process Injection: PROPagate
- Shellcode: Encrypting traffic
- Shellcode: Synchronous shell for Linux in ARM32 assembly
- Windows Process Injection: Sharing the payload
- Windows Process Injection: Writing the payload
- Shellcode: Synchronous shell for Linux in amd64 assembly
- Shellcode: Synchronous shell for Linux in x86 assembly
- Stopping the Event Logger via Service Control Handler
- Shellcode: Encryption Algorithms in ARM Assembly
- Shellcode: A Tweetable Reverse Shell for x86 Windows
- Polymorphic Mutex Names
- Shellcode: Linux ARM (AArch64)
- Shellcode: Linux ARM Thumb mode
- Shellcode: Windows API hashing with block ciphers (Maru Hash)
- Using Windows Schannel for Covert Communication

```
// 2. Copy each section to RWX memory
sh = IMAGE_FIRST_SECTION(nt);

for(i=0; i<nt->FileHeader.NumberOfSections; i++) {
  memcpy((PBYTE)cs + sh[i].VirtualAddress,
      (PBYTE)base + sh[i].PointerToRawData,
      sh[i].SizeOfRawData);
}

// 3. Process the Import Table
rva = nt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMP(
imp = RVA2VA(PIMAGE_IMPORT_DESCRIPTOR, cs, rva);

// For each DLL
for (;imp->Name!=0; imp++) {
  name = RVA2VA(PCHAR, cs, imp->Name);

  // Load it
  dll = LoadLibrary(name);

  // Resolve the API for this library
  oft = RVA2VA(PIMAGE_THUNK_DATA, cs, imp->OriginalFirstThunk);
  ft  = RVA2VA(PIMAGE_THUNK_DATA, cs, imp->FirstThunk);

  // For each API
  for (;; oft++, ft++) {
    // No API left?
    if (oft->u1.AddressOfData == 0) break;

    PULONG_PTR func = (PULONG_PTR)&ft->u1.Function;
```

- Shellcode: x86 optimizations part 1
- WanaCryptor File Encryption and Decryption
- Shellcode: Dual Mode (x86 + amd64) Linux shellcode
- Shellcode: Fido and how it resolves GetProcAddress and LoadLibraryA
- Shellcode: Dual mode PIC for x86 (Reverse and Bind Shells for Windows)
- Shellcode: Solaris x86
- Shellcode: Mac OSX amd64
- Shellcode: Resolving API addresses in memory
- Shellcode: A Windows PIC using RSA-2048 key exchange, AES-256, SHA-3
- Shellcode: Execute command for x32/x64 Linux / Windows / BSD
- Shellcode: Detection between Windows/Linux/BSD on x86 architecture
- Shellcode: FreeBSD / OpenBSD amd64
- Shellcode: Linux amd64
- Shellcodes: Executing Windows and Linux Shellcodes
- DLL/PIC Injection on Windows from Wow64 process
- Asmcodes: Platform Independent PIC for Loading DLL and Executing Commands

```
        // Resolve by ordinal?
        if (IMAGE_SNAP_BY_ORDINAL(oft->u1.Ordinal)) {
          *func = (ULONG_PTR)GetProcAddress(dll, (LPCSTR)IMAGE_ORDINA
        } else {
          // Resolve by name
          ibn   = RVA2VA(PIMAGE_IMPORT_BY_NAME, cs, oft->u1.AddressOf
          *func = (ULONG_PTR)GetProcAddress(dll, ibn->Name);
        }
      }
    }

    // 4. Apply Relocations
    rva  = nt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BAS
    ibr  = RVA2VA(PIMAGE_BASE_RELOCATION, cs, rva);
    ofs  = (PBYTE)cs - nt->OptionalHeader.ImageBase;

    while(ibr->VirtualAddress != 0) {
      list = (PIMAGE_RELOC)(ibr + 1);

      while ((PBYTE)list != (PBYTE)ibr + ibr->SizeOfBlock) {
        if(list->type == IMAGE_REL_TYPE) {
          *(ULONG_PTR*)((PBYTE)cs + ibr->VirtualAddress + list->offse
        }
        list++;
      }
      ibr = (PIMAGE_BASE_RELOCATION)list;
    }

    // 5. Execute entrypoint
    DllMain = RVA2VA(DllMain_t, cs, nt->OptionalHeader.AddressOfEntry
    DllMain(cs, DLL_PROCESS_ATTACH, NULL);
}
```

