Shellcode: Using the Exception Directory to find GetProcAddress

Posted on May 19, 2019

Introduction

Let's say you want the location of the `GetProcAddress` API in memory, but you can't use the Import Address Table (IAT) or the Export Address Table (EAT). What other ways can you do it?. Perhaps there are many ways, but let me suggest one that's relatively simple to implement and only involves searching for immediate values in the code section. When

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

`GetProcAddress` or `GetProcAddressForCaller` cannot locate the address of a function in a dynamic library, they will return the error code **STATUS_ORDINAL_NOT_FOUND**. If we search in `kernelbase.dll` for this immediate value, we should land somewhere in the address range of these API. From there, we locate the entry point.

Method 1 (32-bit)

Search the code section (`.text`) of each Dynamic-link Library (DLL) for the immediate value `0xC0000138`. If we find it, reverse the direction of search until we find the prolog bytes. For `stdcall` convention, prolog bytes normally begin with `push ebp` and `mov ebp, esp`. If the prolog contains `mov edi, edi` we can safely skip that because it's only used for hot-patching systems after XP SP2. The following pseudo-code attempts to describe this idea.

```
func GetGPA
  set addr = 0

  foreach (DLL in PEB) and addr is 0
    for pos = start(DLL.text) to end(DLL.text) - 4
      if pos[0] equal to STATUS_ORDINAL_NOT_FOUND
        while (pos[0] not equal to prolog (push ebp, mov ebp, esp))
          set pos = pos - 1
        set addr = pos
        break
      end if
      set pos = pos + 1
    end for
  end for
```


- 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 Lopht (probably) optimized attack against the LanMan hash.

```
    set GetGPA = addr
  end func
```

The following code in C demonstrates the idea.

```
LPVOID GetGPA(VOID) {
    PPEB                  peb;
    PPEB_LDR_DATA         ldr;
    PLDR_DATA_TABLE_ENTRY dte;
    LPVOID                addr=NULL;
    BYTE                  c;
    PIMAGE_DOS_HEADER     dos;
    PIMAGE_NT_HEADERS     nt;
    PIMAGE_SECTION_HEADER sh;
    DWORD                 i, j, h;
    PBYTE                 cs;

    peb = (PPEB) __readfsdword(0x30);
    ldr = (PPEB_LDR_DATA)peb->Ldr;

    // for each DLL loaded
    for (dte=(PLDR_DATA_TABLE_ENTRY)ldr->InLoadOrderModuleList.Flink;
         dte->DllBase != NULL && addr == NULL;
         dte=(PLDR_DATA_TABLE_ENTRY)dte->InLoadOrderLinks.Flink)
    {
      // is this kernel32.dll or kernelbase.dll?
      for (h=i=0; i<dte->BaseDllName.Length/2; i++) {
        c = dte->BaseDllName.Buffer[i];
```


- 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

```
      h += (c | 0x20);
      h = ROTR32(h, 13);
    }
    if (h != 0x22901A8D) continue;

    dos = (PIMAGE_DOS_HEADER)dte->DllBase;
    nt  = RVA2VA(PIMAGE_NT_HEADERS, dte->DllBase, dos->e_lfanew);
    sh  = (PIMAGE_SECTION_HEADER)((LPBYTE)&nt->OptionalHeader +
           nt->FileHeader.SizeOfOptionalHeader);

    for (i=0; i<nt->FileHeader.NumberOfSections && addr == NULL; i+
      if (sh[i].Characteristics & IMAGE_SCN_MEM_EXECUTE) {
        cs = RVA2VA (PBYTE, dte->DllBase, sh[i].VirtualAddress);
        for(j=0; j<sh[i].Misc.VirtualSize - 4 && addr == NULL; j++)
          // is this STATUS_ORDINAL_NOT_FOUND?
          if(*(DWORD*)&cs[j] == 0xC0000138) {
            while(--j) {
              // is this the prolog?
              if(cs[j  ] == 0x55 &&
                 cs[j+1] == 0x8B &&
                 cs[j+2] == 0xEC) {
                addr = &cs[j];
                break;
              }
            }
          }
        }
      }
    }
    return addr;
}
```

- 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

This approach should work fine on 32-bit legacy systems, but not 64-bit systems.

