REV HW 1

응용수리학부 사이버보안전공
2017270718
박찬희

<목차>

Overview

1. Target

File	Detail	File Version	md5	sha256
firefox.exe	Firefox Browser Executable File	75.0	d388df6ed5ccbf1acdeda5af2d18cb0b	8bcfd8420d721cc0ca50c1bef653e63e013ce201dfcca5927228eb25c9abf606
mozglue.dll	dll used by firefox	75.0	6f5f3843fa88734e3cc5f72cff0c1be4	6b27c9019a3209f807cf5c3f5e78ed4c03717967811a7b94eddc63960e55c8c2

2. Environment

category	값
OS	Microsoft Windows 10 Pro
Version	10.0.17763 빌드 17763

분석 환경과 분석 대상 구동 환경은 동일하다.

firefox.exe

1. DOS Header

DOS_HEADER의 구조체는 다음과 같다.

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value

    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)

    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

대상 파일에 대해 위의 구조체를 기반으로 각 필드 데이터를 나누면 다음과 같다.

```
0000000  4D 5A 78 00 01 00 00 00 04 00 00 00 00 00 00 00
0000010  00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
0000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000030  00 00 00 00 00 00 00 00 00 00 00 00 78 00 00 00
```

① Magic number

0x5A4D (“MZ”) -> PE 파일의 시그니처를 나타낸다.

② Bytes on last page of file

0x0078

③ Pages in file

0x0001

④ Relocations

0x0000

⑤ Size of header in paragraphs

0x0004

⑥ Minimum extra paragraphs needed

0x0000

⑦ Maximum extra paragraphs needed

0x0000

⑧ Initial (relative) SS value

0x0000

⑨ Initial SP value

0x0000

⑩ Checksum

0x0000

⑪ Initial IP value

0x0000

⑫ Initial (relative) CS value

0x0000

⑬ File address of relocation table

0x0040

⑭ Overlay number

0x0000

⑮ Reserved words [4]

{ 0x0000, 0x0000, 0x0000, 0x0000 }

⑯ OEM identifier (for e_oeminfo)

0x0000

⑰ OEM information; e_oemid specific

0x0000

⑱ Reserved words [10]

{ 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000 }

⑲ File address of new exe header

0x00000078 -> NT Header의 시작 Offset을 나타낸다.

2. DOS Stub

DOS Stub 영역은 다음의 색칠된 영역과 같다. 갈색 영역은 실행 코드, 회색 영역은 데이터를 나타낸다.

```
0000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
0000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
0000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20
0000070  6D 6F 64 65 2E 24 00 00 50 45 00 00 64 86 09 00
```

3. NT Header

x64 PE 파일의 NT Header는 다음의 구조체로 구성된다.

```
typedef struct _IMAGE_NT_HEADERS64 {
  DWORD Signature;
  IMAGE_FILE_HEADER FileHeader;
  IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

각 각의 필드를 나타냈을 때, 빨간 영역은 PE 파일의 시그니처를 나타내고, 초록색 영역은 File Header, 노란색 영역은 x64 Optional Header를 나타낸다.

```
0000070  6D 6F 64 65 2E 24 00 00 50 45 00 00 64 86 09 00
0000080  E1 85 87 5E 00 00 00 00 00 00 00 00 F0 00 22 00
0000090  0B 02 0E 00 00 84 04 00 00 1E 04 00 00 00 00 00
00000A0  70 82 04 00 00 10 00 00 00 00 00 40 01 00 00 00
00000B0  00 10 00 00 00 02 00 00 06 00 01 00 00 00 00 00
00000C0  06 00 01 00 00 00 00 00 00 30 09 00 00 04 00 00
00000D0  88 BD 09 00 02 00 60 C1 00 00 80 00 00 00 00 00
00000E0  00 10 00 00 00 00 00 00 00 00 04 00 00 00 00 00
00000F0  00 10 00 00 00 00 00 00 00 00 00 00 10 00 00 00
0000100  31 FB 04 00 31 0D 00 00 62 08 05 00 68 01 00 00
0000110  00 F0 05 00 A8 25 03 00 00 90 05 00 18 2D 00 00
0000120  00 A6 08 00 C8 22 00 00 00 20 09 00 30 03 00 00
0000130  82 F1 04 00 1C 00 00 00 00 00 00 00 00 00 00 00
0000140  00 00 00 00 00 00 00 00 30 C1 04 00 28 00 00 00
0000150  F0 A0 04 00 00 01 00 00 00 00 00 00 00 00 00 00
0000160  10 15 05 00 40 0B 00 00 68 F7 04 00 E0 00 00 00
0000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

1) Signature

0x00004550 (“PE”) -> PE 파일의 시그니처를 나타낸다.

2) File Header

File Header의 구조체는 다음과 같다.

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Offset 0x7C에서부터 시작하는 File Header의 각 필드를 나타내면 다음과 같다.

```
0000070  6D 6F 64 65 2E 24 00 00 50 45 00 00 64 86 09 00
0000080  E1 85 87 5E 00 00 00 00 00 00 00 00 F0 00 22 00
```

① Machine

0x8664 (AMD64)

정의된 Machine type을 통해 AMD64를 타겟으로 빌드된 것을 확인할 수 있다.

```
#define IMAGE_FILE_MACHINE_IA64              0x0200  // Intel 64
#define IMAGE_FILE_MACHINE_MIPS16            0x0266  // MIPS
#define IMAGE_FILE_MACHINE_ALPHA64           0x0284  // ALPHA64
#define IMAGE_FILE_MACHINE_MIPSFPU           0x0366  // MIPS
#define IMAGE_FILE_MACHINE_MIPSFPU16         0x0466  // MIPS
#define IMAGE_FILE_MACHINE_AXP64             IMAGE_FILE_MACHINE_ALPHA64
#define IMAGE_FILE_MACHINE_TRICORE           0x0520  // Infineon
#define IMAGE_FILE_MACHINE_CEF               0x0CEF
#define IMAGE_FILE_MACHINE_EBC               0x0EBC  // EFI Byte Code
#define IMAGE_FILE_MACHINE_AMD64             0x8664  // AMD64 (K8)
#define IMAGE_FILE_MACHINE_M32R              0x9041  // M32R little-endian
#define IMAGE_FILE_MACHINE_ARM64             0xAA64  // ARM64 Little-Endian
#define IMAGE_FILE_MACHINE_CEE               0xC0EE
```

② Number Of Section

0x0009 -> PE 파일에 포함된 Section의 개수, 이 값을 이용하여 header에 연이어 나오는 section header의 개수를 알 수 있다.

③ Time Date Stamp

0x5E8785E1 -> UTC를 기준으로 파일의 생성 시간을 나타낸다. 해당 값은 GMT+9로 변환했을 때 [2020년 4월 4일 토요일 오전 3:52:17]을 나타낸다.

④ Pointer To Symbol Table

0x00000000

⑤ Number Of Symbols

0x00000000

⑥ Size Of Optional Header

0x00F0 -> Optional Header의 크기를 나타낸다. object 파일의 경우 이 값은 0으로 세팅된다.

⑦ Characteristics

0x0022 -> 해당 파일의 특성을 나타내는 플래그이다. 각각의 플래그는 다음을 참고

```
#define IMAGE_FILE_RELOCS_STRIPPED           0x0001  // Relocation info stripped from file.
#define IMAGE_FILE_EXECUTABLE_IMAGE          0x0002  // File is executable  (i.e. no unresolved external references).
#define IMAGE_FILE_LINE_NUMS_STRIPPED        0x0004  // Line nunbers stripped from file.
#define IMAGE_FILE_LOCAL_SYMS_STRIPPED       0x0008  // Local symbols stripped from file.
#define IMAGE_FILE_AGGRESIVE_WS_TRIM         0x0010  // Aggressively trim working set
#define IMAGE_FILE_LARGE_ADDRESS_AWARE       0x0020  // App can handle >2gb addresses
#define IMAGE_FILE_BYTES_REVERSED_LO         0x0080  // Bytes of machine word are reversed.
#define IMAGE_FILE_32BIT_MACHINE             0x0100  // 32 bit word machine.
#define IMAGE_FILE_DEBUG_STRIPPED            0x0200  // Debugging info stripped from file in .DBG file
#define IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP   0x0400  // If Image is on removable media, copy and run from the swap file.
#define IMAGE_FILE_NET_RUN_FROM_SWAP         0x0800  // If Image is on Net, copy and run from the swap file.
#define IMAGE_FILE_SYSTEM                    0x1000  // System File.
#define IMAGE_FILE_DLL                       0x2000  // File is a DLL.
#define IMAGE_FILE_UP_SYSTEM_ONLY            0x4000  // File should only be run on a UP machine
#define IMAGE_FILE_BYTES_REVERSED_HI         0x8000  // Bytes of machine word are reversed.
```

3) Optional Header64

x64 Optional Header 구조체는 다음과 같다.