x86 assembly

Using the exact same logic except implemented in hand-written assembly ... for illustration of course!.

```
; DLL loader in 306 bytes of x86 assembly (written for fun)
; odzhan

      %include "ds.inc"

      bits   32

      struc _ds
          .VirtualAlloc        resd 1 ; edi
          .LoadLibraryA        resd 1 ; esi
          .GetProcAddress      resd 1 ; ebp
          .AddressOfEntryPoint resd 1 ; esp
          .ImportTable         resd 1 ; ebx
          .BaseRelocationTable resd 1 ; edx
          .ImageBase           resd 1 ; ecx
      endstruc

      %ifndef BIN
        global load_dllx
        global _load_dllx
      %endif
```

```
        load_dllx:
        _load_dllx:
              pop    eax              ; eax = return address
              pop    ebx              ; ebx = base of PE file
              push   eax              ; save return address on stack
              pushad                  ; save all registers
              call   init_api         ; load address of api hash onto stack
              dd     0x38194E37       ; VirtualAlloc
              dd     0xFA183D4A       ; LoadLibraryA
              dd     0x4AAC90F7       ; GetProcAddress
        init_api:
              pop    esi              ; esi = api hashes
              pushad                  ; allocate 32 bytes of memory for _ds
              mov    edi, esp         ; edi = _ds
              push   TEB.ProcessEnvironmentBlock
              pop    ecx
              cdq                     ; eax should be < 0x80000000
        get_apis:
              lodsd                   ; eax = hash
              pushad
              mov    eax, [fs:ecx]
              mov    eax, [eax+PEB.Ldr]
              mov    edi, [eax+PEB_LDR_DATA.InLoadOrderModuleList + LIST_ENTR
              jmp    get_dll
        next_dll:
              mov    edi, [edi+LDR_DATA_TABLE_ENTRY.InLoadOrderLinks + LIST_E
        get_dll:
              mov    ebx, [edi+LDR_DATA_TABLE_ENTRY.DllBase]
              mov    eax, [ebx+IMAGE_DOS_HEADER.e_lfanew]
              ; ecx = IMAGE_DATA_DIRECTORY.VirtualAddress
              mov    ecx, [ebx+eax+IMAGE_NT_HEADERS.OptionalHeader + \
                                     IMAGE_OPTIONAL_HEADER32.DataDirectory + \
                                   IMAGE_DIRECTORY_ENTRY_EXPORT * IMAGE_DATA_
                                   IMAGE_DATA_DIRECTORY.VirtualAddress]
              jecxz  next_dll
              ; esi = offset IMAGE_EXPORT_DIRECTORY.NumberOfNames
              lea    esi, [ebx+ecx+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
              lodsd
              xchg   eax, ecx
              jecxz  next_dll        ; skip if no names
              ; ebp = IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
              lodsd
              add    eax, ebx        ; ebp = RVA2VA(eax, ebx)
              xchg   eax, ebp        ;
              ; edx = IMAGE_EXPORT_DIRECTORY.AddressOfNames
              lodsd
              add    eax, ebx        ; edx = RVA2VA(eax, ebx)
              xchg   eax, edx        ;
              ; esi = IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
              lodsd
              add    eax, ebx        ; esi = RVA(eax, ebx)
              xchg   eax, esi
        get_name:
              pushad
              mov    esi, [edx+ecx*4-4] ; esi = AddressOfNames[ecx-1]
              add    esi, ebx           ; esi = RVA2VA(esi, ebx)
              xor    eax, eax           ; eax = 0