Method 2 (64-bit)

The first method doesn't work for x64 builds because of compiler optimizations and different calling convention. `stdcall` is replaced with Microsoft `fastcall`, and chunking can break up a function over a wider address range. For 64-bit, both problems can be solved parsing the Exception Directory (`.pdata` section), which is an array of `IMAGE_RUNTIME_FUNCTION_ENTRY` structures. When an exception occurs, the dispatcher will enumerate this array until it finds the primary function associated with the address of exception, and will use the unwind information to try fix up the stack. You can find more information about x64 exception handling here.

```
typedef struct _IMAGE_RUNTIME_FUNCTION_ENTRY {
    ULONG BeginAddress;
    ULONG EndAddress;
    ULONG UnwindInfoAddress;
} _IMAGE_RUNTIME_FUNCTION_ENTRY, *_PIMAGE_RUNTIME_FUNCTION_ENTRY;
```

The following pseudo-code attempts to describe the idea..

```
  func GetGPA
    set addr = 0
```

```
  foreach (DLL in PEB) and addr is 0
    foreach runtime in DLL.DataDirectory[Exception] and addr is 0
      set baddr = runtime.BeginAddress
      set start = runtime.BeginAddress + DLL.DllBase
      set end   = runtime.EndAddress   + DLL.DllBase
      for start to end and addr is 0
        if start[0] == near conditional jump
          set rva = (*(DWORD*)(start + 2) + 6 + start) - DLL.DllBas
          foreach runtime in DLL.DataDirectory[Exception] and addr
            if rva == runtime.BeginAddress
              set start2 = runtime.BeginAddress + DLL.DllBase
              set end2   = runtime.EndAddress   + DLL.DllBase
              for start2 to end2
                if start2[0] == STATUS_ORDINAL_NOT_FOUND
                  addr = baddr + DLL.DllBase
                  break
                end if
              end for
            end if
          end foreach
        end if
      end for
    end foreach
  end foreach

  set GetGPA = addr
end func
```