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
    WORD        Magic;
    BYTE        MajorLinkerVersion;
    BYTE        MinorLinkerVersion;
    DWORD       SizeOfCode;
    DWORD       SizeOfInitializedData;
    DWORD       SizeOfUninitializedData;
    DWORD       AddressOfEntryPoint;
    DWORD       BaseOfCode;
    ULONGLONG   ImageBase;
    DWORD       SectionAlignment;
    DWORD       FileAlignment;
    WORD        MajorOperatingSystemVersion;
    WORD        MinorOperatingSystemVersion;
    WORD        MajorImageVersion;
    WORD        MinorImageVersion;
    WORD        MajorSubsystemVersion;
    WORD        MinorSubsystemVersion;
    DWORD       Win32VersionValue;
    DWORD       SizeOfImage;
    DWORD       SizeOfHeaders;
    DWORD       CheckSum;
    WORD        Subsystem;
    WORD        DllCharacteristics;
    ULONGLONG   SizeOfStackReserve;
    ULONGLONG   SizeOfStackCommit;
    ULONGLONG   SizeOfHeapReserve;
    ULONGLONG   SizeOfHeapCommit;
    DWORD       LoaderFlags;
    DWORD       NumberOfRvaAndSizes;

    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

Optional Header는 0x90에서 시작하면 각 각의 필드를 나타내면 다음과 같다.

```
0000090  0B 02 0E 00 00 84 04 00 00 1E 04 00 00 00 00 00
00000A0  70 82 04 00 00 10 00 00 00 00 00 40 01 00 00 00
00000B0  00 10 00 00 00 02 00 00 06 00 01 00 00 00 00 00
00000C0  06 00 01 00 00 00 00 00 00 30 09 00 00 04 00 00
00000D0  88 BD 09 00 02 00 60 C1 00 00 80 00 00 00 00 00
00000E0  00 10 00 00 00 00 00 00 00 00 04 00 00 00 00 00
00000F0  00 10 00 00 00 00 00 00 00 00 00 00 10 00 00 00
0000100  31 FB 04 00 31 0D 00 00 62 08 05 00 68 01 00 00
0000110  00 F0 05 00 A8 25 03 00 00 90 05 00 18 2D 00 00
0000120  00 A6 08 00 C8 22 00 00 00 20 09 00 30 03 00 00
0000130  82 F1 04 00 1C 00 00 00 00 00 00 00 00 00 00 00
0000140  00 00 00 00 00 00 00 00 30 C1 04 00 28 00 00 00
0000150  F0 A0 04 00 00 01 00 00 00 00 00 00 00 00 00 00
0000160  10 15 05 00 40 0B 00 00 68 F7 04 00 E0 00 00 00
0000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

① Magic

0x020B (PE32+) -> PE32의 경우 0x010B가, PE32+의 경우 0x020B가 세팅된다.

② Major Linker Version

0x0E

③ Minor Linker Version

0x00

④ Size Of Code

0x00048400 -> Code Section의 크기를 나타낸다. Code Section이 .text 만 존재한다면 해당 영역의 크기를, Code Section이 여러 개 존재한다면 해당 Section 들의 크기 합을 나타낸다.

⑤ Size Of Initialized Data

0x00041E00 -> 초기화된 Data Section의 크기를 나타낸다. 여러 개 존재한다면 해당 Section 들의 크기 합을 나타낸다.

⑥ Size Of Uninitialized Data

0x00000000 -> 초기화되지 않은 Data Section (bss) 영역의 크기를 나타낸다. 여러 개 존재한다면 해당 Section 들의 크기 합을 나타낸다.

⑦ Address of Entry Point

0x00048270 -> 실행 가능한 파일이 메모리에 로드될 때, Entry Point의 image base에 대한 상대 주소를 나타낸다. Program Image에 대해서 이 주소는 시작 주소가 되며, Device Driver에 대해서는 Initialization Function의 주소가 된다. DLL에 대해서는 선택적으로 사용할 수 있다. Entry Point가 존재하지 않는다면 이 필드는 0으로 세팅된다.

⑧ Base Of Code

0x00001000 -> 실행 가능한 파일이 메모리에 로드될 때, 코드의 시작 Section의 image base에 대한 상대 주소를 나타낸다.

⑨ ImageBase

0x0000000140000000 -> 이미지가 메모리에 로드될 때, 이미지의 시작으로 선호되는 주소를 나타낸다. 반드시 64K의 배수가 되어야 한다. DLL의 경우 Default 값은 0x10000000이다.

⑩ Section Alignment

0x00001000 -> 메모리에 로드될 때, Section의 Alignment를 나타낸다. File Alignment보다 크거나 같아야 하며, Default 값은 해당 Architecture의 Page Size이다.

⑪ File Alignment

0x00000200 -> 이미지 파일 내부의 Section의 raw data를 align 하기 위해서 사용된다. 512 ~ 64K 사이의 2의 거듭제곱이어야 하며, Default 값은 512이다. 만약 Section Alignment 값이 Architecture의 Page Size보다 작다면, File Alignment 값은 Section Alignment 값과 동일해야 한다.

⑫ Major Operating System Version

0x0006

⑬ Minor Operating System Version

0x0001

⑭ Major Image Version

0x0000

⑮ Minor Image Version

0x0000

⑯ Major Subsystem Version

0x0006

⑰ Minor Subsystem Version

0x0001

⑱ Win32 Version Value

0x00000000 -> Reserved 영역, 0이어야 한다.

⑲ Size Of Image

0x00093000 -> 모든 Header를 포함한 상태로, 메모리에 로드되었을 때의 크기를 나타낸다. 해당 값은 Section Alignment의 배수여야 한다.

⑳ Size Of Headers

0x00000400 -> MS-DOS stub, PE header, section header의 크기를 모두 더한 값을 Section Alignment 값으로 반올림한 값을 나타낸다.

① CheckSum

0x0009BD88 -> 해당 이미지 파일에 대한 CheckSum을 나타낸다. 부팅 시에 로드되는 모든 Drivers, DLL 및 중요한 window process 안에 로드되는 DLL에 대해 유효성 검사를 한다.

② Subsystem

0x0002 -> 이미지를 실행할 때 필요한 Windows Subsystem을 나타낸다.

```
#define IMAGE_SUBSYSTEM_UNKNOWN              0   // Unknown subsystem.
#define IMAGE_SUBSYSTEM_NATIVE               1   // Image doesn't require a subsystem.
#define IMAGE_SUBSYSTEM_WINDOWS_GUI          2   // Image runs in the Windows GUI subsystem.
#define IMAGE_SUBSYSTEM_WINDOWS_CUI          3   // Image runs in the Windows character subsystem.
#define IMAGE_SUBSYSTEM_OS2_CUI              5   // image runs in the OS/2 character subsystem.
#define IMAGE_SUBSYSTEM_POSIX_CUI            7   // image runs in the Posix character subsystem.
#define IMAGE_SUBSYSTEM_NATIVE_WINDOWS       8   // image is a native Win9x driver.
#define IMAGE_SUBSYSTEM_WINDOWS_CE_GUI       9   // Image runs in the Windows CE subsystem.
#define IMAGE_SUBSYSTEM_EFI_APPLICATION      10  //
#define IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER  11   //
#define IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER   12  //
#define IMAGE_SUBSYSTEM_EFI_ROM              13
#define IMAGE_SUBSYSTEM_XBOX                 14
#define IMAGE_SUBSYSTEM_WINDOWS_BOOT_APPLICATION 16
#define IMAGE_SUBSYSTEM_XBOX_CODE_CATALOG    17
```

③ Dll Characteristics

0xC160

④ Size Of Stack Reserve

0x0000000000800000

⑤ Size Of Stack Commit

0x0000000000001000

⑥ Size Of Heap Reserve

0x0000000000040000

⑦ Size Of Heap Commit

0x0000000000001000

⑧ Loader Flags

0x00000000 -> Reserved 영역, 0이어야 한다.

⑨ Number Of Rva And Sizes

0x00000010 -> 뒤에 오는 Data Directory Entry의 개수를 나타낸다.

⑩ Data Directory [16]

Data Directory의 구조체는 다음과 같다.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

16개의 Entry를 사용하며 각 각의 Entry는 다음의 용도로 사용된다.

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0   // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1   // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2   // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3   // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4   // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5   // Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6   // Debug Directory
//      IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7   // (X86 usage)
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7   // Architecture Specific Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8   // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS             9   // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10   // Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT   11   // Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_IAT            12   // Import Address Table
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT   13   // Delay Load Import Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14   // COM Runtime descriptor
```

Entry에 따라 나타낸 Data Directory는 다음과 같다.

```
0000100  31 FB 04 00 31 0D 00 00 62 08 05 00 68 01 00 00
0000110  00 F0 05 00 A8 25 03 00 00 90 05 00 18 2D 00 00
0000120  00 A6 08 00 C8 22 00 00 00 20 09 00 30 03 00 00
0000130  82 F1 04 00 1C 00 00 00 00 00 00 00 00 00 00 00
0000140  00 00 00 00 00 00 00 00 30 C1 04 00 28 00 00 00
0000150  F0 A0 04 00 00 01 00 00 00 00 00 00 00 00 00 00
0000160  10 15 05 00 40 0B 00 00 68 F7 04 00 E0 00 00 00
0000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

(I) Virtual Address (IAT)

0x00050862 -> 해당 테이블이 메모리에 로드되었을 때, Image base를 기준으로 하는 RVA 값을 나타낸다.

(II) Size (IAT)

0x00000168 -> 해당 테이블의 사이즈를 나타낸다.

4. Section Header

File Header의 Number of Section 필드의 값을 통해 Section의 개수가 9인 것을 알 수 있으며, 실제 바이너리를 살펴보면 다음과 같이 9개의 Section을 볼 수 있다.