              cdq                       ; h = 0
        hash_name:
              lodsb
              add    edx, eax
              ror    edx, 8
              dec    eax
              jns    hash_name
```

```
cmp    edx, [esp + _eax + pushad_t_size]   ; hashes match?
popad
loopne get_name              ; --ecx && edx != hash
jne    next_dll              ; get next DLL
movzx  eax, word [esi+ecx*2] ; eax = AddressOfNameOrdinals[eax]
add    ebx, [ebp+eax*4]      ; ecx = base + AddressOfFunctions|
mov    [esp+_eax], ebx
popad                        ; restore all
stosd
inc    edx
jnp    get_apis              ; until PF = 1

; dos = (PIMAGE_DOS_HEADER)ebx
push   ebx
add    ebx, [ebx+IMAGE_DOS_HEADER.e_lfanew]
add    ebx, ecx
; esi = &nt->OptionalHeader.AddressOfEntryPoint
lea    esi, [ebx+IMAGE_NT_HEADERS.OptionalHeader + \
                 IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint -
movsd          ; [edi+ 0] = AddressOfEntryPoint
mov    eax, [ebx+IMAGE_NT_HEADERS.OptionalHeader + \
                 IMAGE_OPTIONAL_HEADER32.DataDirectory + \
                 IMAGE_DIRECTORY_ENTRY_IMPORT * IMAGE_DATA_DIRE
                 IMAGE_DATA_DIRECTORY.VirtualAddress - 30h]
stosd          ; [edi+ 4] = Import Directory Table RVA
mov    eax, [ebx+IMAGE_NT_HEADERS.OptionalHeader + \
                 IMAGE_OPTIONAL_HEADER32.DataDirectory + \
                 IMAGE_DIRECTORY_ENTRY_BASERELOC * IMAGE_DATA_[
                 IMAGE_DATA_DIRECTORY.VirtualAddress - 30h]
stosd          ; [edi+ 8] = Base Relocation Table RVA
lodsd          ; skip BaseOfCode
lodsd          ; skip BaseOfData
```

```
      movsd          ; [edi+12] = ImageBase
      ; cs  = VirtualAlloc(NULL, nt->OptionalHeader.SizeOfImage,
      ;          MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
      push   PAGE_EXECUTE_READWRITE
      xchg   cl, ch
      push   ecx
      push   dword[esi + IMAGE_OPTIONAL_HEADER32.SizeOfImage - \
                   IMAGE_OPTIONAL_HEADER32.SectionAlignment]
      push   0                           ; NULL
      call   dword[esp + _ds.VirtualAlloc + 5*4]
      xchg   eax, edi                    ; edi = cs
      pop    esi                         ; esi = base

      ; load number of sections
      movzx  ecx, word[ebx + IMAGE_NT_HEADERS.FileHeader + \
                       IMAGE_FILE_HEADER.NumberOfSections - 30h
      ; edx = IMAGE_FIRST_SECTION()
      movzx  edx, word[ebx + IMAGE_NT_HEADERS.FileHeader + \
                       IMAGE_FILE_HEADER.SizeOfOptionalHeader -
      lea    edx, [ebx + edx + IMAGE_NT_HEADERS.OptionalHeader - 30h]
map_section:
      pushad
      add    edi, [edx + IMAGE_SECTION_HEADER.VirtualAddress]
      add    esi, [edx + IMAGE_SECTION_HEADER.PointerToRawData]
      mov    ecx, [edx + IMAGE_SECTION_HEADER.SizeOfRawData]
      rep    movsb
      popad
      add    edx, IMAGE_SECTION_HEADER_size
      loop   map_section
      mov    ebp, edi
      ; process the import table
      pushad
```

```
          mov    ecx, [esp + _ds.ImportTable + pushad_t_size]
          jecxz  imp_l2
          lea    ebx, [ecx + ebp]
    imp_l0:
          ; esi / oft = RVA2VA(PIMAGE_THUNK_DATA, cs, imp->OriginalFirstT
          mov    esi, [ebx+IMAGE_IMPORT_DESCRIPTOR.OriginalFirstThunk]
          add    esi, ebp
          ; edi / ft  = RVA2VA(PIMAGE_THUNK_DATA, cs, imp->FirstThunk);
          mov    edi, [ebx+IMAGE_IMPORT_DESCRIPTOR.FirstThunk]
          add    edi, ebp
          mov    ecx, [ebx+IMAGE_IMPORT_DESCRIPTOR.Name]
