Итак, целью создания данного документа является отработка практических навыков работы со структурами и указателями в C++.
Идея заключается в своеобразном парсинге исполняемого файла (PE32+) с целью нахождения смещений необходимых для выполнения задачи.
Чтоб было интересней, парсить программа будет сама себя (т. е. свою «копию» уже загруженную в оперативную память).

Задача будет заключаться в том, чтобы найти адрес какой-либо функции из таблицы импорта и используя этот адрес, вызвать функцию.
*Доп. задача: получить адрес любой функции (не обязательно из таблицы импорта) какой-либо библиотеки и вызвать ее. В идеале получить адрес функции LoadLibrary из KERNEL32.dll, загрузить библиотеку USER32.dll и вызвать MessageBox.

Инструментарий:
для кодинга — Notepad++ и компилятор GCC
вспомогательный — CFF explorer, HxD, x64dbg.

Шаг 1.
Создадим «стандартную обертку» функции main:

```
1  #include <iostream>
2  #include <stdio.h>
3
4  int main(int argc, char* argv[]){
5      return 0;
6  }
```

Создадим репозиторий Git (я использую Git Bash):
git init
git add .
git commit -m “First commit”
Создадим новую ветку:
git checkout -b test
Добавим файл .gitignore и запишем в него test.exe:
touch .gitignore
echo "test.exe" > .gitignore
Коммитим
git add .
git commit -m “Add .gitignore”

Шаг 2.
Определяем местоположение нашей программы в оперативной памяти.
Я знаю как это сделать только с помощью ассемблера, потому буду использовать ассемблерную вставку.
Тут пришлось немного почитать про ассемблерные вставки для компилятора GCC.
Синтаксис AT&T ассемблера (который использует GCC) очень непривычен после Intel-ассемблера (у меня был небольшой опыт использования FASM).
К счастью нашлась книжка GCC-Inline-Assembly-HOWTO (добавлю в репозиторий), где все неплохо расписано. Да и кода на ассемблере нам потребуется 2-3 строки.

```cpp
int main(int argc, char* argv[]){
    long long addr;
    //ассемблерная вставка призвана определить местоположение
    //метки label (то есть самого кода) в оперативной памяти
    asm ("label:\n\t"                   //метка по сути является адресом в памяти
         "movq $label, %%rax\n\t"       //помещаем адрес метки(то есть адрес
                                        //текущей инструкции) в регистр rax
         "movq %%rax, %0\n\t"           //копируем содержимое регистра rax в первую
                                        //переданную переменную
          :"=r"(addr)                   //передаем переменную, в которую записать значение
                                        //(ключ "=r" указывает,что адрес переменной
                                        //можно поместить в любой регистр и эта переменная
                                        //используется только для записи)
         :
         :"%rax"                        //освобождаем rax. но это не точно
    );
    return 0;
}
```

Запустим компилятор с опцией -S, и посмотрим как там выглядит наша вставка.
g++ test.cpp -S test.s
Часть содержимого test.s:

```
# 15 "test.cpp" 1
    label:
    movq $label, %rax
    movq %rax, %rdx
```

Видим, что вставка добавилась почти без изменений (и что адрес будет передан в переменную через регистр rdx).

Теперь у нас есть адрес, по которому располагается инструкция нашей программы в ОП (оперативной памяти).

Далее зная некоторые особенности ОС Windows мы можем определить адрес, по которому ОС загрузила образ нашей программы в ОП. Этот адрес назовем ImageBase.

Первое что для этого нужно знать - страницы памяти в Windows выровнены по 4кб границе, а значит ImageBase будет кратен 0x1000 (4кб в шестнадцатиричной системе исчисления 4096 = 0x1000).