```
0000180  2E 74 65 78 74 00 00 00 30 83 04 00 00 10 00 00   .text...0.......
0000190  00 84 04 00 00 04 00 00 00 00 00 00 00 00 00 00   ................
00001A0  00 00 00 00 20 00 00 60 2E 72 64 61 74 61 00 00   ....... `.rdata..
00001B0  E4 BC 00 00 00 A0 04 00 00 BE 00 00 00 88 04 00   ................
00001C0  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40   ............@..@
00001D0  2E 64 61 74 61 00 00 00 58 29 00 00 00 60 05 00   .data...X)...`..
00001E0  00 02 00 00 00 46 05 00 00 00 00 00 00 00 00 00   .....F..........
00001F0  00 00 00 00 40 00 00 C0 2E 70 64 61 74 61 00 00   ....@....pdata..
0000200  18 2D 00 00 00 90 05 00 00 2E 00 00 00 48 05 00   .-...........H..
0000210  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40   ............@..@
0000220  2E 30 30 63 66 67 00 00 10 00 00 00 00 C0 05 00   .00cfg..........
0000230  00 02 00 00 00 76 05 00 00 00 00 00 00 00 00 00   .....v..........
0000240  00 00 00 00 40 00 00 40 2E 66 72 65 65 73 74 64   ....@..@.freestd
0000250  10 00 00 00 00 D0 05 00 00 02 00 00 00 78 05 00   .............x..
0000260  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40   ............@..@
0000270  2E 74 6C 73 00 00 00 00 11 00 00 00 00 E0 05 00   .tls............
0000280  00 02 00 00 00 7A 05 00 00 00 00 00 00 00 00 00   .....z..........
0000290  00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00   ....@....rsrc...
00002A0  A8 25 03 00 00 F0 05 00 00 26 03 00 00 7C 05 00   .%.......&...|..
00002B0  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40   ............@..@
00002C0  2E 72 65 6C 6F 63 00 00 30 03 00 00 00 20 09 00   .reloc..0.......
00002D0  00 04 00 00 00 A2 08 00 00 00 00 00 00 00 00 00   ................
00002E0  00 00 00 00 40 00 00 42 00 00 00 00 00 00 00 00   ....@..B........
```

1) Section Header 구조

Section Header 구조체는 다음과 같이 정의되어 있다.

```
#define IMAGE_SIZEOF_SHORT_NAME              8

typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

각 각의 필드를 나타내면 다음과 같다.

```
0000180  2E 74 65 78 74 00 00 00 30 83 04 00 00 10 00 00   .text...0.......
0000190  00 84 04 00 00 04 00 00 00 00 00 00 00 00 00 00   ................
00001A0  00 00 00 00 20 00 00 60 2E 72 64 61 74 61 00 00   ....... `.rdata..
```

① Name [8]

".text" -> Section의 이름을 나타낸다. 8바이트의 null-padded UTF-8로 표현된다. 정확히 8글자일 경우 null로 끝나지 않으며, 8바이트보다 길 경우, "/"와 string table의 Offset을 나타내는 ASCII 10진수로 표현된다. 실행 가능한 파일의 경우 string table을 지원하지 않아 8바이트 이후는 그냥 잘린다.

② Virtual Size

0x00048330 -> 메모리에 올라갔을 때, Section의 Total Size를 나타낸다. SizeOf RawData 값보다 크다면 0으로 패딩된다. 이 필드는 실행 가능한 이미지에 대해서만 유효하다. Object 파일의 경우 0으로 세팅된다.

③ Virtual Address

0x00001000 -> 실행 가능한 이미지의 경우, Section이 메모리에 로드될 때의 image base에 대한 상대 주소를 나타낸다. Object 파일의 경우, 재배치(relocation)이 적용되기 전의 시작 주소를 나타낸다. 편의상 컴파일러는 이 값을 0으로 세팅한다. 그렇지 않으면, 재배치 중 Offset에서 빼어야 하는 임의의 값이다.

④ Size Of RawData

0x00048400 -> Object 파일의 경우 Section의 크기를 나타내며, 이미지 파일의 경우 디스크에 존재하는 초기화된 데이터의 크기를 나타낸다. 실행 가능한 이미지 파일의 경우 Optional Header에 존재하는 File Alignment의 배수가 되어야 한다. 이 값이 Virtual Size보다 작다면, 해당 Section의 나머지 부분은 0으로 채워진다. Size Of RawData 필드는 반올림이 되지만, Virtual Size는 그렇지 않으므로, Size Of RawData가 Virtual Size보다 클 수 있다. Section에 초기화되지 않은 데이터만 포함할 경우, 이 필드는 0으로 세팅된다.

⑤ Pointer To RawData

0x00000400 -> COFF(Common Object File Format) 파일 내부의 Section의 첫 번째 페이지에 해당하는 파일 포인터이다. 이 값은 실행 가능한 이미지에 대해 Optional Header에 존재하는 FileAlignment 값의 배수여야만 한다. Object 파일의 경우, 최적의 성능을 위해 4byte 단위로 align 된다. Section이 초기화되지 않은 데이터만 포함할 경우, 0으로 세팅된다.

⑥ Pointer To Relocations

0x00000000 -> 해당 Section의 재배치(relocation) entry의 시작 주소를 나타내는 파일 포인터이다.

⑦ Pointer To Line numbers

0x00000000 -> 해당 Section의 line-number entry의 시작 주소를 나타낸다. COFF line number가 존재하지 않는다면 0으로 세팅된다. 이미지의 경우, COFF 디버깅 정보가 사용되지 않으므로, 0으로 세팅된다.

⑧ Number of Relocations

0x0000 -> 해당 Section의 재배치(relocation) entry의 개수를 나타낸다. 실행가능한 이미지의 경우 이 값은 0으로 세팅된다.

⑨ Number Of Line numbers

0x0000 -> 해당 Section의 line-number entry의 개수를 나타낸다. 이미지의 경우, COFF 디버깅 정보가 사용되지 않으므로, 0으로 세팅된다.

⑩ **Characteristic**

0x60000020 -> 해당 Section의 특징을 나타내는 플래그이다.
Section Flag는 다음과 같이 다양하게 존재하며, 본 필드의 값은 다음을 나타낸다.

IMAGE_SCN_CNT_CODE	0x00000020	The section contains executable code.
IMAGE_SCN_MEM_EXECUTE	0x20000000	The section can be executed as code.
IMAGE_SCN_MEM_READ	0x40000000	The section can be read.

2) 바이너리 구조 분석

firefox.exe 바이너리에 존재하는 9개의 각 Section Header 중 0으로 세팅된 필드를 제외한 필드를 분석한 결과는 다음과 같다.

Name	VirtualSize	VirtualAddress	SizeOfRawData	PointerToRawData	Characteristic
.text	0x00048330	0x00001000	0x00048400	0x00000400	0x60000020
.rdata	0x0000BCE4	0x0004A000	0x0000BE00	0x00048800	0x40000040
.data	0x00002958	0x00056000	0x00000200	0x00054600	0xC0000040
.pdata	0x00002D18	0x00059000	0x00002E00	0x00054800	0x40000040
.00cfg	0x00000010	0x0005C000	0x00000200	0x00057600	0x40000040
.freestd	0x00000010	0x0005D000	0x00000200	0x00057800	0x40000040
.tls	0x00000011	0x0005E000	0x00000200	0x00057A00	0xC0000040
.rsrc	0x000325A8	0x0005F000	0x00032600	0x00057C00	0x40000040
.reloc	0x00000330	0x00092000	0x00000400	0x0008A200	0x42000040