The following code has been tested on 64-bit builds of Windows 7 and Windows 10. GetGPA returned the address of `GetProcAddress` in both tests.

```
LPVOID GetGPA(VOID) {
    PPEB                            peb;
    PPEB_LDR_DATA                   ldr;
    PLDR_DATA_TABLE_ENTRY           dte;
    LPVOID                          addr=NULL;
    BYTE                            c;
    PIMAGE_DOS_HEADER               dos;
    PIMAGE_NT_HEADERS               nt;
    PIMAGE_DATA_DIRECTORY           dir;
    PIMAGE_RUNTIME_FUNCTION_ENTRY   rf;
    DWORD                           i, j, h, rva, ba;
    PBYTE                           s1, e1, s2, e2;
    PUNWIND_INFO                    ui;

    peb = (PPEB) __readgsqword(0x60);
    ldr = (PPEB_LDR_DATA)peb->Ldr;

    for (dte=(PLDR_DATA_TABLE_ENTRY)ldr->InLoadOrderModuleList.Flink;
         dte->DllBase != NULL && addr == NULL;
         dte=(PLDR_DATA_TABLE_ENTRY)dte->InLoadOrderLinks.Flink)
    {
      // is this kernelbase.dll?
      for (h=0, i=0; i<dte->BaseDllName.Length/2; i++) {
        c = (BYTE)dte->BaseDllName.Buffer[i];
        h += (c | 0x20);
        h = ROTR32(h, 13);
      }
      // if not, skip it
      if (h != 0x22901A8D) continue;

      dos = (PIMAGE_DOS_HEADER)dte->DllBase;
```

```
nt  = RVA2VA(PIMAGE_NT_HEADERS, dte->DllBase, dos->e_lfanew);
dir = (PIMAGE_DATA_DIRECTORY)nt->OptionalHeader.DataDirectory;
rva = dir[IMAGE_DIRECTORY_ENTRY_EXCEPTION].VirtualAddress;
rf  = (PIMAGE_RUNTIME_FUNCTION_ENTRY) RVA2VA(ULONG_PTR, dte->Dl

// foreach runtime function and address not found
for(i=0; rf[i].BeginAddress != 0 && addr == NULL; i++) {
  ba = rf[i].BeginAddress;
  // we will search the code between BeginAddress and EndAddres
  s1 = (PBYTE)RVA2VA(ULONG_PTR, dte->DllBase, rf[i].BeginAddres
  e1 = (PBYTE)RVA2VA(ULONG_PTR, dte->DllBase, rf[i].EndAddress)

  // if chained unwind information is specified in the next ent
  ui = (PUNWIND_INFO)RVA2VA(ULONG_PTR, dte->DllBase, rf[i+1].Ur

  if(ui->Flags & UNW_FLAG_CHAININFO) {
    // find the last entry in the chain
    for(;;) {
      i++;
      e1 = (PBYTE)RVA2VA(ULONG_PTR, dte->DllBase, rf[i].EndAddr
      ui = (PUNWIND_INFO)RVA2VA(ULONG_PTR, dte->DllBase, rf[i].
      if(!(ui->Flags & UNW_FLAG_CHAININFO)) break;
    }
  }
  // for this address range minus the length of a near conditic
  while(s1 < (e1 - 6)) {
    // is the next instruction a near conditional jump?
    if(s1[0] == 0x0F && s1[1] >= 0x80 && s1[1] <= 0x8F) {
      // calculate the relative virtual address of jump
      rva = (DWORD)(((*(DWORD*)(s1 + 2)) + 6 + s1) - (PBYTE)dte
      // try find the rva in exception list
      for(j=0; rf[j].BeginAddress != 0 && addr == NULL; j++) {
```

```
                if(rf[j].BeginAddress == rva) {
                  s2 = (PBYTE)RVA2VA(ULONG_PTR, dte->DllBase, rf[j].Beg
                  e2 = (PBYTE)RVA2VA(ULONG_PTR, dte->DllBase, rf[j].End
                  // try find the error code in this address range
                  while(s2 < (e2 - 4)) {
                    // if this is STATUS_ORDINAL_NOT_FOUND
                    if(*(DWORD*)s2 == 0xC0000138) {
                      // calculate the virtual address of primary funct
                      addr = (PBYTE)RVA2VA(ULONG_PTR, dte->DllBase, ba
                      break;
                    }
                    s2++;
                  }
                }
              }
            }
            s1++;
          }
        }
      }
      return addr;
  }
```

[Sources here.]

Share this:

Twitter Facebook

Like 

One blogger likes this.

Related

Shellcode: Resolving API addresses in memory
In "assembly"

Shellcode: A reverse shell for Linux in C with support for TLS/SSL
In "assembly"

Shellcode: Fido and how it resolves GetProcAddress and LoadLibraryA
In "assembly"

This entry was posted in assembly, programming, security, shellcode, windows and tagged assembly, getprocaddress, shellcode, windows, x64, x86. Bookmark the permalink.

← Shellcode: Loading .NET Assemblies From Memory

Windows Process Injection: CLIPBRDWNDCLASS →

3 Responses to *Shellcode: Using the Exception Directory to find GetProcAddress*

1.
Pingback: *2017.06.22 - Daily Security Issue > ReverseNote*

2.
Pingback: *【技术分享】Shellcode编程之特征搜索定位GetProcAddress-安全路透社*

3.
Pingback: *【技术分享】Shellcode编程之特征搜索定位GetProcAddress – 安百科技*

Leave a Reply

Enter your comment here...

modexp