          add    ebx, IMAGE_IMPORT_DESCRIPTOR_size
          jecxz  imp_l2
          add    ecx, ebp         ; name = RVA2VA(PCHAR, cs, imp->Name);
          ; dll = LoadLibrary(name);
          push   ecx
          call   dword[esp + _ds.LoadLibraryA + 4 + pushad_t_size]
          xchg   edx, eax         ; edx = dll
    imp_l1:
          lodsd                   ; eax = oft->u1.AddressOfData, oft++;
          xchg   eax, ecx
          jecxz  imp_l0           ; if (oft->u1.AddressOfData == 0) break
          btr    ecx, 31
          jc     imp_Lx           ; IMAGE_SNAP_BY_ORDINAL(oft->u1.Ordinal
          ; RVA2VA(PIMAGE_IMPORT_BY_NAME, cs, oft->u1.AddressOfData)
          lea    ecx, [ebp + ecx + IMAGE_IMPORT_BY_NAME.Name]
    imp_Lx:
          ; eax = GetProcAddress(dll, ecx);
          push   edx
          push   ecx
          push   edx
          call   dword[esp + _ds.GetProcAddress + 3*4 + pushad_t_size]
          pop    edx
          stosd                   ; ft->u1.Function = eax
          jmp    imp_l1
    imp_l2:
          popad
          ; ibr  = RVA2VA(PIMAGE_BASE_RELOCATION, cs, dir[IMAGE_DIRECTORY
          mov    esi, [esp + _ds.BaseRelocationTable]
          add    esi, ebp
          ; ofs  = (PBYTE)cs - opt->ImageBase;
          mov    ebx, ebp
          sub    ebp, [esp + _ds.ImageBase]
    reloc_L0:
          ; while (ibr->VirtualAddress != 0) {
          lodsd                  ; eax = ibr->VirtualAddress
          xchg   eax, ecx
          jecxz  call_entrypoint
          lodsd                  ; skip ibr->SizeOfBlock
          lea    edi, [esi + eax - 8]
    reloc_L1:
          lodsw                  ; ax = *(WORD*)list;
          and    eax, 0xFFF      ; eax = list->offset
          jz     reloc_L2        ; IMAGE_REL_BASED_ABSOLUTE is used for p
          add    eax, ecx        ; eax += ibr->VirtualAddress
          add    eax, ebx        ; eax += cs
          add    [eax], ebp      ; *(DWORD*)eax += ofs
          ; ibr = (PIMAGE_BASE_RELOCATION)list;
    reloc_L2:
          ; (PBYTE)list != (PBYTE)ibr + ibr->SizeOfBlock
          cmp    esi, edi
          jne    reloc_L1
          jmp    reloc_L0
    call_entrypoint:
```

```
%ifndef EXE
    push   ecx                 ; lpvReserved
    push   DLL_PROCESS_ATTACH  ; fdwReason
    push   ebx                 ; HINSTANCE
    ; DllMain = RVA2VA(entry_exe, cs, opt->AddressOfEntryPoint);
    add    ebx, [esp + _ds.AddressOfEntryPoint + 3*4]
%else
    add    ebx, [esp + _ds.AddressOfEntryPoint]
%endif
    call   ebx
    popad                  ; release _ds
    popad                  ; restore registers
    ret
```

Running a DLL from memory isn’t difficult if we ignore the export table, resources, TLS and subsystem. The only requirement is that the DLL has a relocation section. The C generated assembly will be used in a new version of Donut while sources in this post can be found here.

Share this:

Twitter Facebook

★ Like

Be the first to like this.

Related

Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL
In "assembly"

Shellcode: Loading .NET Assemblies From Memory
In "assembly"

Windows Process Injection: Writing the payload
In "assembly"

This entry was posted in assembly, injection, programming, security, shellcode, windows and tagged DLL, EXE, in-memory, x86 assembly. Bookmark the permalink.

← Windows Process Injection : Windows Notification Facility

Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL →

Leave a Reply

Enter your comment here...