5. IAT 분석

Import Address Table의 주소는 Optional Header의 DataDirectory[1]을 참조함으로써 구할 수 있다. firefox.exe 에서는 다음의 영역 중 0x108에 위치한다.

```
0000100  31 FB 04 00 31 0D 00 00  62 08 05 00 68 01 00 00
0000110  00 F0 05 00 A8 25 03 00  00 90 05 00 18 2D 00 00
0000120  00 A6 08 00 C8 22 00 00  00 20 09 00 30 03 00 00
0000130  82 F1 04 00 1C 00 00 00  00 00 00 00 00 00 00 00
0000140  00 00 00 00 00 00 00 00  30 C1 04 00 28 00 00 00
0000150  F0 A0 04 00 00 01 00 00  00 00 00 00 00 00 00 00
0000160  10 15 05 00 40 0B 00 00  68 F7 04 00 E0 00 00 00
0000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

DataDirectory는 위에서 언급했다시피, _IMAGE_DATA_DIRECTORY 구조체로 표현되는

데, 이를 이용하여 Import Address Table Entry의 각 필드의 값을 나타내면 다음과 같다.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Virtual Address (RVA) : 0x00050862
Size : 0x00000168

앞선 Section Header 정보를 참고하였을 때, IAT의 RVA 값 0x00050862은 .rdata의 VirtualAddress 영역(0x0004A000 ~ 0x00055CE4)에 포함되므로, IAT의 RAW 값은 .rdata의 VirtualAddress를 빼고, PointerToRawData를 더함으로써 구할 수 있다.

IAT's Raw = 0x00050862 - 0x0004A000 + 0x00048800 = 0x0004F062

따라서, IAT의 Raw 값은 0x0004F062이다.

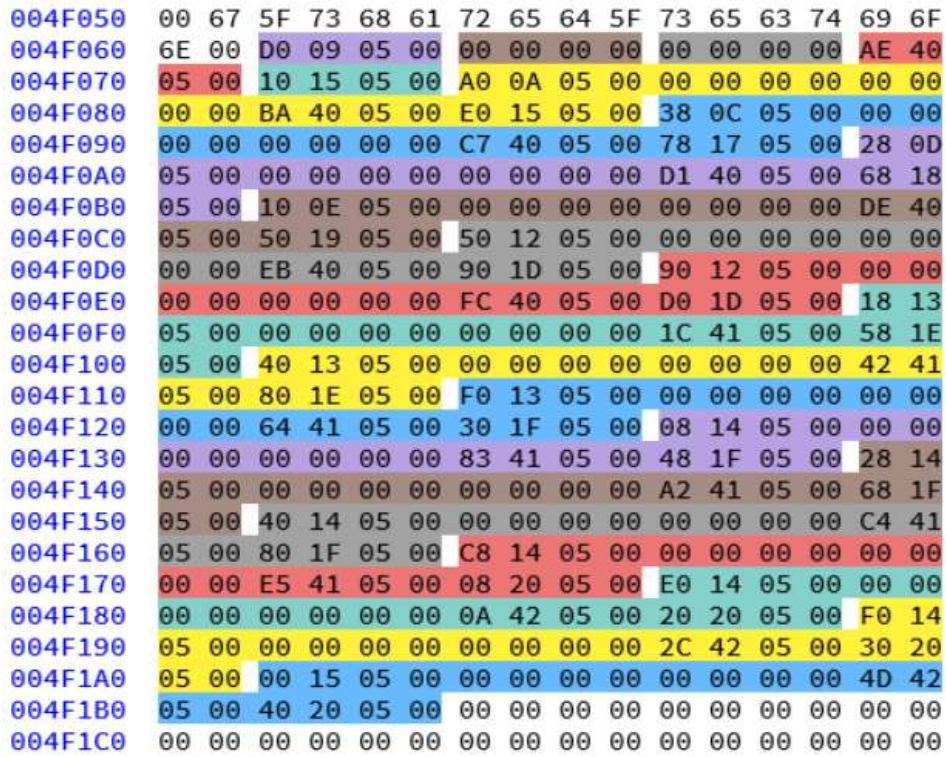
1) _IMAGE_IMPORT_DESCRIPTOR 분석

_IMAGE_DATA_DIRECTORY의 VirtualAddress는 다음의 _IMAGE_IMPORT_DESCRIPTOR의 주소를 나타낸다.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;            // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;         // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;                  // 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)

    DWORD   ForwarderChain;                 // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;                     // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

해당 구조체는 본 바이너리에 사용되는 라이브러리의 개수만큼 배열로 존재하며, 실제 해당 영역을 보면 다음과 같다.



_IMAGE_IMPORT_DESCRIPTOR는 총 20byte의 구조체이며, 위의 그림은 첫 번째 구조체만 각 필드 별로 구분하고 나머지 구조체는 단색으로 나타내었다. 배열의 마지막은 Null로 나타내기 때문에, firefox.exe에서는 총 17개의 라이브러리를 사용하는 것을 알 수 있다.

다음은 첫 번째 구조체를 분석한 결과이다.

① OriginalFirstThunk

0x000509D0 -> Import Name Table을 나타내는 RVA 값이다.
해당 RVA 값은 .rdata Section에 있으므로, Raw 값으로 변환하면 다음과 같다.

Raw = 0x000509D0 – 0x0004A000 + 0x00048800 = 0x0004F1D0

다음은 firefox.exe 파일 내부의 첫 번째 라이브러리의 Import Name Table List를 나타낸 것이다.

```
004F1D0  50 20 05 00 00 00 00 00 62 20 05 00 00 00 00 00
004F1E0  72 20 05 00 00 00 00 00 84 20 05 00 00 00 00 00
004F1F0  96 20 05 00 00 00 00 00 A8 20 05 00 00 00 00 00
004F200  E4 20 05 00 00 00 00 00 08 21 05 00 00 00 00 00
004F210  5C 21 05 00 00 00 00 00 B0 21 05 00 00 00 00 00
004F220  E6 21 05 00 00 00 00 00 3E 22 05 00 00 00 00 00
004F230  64 22 05 00 00 00 00 00 98 22 05 00 00 00 00 00
004F240  C0 22 05 00 00 00 00 00 04 23 05 00 00 00 00 00
004F250  2E 23 05 00 00 00 00 00 60 23 05 00 00 00 00 00
004F260  92 23 05 00 00 00 00 00 9C 23 05 00 00 00 00 00
004F270  A4 23 05 00 00 00 00 00 AE 23 05 00 00 00 00 00
004F280  BC 23 05 00 00 00 00 00 CE 23 05 00 00 00 00 00
004F290  D8 23 05 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Import Name Table은 다음의 _IMAGE_THUNK_DATA64 구조체 배열을 사용한다. Import 함수의 개수만큼 존재하며, 마지막은 NULL로 채워지므로, 첫 번째 라이브러리에 대하여 총 25개의 함수를 Import 하는 것을 확인할 수 있다.

```
typedef struct _IMAGE_THUNK_DATA64 {
    union {
        ULONGLONG ForwarderString;  // PBYTE
        ULONGLONG Function;         // PDWORD
        ULONGLONG Ordinal;
        ULONGLONG AddressOfData;    // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA64;
typedef IMAGE_THUNK_DATA64 * PIMAGE_THUNK_DATA64;
```

(I) AddressOfData

0x0000000000052050 -> 64bit의 경우 각 필드의 크기가 8byte로 확장되었으며, _IMAGE_IMPORT_BY_NAME의 RVA를 나타낸다.
.rdata Section에 존재하므로 Raw 값을 계산하면 다음과 같다.

Raw = 0x00052050 – 0x0004A000 + 0x00048800 = 0x00050850

해당 값이 가리키는 _IMAGE_IMPORT_BY_NAME 구조체는 다음과 같이 구성되어 있다.

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;
    CHAR    Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

실제 바이너리 내의 데이터는 다음과 같다.

```
0050850  00 00 3F 3F 32 40 59 41 50 45 41 58 5F 4B 40 5A  ..??2@YAPEAX_K@Z
0050860  00 00 00 00 3F 3F 33 40 59 41 58 50 45 41 58 40  ....??3@YAXPEAX@
```

Hint

0x0000 -> PE Loader가 Import하는 함수를 빠르게 찾기 위해 사용하

는 값이다. Import하는 라이브러리의 EAT의 Index 값을 나타내는데, 필수적이지 않기 때문에 0으로 세팅되는 경우도 있다.

Name

"???@YAPEAX_K@Z" -> Import하는 함수의 이름을 나타낸다. 본 필드의 값은 mangled 값이며, demangle 하였을 때, 다음의 값을 갖는다.
void * __ptr64 __cdecl operator new(unsigned __int64)

② TimeDateStamp

0x00000000 -> bound가 아닐 경우, 0으로 세팅된다.

③ ForwarderChain

0x00000000 -> Forwarder가 아닐 경우 -1로 세팅된다. 바인딩한 뒤에 사용되는 변수이다.

④ Name

0x000540AE -> 라이브러리의 이름이 저장되어 있는 주소의 RVA를 나타낸다.
해당 RVA 값을 Raw 값으로 변환하면, .rdata Section에 존재하므로 다음과 같다.

$$Raw = 0x000540AE - 0x0004A000 + 0x00048800 = 0x000528AE$$

해당 영역의 주소로 가면 다음과 같이 **mozglue.dll** 이라는 라이브러리의 이름을 확인할 수 있다.

```
00528A0  5F 73 65 74 5F 6E 65 77 5F 6D 6F 64 65 00 6D 6F   _set_new_mode.mo
00528B0  7A 67 6C 75 65 2E 64 6C 6C 00 41 44 56 41 50 49   zglue.dll.ADVAPI
00528C0  33 32 2E 64 6C 6C 00 6E 74 64 6C 6C 2E 64 6C 6C   32.dll.ntdll.dll
```

⑤ FirstThunk

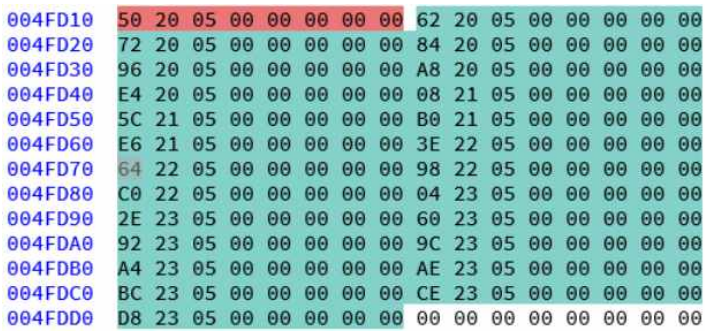
0x00051510 -> Import Address Table의 RVA 값을 나타낸다.
해당 값 역시, .rdata Section에 존재하므로 RVA 값을 Raw 값으로 변환하면 다음과 같다.

$$Raw = 0x00051510 - 0x0004A000 + 0x00048800 = 0x0004FD10$$

Import Address Table 역시, 앞선 INT와 마찬가지로, _IMAGE_THUNK_DATA64 구조체를 사용한다.

```
typedef struct _IMAGE_THUNK_DATA64 {
    union {
        ULONGLONG ForwarderString;  // PBYTE
        ULONGLONG Function;         // PDWORD
        ULONGLONG Ordinal;
        ULONGLONG AddressOfData;    // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA64;
typedef IMAGE_THUNK_DATA64 * PIMAGE_THUNK_DATA64;
```

실제 바이너리를 살펴보면 다음과 같다. 바인딩이 되기 이전에는 IAT는 INT와 동일한 것을 확인할 수 있다.



앞선 INT와 동일한 과정을 통해 Import Function Name을 구할 수 있다.

2) IAT 분석

앞선 INT 및 IAT 분석과정을 토대로 firefox.exe의 Import Entry를 분석한 결과는 다음과 같다.

Name RVA	DLL Name	OriginalFirstThunk	FirstThunk	Hint	Name
0x000540AE	mozglue.dll	0x000509D0	0x00051510	0x0000	??2@YAPEAX_K@Z
				0x0000	??3@YAXPEAX@Z
				0x0000	??3@YAXPEAX_K@Z
				0x0000	??_U@YAPEAX_K@Z
				0x0000	??_V@YAXPEAX@Z
				0x0000	?BeginProcessRuntimeInit@detail@mscom@mozilla@@YAAEA_NXZ
				...	...
0x000540BA	ADVAPI32.dll	0x00050AA0	0x000515E0	0x0005	AccessCheck
				0x005F	CheckTokenMembership
				0x007B	ConvertSidToStringSidW
				0x0081	ConvertStringSecurityDescriptorToSecurityDescriptorW
				...	...
...					
0x0005420A	api-ms-win-crt-utility-l1-1-0.dll	0x000514E0	0x00052020	0x001C	rand_s
0x0005422C	api-ms-win-crt-locale-l1-1-0.dll	0x000514F0	0x00052030	0x0008	_configthreadlocale
0x0005424D	api-ms-win-crt-heap-l1-1-0.dll	0x00051500	0x00052040	0x0016	_set_new_mode

2. EAT 분석

Export Address Table의 주소는 Optional Header의 DataDirectory[0]을 참조함으로써 구할 수 있다. firefox.exe 에서는 다음의 영역 중 0x100에 위치한다.

```
0000100  31 FB 04 00 31 0D 00 00 62 08 05 00 68 01 00 00
0000110  00 F0 05 00 A8 25 03 00 00 90 05 00 18 2D 00 00
0000120  00 A6 08 00 C8 22 00 00 00 20 09 00 30 03 00 00
0000130  82 F1 04 00 1C 00 00 00 00 00 00 00 00 00 00 00
0000140  00 00 00 00 00 00 00 00 30 C1 04 00 28 00 00 00
0000150  F0 A0 04 00 00 01 00 00 00 00 00 00 00 00 00 00
0000160  10 15 05 00 40 0B 00 00 68 F7 04 00 E0 00 00 00
0000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

_IMAGE_DATA_DIRECTORY 구조체의 각 필드의 값을 나타내면 다음과 같다.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Virtual Address (RVA) : 0x0004FB31
Size : 0x00000D31

RVA 값이 .rdata Section에 속하므로 다음과 같이 Raw 값을 구할 수 있다.

EAT's Raw = 0x0004FB31 - 0x0004A000 + 0x00048800 = 0x0004E331

1) _IMAGE_EXPORT_DIRECTORY 분석

Export Address Table은 다음과 같은 _IMAGE_EXPORT_DIRECTORY 구조체를 사용한다.

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

실제 바이너리 내부에서 살펴봤을 때, 다음과 같다.

```
004E330  00 00 00 00 00 00 00 00 00 00 00 00 59 FB 04
004E340  00 00 00 00 00 60 00 00 00 5F 00 00 00 65 FB 04
004E350  00 E5 FC 04 00 61 FE 04 00 66 69 72 65 66 6F 78
```

① **Characteristics**

0x000000

② **TimeDateStamp**

0x000000

③ **MajorVersion**

0x0000

④ **MinorVersion**

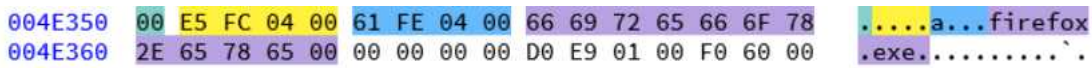
0x0000

⑤ **Name**

0x0004FB59 -> 바이너리의 이름이 저장된 주소의 RVA 값을 나타낸다.

Raw = 0x0004FB59 – 0x0004A000 + 0x00048800 = 0x0004E359

다음과 같이 firefox.exe가 저장된 것을 확인할 수 있다.



⑥ **Base**

0x00000000

⑦ **NumberOfFunctions**

0x00000060 -> Export 함수의 개수를 나타낸다.

⑧ **NumberOfNames**

0x0000005F -> Export 함수 중 이름이 존재하는 함수의 개수를 나타낸다. 이 값은 NumberOfFunctions보다 항상 작거나 같다.

⑨ **AddressOfFunctions**

0x0004FB65 -> Export 함수 주소 배열의 RVA 값을 나타낸다.

⑩ **AddressOfNames**

0x0004FCE5 -> (Name이 존재하는) Export 함수의 Name 배열의 RVA 값을 나타낸다.

⑪ **AddressOfNameOrdinals**

0x0004FE61 -> (Name이 존재하는) Export 함수에 대한 Ordinal 배열의 RVA 값을 나타낸다.

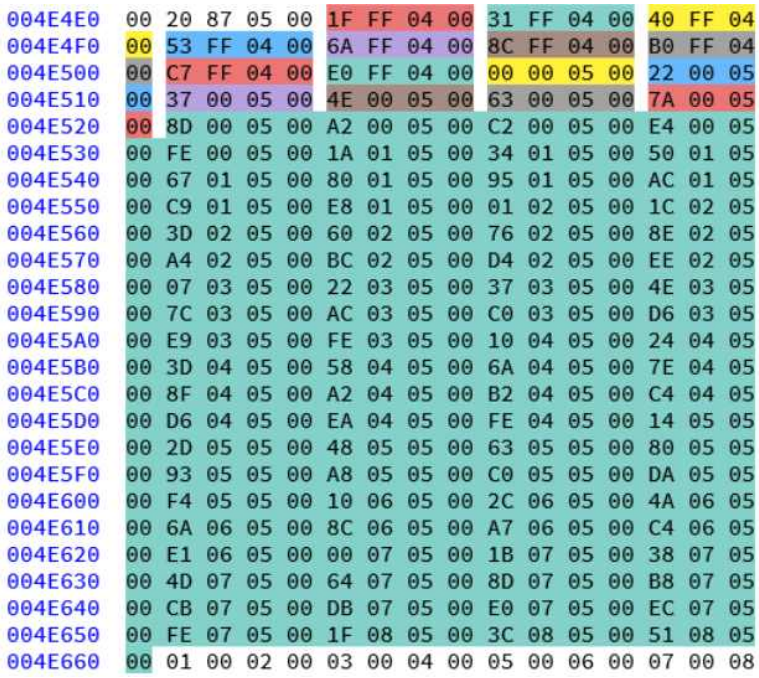
2) Name address list 분석

Name address list의 주소는 AddressOfNames 값을 이용하여 구할 수 있다.

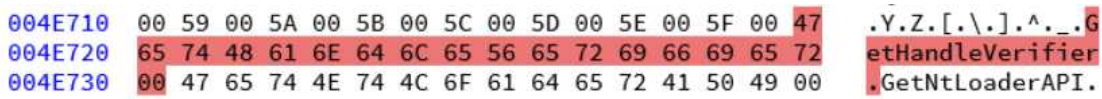
Raw = 0x0004FCE5 – 0x0004A000 + 0x00048800 = 0x0004E4E5

Name address list는 4byte의 주소를 나타내는 배열인데, Raw 값은 0x0004E4E5이며, NumberOfNames의 값이 0x5F이므로 0x0004E4E5 +4 * 0x5F 까지 Name address list임을 알 수 있다.

다음은 바이너리 내의 Name address list를 나타낸 것이다. (개수가 너무 많아 뒷부분은 단색으로 나타내었다.)



그중 가장 첫 번째에 위치하는 0x0004FF1F를 살펴보면, Raw로 변환했을 때의 값인 0x0004E71F에 해당 함수의 이름이 존재하는 것을 확인할 수 있다.



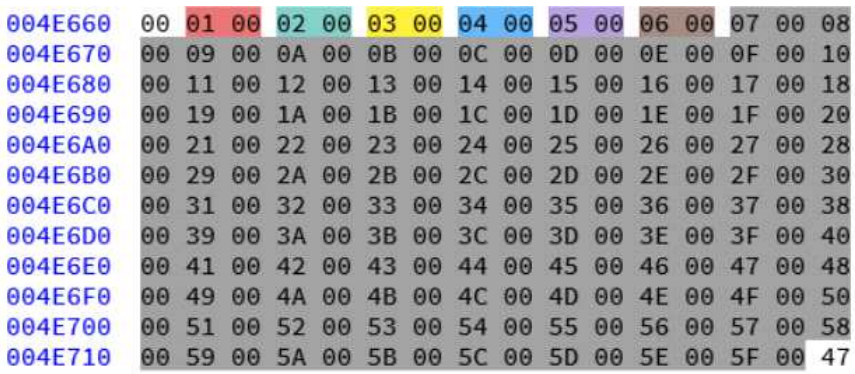
3) Ordinal list 분석

Ordinal list의 주소는 AddressOfNameOrdinals 값을 이용하여 구할 수 있다.

Raw = 0x0004FE61 - 0x0004A000 + 0x00048800 = 0x0004E661

Ordinal은 WORD로 표현되며, NumberOfNames의 값인 0x005F만큼 존재한다.

다음은 바이너리 내의 존재하는 Ordinal list를 나타낸 것이다. (뒷부분은 단색으로 나타내었다)



4) Function address list 분석

Function address list의 주소는 AddressOfFunctions 필드의 값을 이용하여 구할 수 있다.

Raw = 0x0004FB65 - 0x0004A000 + 0x00048800 = 0x0004E365

Function의 주소는 4byte로 표현되며, NumberOfFunctions 값인 0x60개 존재한다.

다음은 바이너리 내의 Function address list를 나타낸 것이다.

```
004E360  2E 65 78 65 00 00 00 00 00 D0 E9 01 00 F0 60 00
004E370  00 90 AB 03 00 00 26 01 00 50 29 03 00 A0 C9 02
004E380  00 D0 D2 02 00 40 C4 02 00 A0 33 03 00 50 C8 02
004E390  00 30 5E 03 00 B0 C5 02 00 30 59 03 00 50 C5 02
004E3A0  00 30 64 03 00 10 C6 02 00 F0 27 03 00 30 C9 02
004E3B0  00 30 8D 00 00 C0 C7 02 00 40 1D 03 00 90 C7 02
004E3C0  00 20 1D 03 00 50 C7 02 00 10 23 03 00 90 C8 02
004E3D0  00 D0 27 03 00 E0 C8 02 00 F0 25 03 00 C0 C8 02
004E3E0  00 E0 27 03 00 10 C9 02 00 E0 1F 03 00 F0 C7 02
004E3F0  00 A0 21 03 00 10 C8 02 00 D0 2B 03 00 40 C9 02
004E400  00 B0 2D 03 00 60 C9 02 00 30 8D 00 00 70 C7 02
004E410  00 10 30 03 00 30 C8 02 00 D0 EE 03 00 F0 C6 02
004E420  00 10 53 02 00 F0 C2 02 00 60 73 03 00 50 C6 02
004E430  00 A0 FB 03 00 C0 C1 02 00 C0 F0 03 00 30 C7 02
004E440  00 E0 57 02 00 80 C3 02 00 F0 7C 03 00 A0 C6 02
004E450  00 60 7D 03 00 C0 C6 02 00 20 54 03 00 D0 C4 02
004E460  00 B0 55 03 00 00 C5 02 00 60 57 03 00 20 C5 02
004E470  00 20 52 03 00 A0 C4 02 00 10 F9 02 00 80 C2 02
004E480  00 70 F9 02 00 B0 C2 02 00 D0 5B 02 00 C0 C3 02
004E490  00 60 5F 02 00 E0 C3 02 00 10 61 02 00 00 C4 02
004E4A0  00 60 F8 02 00 50 C2 02 00 80 FD 03 00 30 C2 02
004E4B0  00 30 1D 03 00 80 C7 02 00 90 32 03 00 80 C9 02
004E4C0  00 20 85 05 00 88 86 05 00 28 87 05 00 40 85 05
004E4D0  00 E0 87 05 00 88 60 05 00 50 88 05 00 E8 87 05
004E4E0  00 20 87 05 00 1F FF 04 00 31 FF 04 00 40 FF 04
```

주소 값이 0x00000000인 0번째 Function은 제외하고, 1번째 Function을 살펴보면 다음과 같다. 1번째 Function의 RVA 값은 0x0001E9D0인데, 해당 값은 .text Section에 해당한다.

Raw = 0x0001E9D0 - 0x00001000 + 0x00000400 = 0x0001DDD0

해당 Raw 위치로 가보면 다음과 같은 opcode를 확인할 수 있는데,

```
001DDD0  48 83 EC 28 48 8B 05 9D 9A 03 00 48 85 C0 74 05
001DDE0  48 83 C4 28 C3 E8 06 01 00 00 48 8B 05 87 9A 03
001DDF0  00 EB ED CC CC CC CC CC CC CC CC CC CC CC CC CC
```

실제로 디스어셈블러를 통해 살펴보면 다음과 같이 함수의 코드임을 확인할 수 있다.

```
00007FF76735E9D0   48:83EC 28          sub rsp,28                                  GetHandleVerifier
00007FF76735E9D4   48:8B05 9D9A0300    mov rax,qword ptr ds:[7FF767398478]
00007FF76735E9DB   48:85C0             test rax,rax
00007FF76735E9DE   74 05               je firefox.7FF76735E9E5
00007FF76735E9E0   48:83C4 28          add rsp,28
00007FF76735E9E4   C3                  ret
00007FF76735E9E5   E8 06010000         call firefox.7FF76735EAF0
00007FF76735E9EA   48:8B05 879A0300    mov rax,qword ptr ds:[7FF767398478]
00007FF76735E9F1   EB ED               jmp firefox.7FF76735E9E0
00007FF76735E9F3   CC                  int3
```

5) EAT 분석

위의 내용을 바탕으로 firefox.exe의 EAT를 분석한 내용은 다음과 같다.

Function Name	Ordinal	Address
GetHandleVerifier	0x0001	0x0001E9D0
GetNtLoaderAPI	0x0002	0x000060F0
IsSandboxedProcess	0x0003	0x0003AB90
NativeNtBlockSet_Write	0x0004	0x00012600
...	...	...
g_shared_policy_size	0x005E	0x000587E8
g_shared_section	0x005F	0x00058720

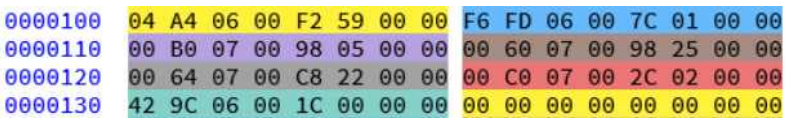
mozglue.dll

Section Header를 정리한 표는 다음과 같다.

Name	VirtualSize	VirtualAddress	SizeOfRawData	PointerToRawData
.text	0x00062FC1	0x00001000	0x00063000	0x00000400
.rdata	0x0000F624	0x00064000	0x0000F800	0x00063400
.data	0x00001520	0x00074000	0x00000400	0x00072C00
.pdata	0x00002598	0x00076000	0x00002600	0x00073000
.00cfg	0x00000010	0x00079000	0x00000200	0x00075600
.tls	0x00000021	0x0007A000	0x00000200	0x00075800
.rsrc	0x00000598	0x0007B000	0x00000600	0x00075A00
.reloc	0x0000022C	0x0007C000	0x00000400	0x00076000

1. IAT 분석

Import Address Table의 주소는 Optional Header의 DataDirectory[1]을 참조함으로써 구할 수 있다. mozglue.dll에서는 다음의 영역 중 0x108에 위치한다.



Import Address Table Entry의 각 필드의 값을 나타내면 다음과 같다.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Virtual Address (RVA) : 0x0006FDF6
Size : 0x0000017C

IAT의 RVA가 .rdata Section에 존재하므로 다음과 같이 Raw 값을 구할 수 있다.

IAT's Raw = 0x0006FDF6 - 0x00064000 + 0x00063400 = 0x0006F1F6

1) _IMAGE_IMPORT_DESCRIPTOR 분석

_IMAGE_DATA_DIRECTORY의 VirtualAddress는 다음의 _IMAGE_IMPORT_DESCRIPTOR의 주소를 나타낸다.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;            // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;         // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;                  // 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)

    DWORD   ForwarderChain;                 // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;                     // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

해당 구조체는 본 바이너리에 사용되는 라이브러리의 개수만큼 배열로 존재하며, 실제 해당 영역을 보면 다음과 같다.

```
006F1F0  63 73 64 75 70 00 78 FF 06 00 00 00 00 00 00 00
006F200  00 00 D4 23 07 00 00 07 07 00 88 FF 06 00 00 00
006F210  00 00 00 00 00 00 E1 23 07 00 10 07 07 00 D0 FF
006F220  06 00 00 00 00 00 00 00 00 00 ED 23 07 00 58 07
006F230  07 00 28 00 07 00 00 00 00 00 00 00 00 00 F7 23
006F240  07 00 B0 07 07 00 58 00 07 00 00 00 00 00 00 00
006F250  00 00 03 24 07 00 E0 07 07 00 68 00 07 00 00 00
006F260  00 00 00 00 00 00 10 24 07 00 F0 07 07 00 A8 00
006F270  07 00 00 00 00 00 00 00 00 00 1C 24 07 00 30 08
006F280  07 00 D8 01 07 00 00 00 00 00 00 00 00 00 29 24
006F290  07 00 60 09 07 00 68 04 07 00 00 00 00 00 00 00
006F2A0  00 00 36 24 07 00 F0 0B 07 00 C8 04 07 00 00 00
006F2B0  00 00 00 00 00 00 47 24 07 00 50 0C 07 00 68 05
006F2C0  07 00 00 00 00 00 00 00 00 00 67 24 07 00 F0 0C
006F2D0  07 00 F0 05 07 00 00 00 00 00 00 00 00 00 89 24
006F2E0  07 00 78 0D 07 00 18 06 07 00 00 00 00 00 00 00
006F2F0  00 00 A8 24 07 00 A0 0D 07 00 30 06 07 00 00 00
006F300  00 00 00 00 00 00 CD 24 07 00 B8 0D 07 00 50 06
006F310  07 00 00 00 00 00 00 00 00 00 EF 24 07 00 D8 0D
006F320  07 00 B8 06 07 00 00 00 00 00 00 00 00 00 10 25
006F330  07 00 40 0E 07 00 E0 06 07 00 00 00 00 00 00 00
006F340  00 00 2F 25 07 00 68 0E 07 00 F0 06 07 00 00 00
006F350  00 00 00 00 00 00 55 25 07 00 78 0E 07 00 00 00
```

_IMAGE_IMPORT_DESCRIPTOR는 총 20byte의 구조체이며, 위의 그림은 첫 번째 구조체만 각 필드 별로 구분하고 나머지 구조체는 단색으로 나타내었다. 배열의 마지막은 Null로 나타내기 때문에, mozglue.dll에서는 총 18개의 라이브러리를 사용하는 것을 알 수 있다.

다음은 첫 번째 구조체를 분석한 결과이다.

① OriginalFirstThunk

0x0006FF78 -> Import Name Table을 나타내는 RVA 값이다.
해당 RVA 값은 .rdata Section에 있으므로, Raw 값으로 변환하면 다음과 같다.

Raw = 0x0006FF78 - 0x00064000 + 0x00063400 = 0x0006F378

다음은 첫 번째 라이브러리의 Import Name Table List를 나타낸 것이다.

```
006F370  00 00 00 00 00 00 00 00 88 0E 07 00 00 00 00 00
006F380  00 00 00 00 00 00 00 00 9C 0E 07 00 00 00 00 00
```

Import Name Table은 다음의 _IMAGE_THUNK_DATA64 구조체 배열을 사용한다. Import 함수의 개수만큼 존재하며, 마지막은 NULL로 채워지므로, 첫 번째 라이브러리에 대하여, 총 1개의 함수를 Import하는 것을 확인할 수 있다.

```
typedef struct _IMAGE_THUNK_DATA64 {
    union {
        ULONGLONG ForwarderString;  // PBYTE
        ULONGLONG Function;         // PDWORD
        ULONGLONG Ordinal;
        ULONGLONG AddressOfData;    // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA64;
typedef IMAGE_THUNK_DATA64 * PIMAGE_THUNK_DATA64;
```

(I) AddressOfData

0x0000000000070E88

Raw = 0x00070E88 - 0x00064000 + 0x00063400 = 0x00070288

해당 값이 가리키는 _IMAGE_IMPORT_BY_NAME 구조체는 다음과 같이 구성되어 있다.

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;
    CHAR    Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

실제 바이너리 내의 데이터는 다음과 같다.

```
0070280  00 00 00 00 00 00 00 00 23 03 53 79 73 74 65 6D   ........#.System
0070290  46 75 6E 63 74 69 6F 6E 30 33 36 00 12 00 43 65   Function036...Ce
```

Hint

0x0323

Name

“SystemFunction036”

② TimeDateStamp

0x00000000

③ ForwarderChain

0x00000000

④ Name

0x000723D4

해당 RVA 값을 Raw 값으로 변환하면, .rdata Section에 존재하므로 다음과 같다.

$$Raw = 0x000723D4 - 0x00064000 + 0x00063400 = 0x000717D4$$

해당 영역의 주소로 가면 다음과 같이 ADVAPI32.dll 이라는 라이브러리의 이름을 확인할 수 있다.

```
00717D0  5F 73 00 00 41 44 56 41 50 49 33 32 2E 64 6C 6C   _s..ADVAPI32.dll
00717E0  00 43 52 59 50 54 33 32 2E 64 6C 6C 00 6E 74 64   .CRYPT32.dll.ntd
```

⑤ FirstThunk

0x00070700 -> Import Address Table의 RVA 값을 나타낸다.
해당 값 역시, .rdata Section에 존재하므로 RVA 값을 Raw 값으로 변환하면 다음과 같다.

$$Raw = 0x00070700 - 0x00064000 + 0x00063400 = 0x0006FB00$$

Import Address Table 역시, 앞선 INT와 마찬가지로, _IMAGE_THUNK_DATA64 구조체를 사용한다.

```
typedef struct _IMAGE_THUNK_DATA64 {
    union {
        ULONGLONG ForwarderString;  // PBYTE
        ULONGLONG Function;         // PDWORD
        ULONGLONG Ordinal;
        ULONGLONG AddressOfData;    // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA64;
typedef IMAGE_THUNK_DATA64 * PIMAGE_THUNK_DATA64;
```

실제 바이너리를 살펴보면 다음과 같다. 바인딩이 되기 이전에는 IAT는 INT와 동일한 것을 확인할 수 있다.

```
006FB00  88 0E 07 00 00 00 00 00 00 00 00 00 00 00 00 00
```

앞선 INT와 동일한 과정을 통해 Import Function Name을 구할 수 있다.

2) IAT 분석

앞선 INT 및 IAT 분석과정을 토대로 firefox.exe의 Import Entry를 분석한 결과는 다음

과 같다.

Name RVA	DLL Name	OriginalFirstThunk	FirstThunk	Hint	Name
0x000723D4	ADVAPI32.dll	0x0006FF78	0x00070700	0x0323	SystemFunction036
0x000723E1	CRYPT32.dll	0x0006FF88	0x00070710	0x0012	CertCloseStore
				0x0035	CertFindCertificateInStore
				0x0040	CertFreeCertificateContext
				...	...
...					
0x0007252F	api-ms-win-crt-environment-l1-1-0.dll	0x000706E0	0x00070E68	0x0010	getenv
0x00072555	api-ms-win-crt-utility-l1-1-0.dll	0x000706F0	0x00070E78	0x001C	rand_s

2. EAT 분석

Export Address Table의 주소는 Optional Header의 DataDirectory[0]을 참조함으로써 구할 수 있다. mozglue.dll 에서는 다음의 영역 중 0x100에 위치한다.

```
0000100  04 A4 06 00 F2 59 00 00 F6 FD 06 00 7C 01 00 00
0000110  00 B0 07 00 98 05 00 00 00 60 07 00 98 25 00 00
0000120  00 64 07 00 C8 22 00 00 00 C0 07 00 2C 02 00 00
0000130  42 9C 06 00 1C 00 00 00 00 00 00 00 00 00 00 00
```

_IMAGE_DATA_DIRECTORY 구조체의 각 필드의 값을 나타내면 다음과 같다.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Virtual Address (RVA) : 0x0006A404
Size : 0x000059F2

RVA 값이 .rdata Section에 속하므로 다음과 같이 Raw 값을 구할 수 있다.

EAT's Raw = 0x0006A404 - 0x00064000 + 0x00063400 = 0x00069804

1) _IMAGE_EXPORT_DIRECTORY 분석

Export Address Table은 다음과 같은 _IMAGE_EXPORT_DIRECTORY 구조체를 사용한다.

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

실제 바이너리 내부에서 살펴봤을 때, 다음과 같다.

```
0069800  64 6C 6C 00 00 00 00 00 00 00 00 00 00 00 00 00
0069810  2C A4 06 00 00 00 00 00 40 01 00 00 3F 01 00 00
0069820  38 A4 06 00 38 A9 06 00 34 AE 06 00 6D 6F 7A 67
```

① Characteristics

0x000000

② TimeDateStamp

0x000000

③ MajorVersion

0x0000

④ MinorVersion

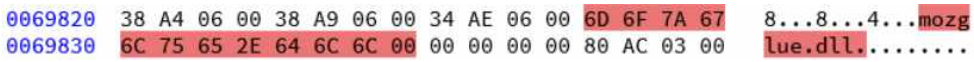
0x0000

⑤ Name

0x0006A42C -> 바이너리의 이름이 저장된 주소의 RVA 값을 나타낸다.

Raw = 0x0006A42C - 0x00064000 + 0x00063400 = 0x0006982C

다음과 같이 mozglue.dll이 저장된 것을 확인할 수 있다.



⑥ Base

0x00000000

⑦ NumberOfFunctions

0x00000140 -> Export 함수의 개수를 나타낸다.

⑧ NumberOfNames

0x0000013F -> Export 함수 중 이름이 존재하는 함수의 개수를 나타낸다. 이 값은 NumberOfFunctions보다 항상 작거나 같다.

⑨ AddressOfFunctions

0x0006A438 -> Export 함수 주소 배열의 RVA 값을 나타낸다.

⑩ AddressOfNames

0x0006A938 -> (Name이 존재하는) Export 함수의 Name 배열의 RVA 값을 나타낸다.

⑪ AddressOfNameOrdinals

0x0006AE34 -> (Name이 존재하는) Export 함수에 대한 Ordinal 배열의 RVA 값을 나타낸다.

2) Name address list 분석

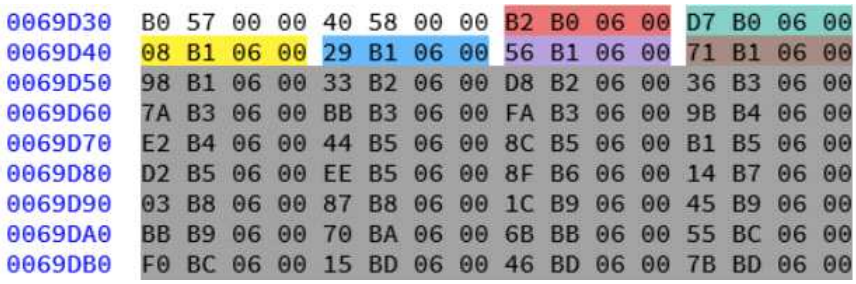
Name address list의 주소는 AddressOfNames 값을 이용하여 구할 수 있다.

Raw = 0x0006A938 - 0x00064000 + 0x00063400 = 0x00069D38

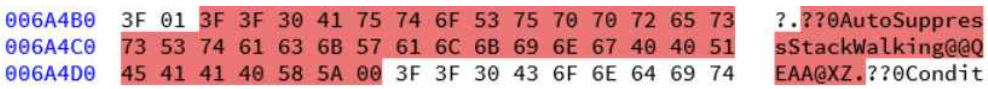
Name address list는 4byte의 주소를 나타내는 배열인데, Raw 값은 0x00069D38이며, NumberOfNames의 값이 0x13F이므로 0x00069D38 +4 * 0x13F 까지 Name address list임을 알 수 있다.

다음은 바이너리 내의 Name address list를 나타낸 것이다. (개수가 너무 많아 뒷부분

은 단색으로 나타내었다)



그중 가장 첫 번째에 위치하는 0x0006B0B2를 살펴보면, Raw로 변환했을 때의 값인 0x0006A4B2에 해당 함수의 이름이 존재하는 것을 확인할 수 있다.



해당 값은 demangle 하였을 때 **"public: __cdecl AutoSuppressStackWalking::AutoSuppressStackWalking(void) __ptr64"**을 나타낸다.

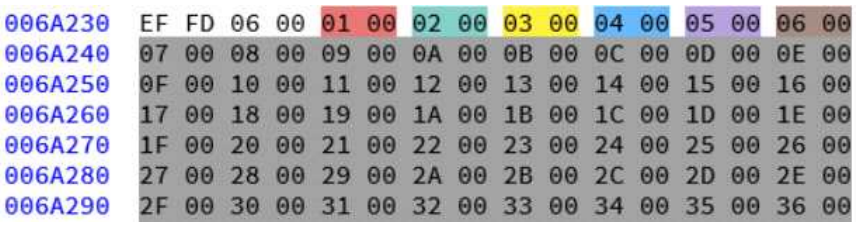
3) Ordinal list 분석

Ordinal list의 주소는 AddressOfNameOrdinals 값을 이용하여 구할 수 있다.

Raw = 0x0006AE34 - 0x00064000 + 0x00063400 = 0x0006A234

Ordinal은 WORD로 표현되며, NumberOfNames의 값인 0x013F만큼 존재한다.

다음은 바이너리 내의 존재하는 Ordinal list를 나타낸 것이다.



4) Function address list 분석

Function address list의 주소는 AddressOfFunctions 필드의 값을 이용하여 구할 수 있다.

Raw = 0x0006A438 - 0x00064000 + 0x00063400 = 0x00069838

Function의 주소는 4byte로 표현되며, NumberOfFunctions 값인 0x140개 존재한다.

다음은 바이너리 내의 Function address list를 나타낸 것이다.

```
0069830  6C 75 65 2E 64 6C 6C 00 00 00 00 00 80 AC 03 00
0069840  80 87 03 00 40 D6 03 00 40 D6 03 00 30 D7 03 00
0069850  B0 D7 03 00 50 D7 01 00 F0 D5 01 00 90 EF 01 00
0069860  40 EF 01 00 10 9B 05 00 E0 9D 05 00 20 EB 01 00
0069870  90 EA 01 00 B0 F1 01 00 60 F1 01 00 10 89 03 00
0069880  40 97 03 00 40 A2 05 00 60 E7 01 00 D0 E5 01 00
0069890  B0 E6 01 00 60 E5 01 00 40 E6 01 00 80 C0 03 00
00698A0  70 D2 01 00 60 D1 01 00 A0 DC 01 00 A0 DB 01 00
00698B0  10 DB 01 00 90 AC 03 00 90 49 00 00 E0 D7 01 00
00698C0  30 D3 01 00 D0 9B 05 00 30 9E 05 00 D0 EB 01 00
00698D0  30 D3 01 00 90 49 00 00 90 F3 01 00 F0 E7 01 00
00698E0  30 D3 01 00 60 DE 01 00 A0 C0 00 00 D0 C0 00 00
00698F0  E0 C0 00 00 E0 C0 00 00 E0 C0 00 00 40 D6 03 00
```

주소 값이 0x00000000인 0번째 Function은 제외하고, 1번째 Function을 살펴보면 다음과 같다. 1번째 Function의 RVA 값은 0x0003AC80인데, 해당 값은 .text Section에 해당한다.

Raw = 0x0003AC80 - 0x00001000 + 0x00000400 = 0x0003A080

해당 Raw 위치로 가보면 다음과 같은 opcode를 확인할 수 있는데,

```
003A080  48 89 C8 F0 48 83 05 14 9D 03 00 01 C3 CC CC CC
003A090  F0 48 83 05 07 9D 03 00 FF C3 CC CC CC CC CC CC
```

실제로 디스어셈블러를 통해 살펴보면 다음과 같이 함수의 코드임을 확인할 수 있다.

```
00007FFAA3F5BC70  48:83EC 38          sub rsp,38                                  RtlIdnToAscii
00007FFAA3F5BC74  48:8B4424 60        mov rax,qword ptr ss:[rsp+60]
00007FFAA3F5BC79  C64424 28 01        mov byte ptr ss:[rsp+28],1
00007FFAA3F5BC7E  48:894424 20        mov qword ptr ss:[rsp+20],rax
00007FFAA3F5BC83  E8 0C000000         call ntdll.7FFAA3F5BC94
00007FFAA3F5BC88  48:83C4 38          add rsp,38
00007FFAA3F5BC8C  C3                  ret
00007FFAA3F5BC8D  CC                  int3
```

5) EAT 분석

위의 내용을 바탕으로 mozglue.dll의 EAT를 분석한 내용은 다음과 같다.

Function Name	Ordinal	Address(RVA)
??0AutoSuppressStackWalking@@QEAA@XZ	0x0001	0x0003AC80
??0ConditionVariableImpl@detail@mozilla@@QEAA@XZ	0x0002	0x00038780
I??0Decimal@blink@@QEAA@AEBV01@@Z	0x0003	0x0003D640
…	…	…
strndup	0x013E	0x000057B0
wcsdup	0x013F	0x00005840