Второе — у исполняемых файлов (привычные нам .exe и .dll) в ОС Windows(и не только в ней, в UEFI прошивках он тоже используется) строго определенный формат, называемый PE32/PE32+ (Portable Executable) для x86/x64 архитектур соответственно. И согласно этому формату в начале файла находится сигнатура «MZ» или 0x4D5A.
Для примера, так выглядит начало нашего исполняемого файла в HEX-редакторе (HxD):

```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZђ.........яя..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ё.......@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ............Ђ...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..є..ґ.H!ё.LH!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  50 45 00 00 64 86 0F 00 31 E7 3B 60 00 6C 00 00  PE..dt..1з;`.l..
```

Из этого можно сделать вывод, что для нахождения ImageBase нам нужно «обнулить» последние 12 бит полученного адреса инструкции и проверить наличие по этому адресу сигнатуры MZ. Если сигнатура не найдена, уменьшаем адрес на 0x1000 (4кб) и снова проверяем наличие сигнатуры.

```
23      addr = addr & 0xFFFFFFFFFFFFF000;    //обнуляем младшие 12 бит для выравнивания
24      short sign;
25      while (sign != 0x5a4d) {             //0x5a4d MZ - сигнатура исполняемого файла
26          sign = *(short*)addr;
27          addr -= 0x1000;
28      }
29      QWORD image_base = addr + 0x1000;
```

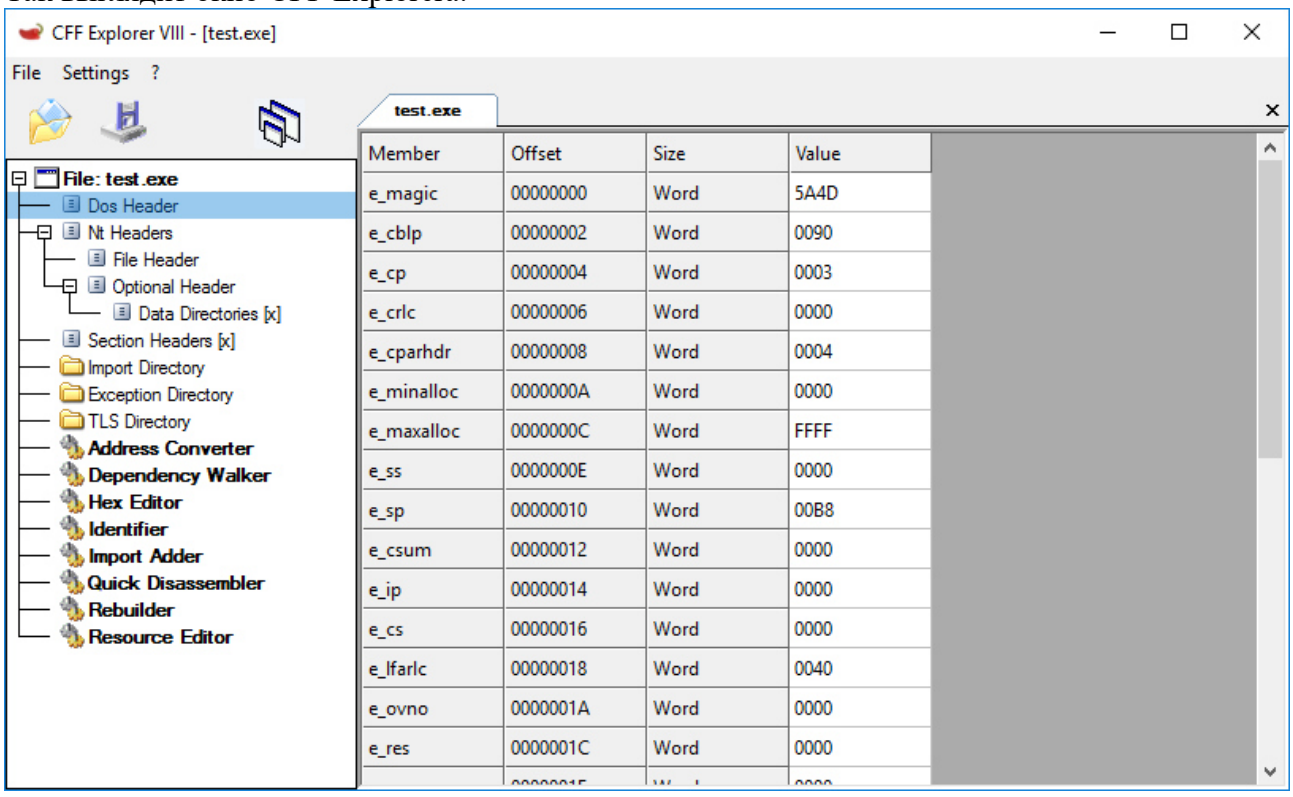
Несколько комментариев:

1. Для переменной sign использован тип данных short, т. к. сигнатура занимает 2 байта.
2. Изменен порядок байт в сигнатуре, т. к. процессор считывая данные из памяти использует порядок байт little endian, то есть от младших к старшим (в отличие от привычного нам big endian от старших к младшим)/
3. Я написал пару typedef'ов для того чтоб сократить объявление переменных.

```
4   typedef unsigned int DWORD;
5   typedef unsigned long long QWORD;
```

Закоммитим полученный результат.
Любопытный момент: я несколько раз запускал программу и каждый раз, ImageBase был равен 0x400000. Это адрес по умолчанию для всех PE файлов. Однако современные ОС используют технологию ASLR (https://ru.wikipedia.org/wiki/ASLR) и на моей ОС она включена. А значит это компилятор ее не задействует. Видимо для этого должна быть специальная опция. **Нужно уточнить у преподавателя**.

Теперь пришло время разбираться с PE форматом.
В этом очень может помочь утилита CFF Explorer (хотя мне HIEW больше нравится, но за него просят деньги) и документация (https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10)) и просто огромное количество информации в инете.
Так выглядит окно CFF Explorera:



Видны основные структуры PE файла.

Для выполнения поставленной перед собой задачи нам нужно добраться до таблицы импорта, а для этого в свою очередь пройтись по заголовкам PE файла.

Первое. DOS Header.
В нем нас интересует по смещению 0x3C поле e_lfanew размером DWORD (то есть 4 байта), в котором указано смещение до заголовка NT Header.
Чтобы прочитать из памяти значение этого смещения, нужно к ImageBase прибавить 0x3C и разыменовать полученный адрес.

```
31      addr = image_base + 0x3C;           //3C - смещение до значения смещения NT заголовка
32      DWORD nt_header_offset = *(DWORD*)addr;
33      addr = image_base + nt_header_offset;
```

Чтобы получить адрес NT заголовка, нужно к ImageBase прибавить полученное смещение.

Второе. NT Header.
У этого заголовка тоже есть своя сигнатура - «PE» или 0x50450000.

test.exe

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Текст декодирован
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZђ.....яя..
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	ё.......@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	80	00	00	00	.....Ђ...
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..є..ґ.H!ё.LH!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.....
00000080	50	45	00	00	64	86	0F	00	F7	F6	3B	60	00	6C	00	00	PE..dt..чц;`.l..
00000090	BD	04	00	00	F0	00	27	00	0B	02	02	1E	00	1E	00	00	S...p.'.....
000000A0	00	3C	00	00	00	0A	00	00	E0	14	00	00	00	10	00	00	.<......a.....
000000B0	00	00	40	00	00	00	00	00	00	10	00	00	00	02	00	00	..@.....

По смещению +0x3C находим nt_header_offset == 0x00000080

По смещению 0x00000080 видим сигнатуру заголовка NT

Проверим наличие сигнатуры.

```
35      DWORD nt_sign = *(DWORD*)addr;
36      if (nt_sign != 0x4550){             //0x50450000 == PE - сигнатура NT header
37          std::cout << "Error!";
38          return 1;
39      }
40      QWORD nt_header_addr = addr;
```

Снова не забываем про порядок байт. (0x50450000 процессор прочитает как 0x00004550). Если сигнатура не совпала — завершаем выполнение программы, т. к. либо PE файл не соответствует формату, либо мы где-то накосячили.

Третье.

Следом за сигнатуром NT заголовка (4 байта) следует файловый заголовок (File Header). В данном случае нас он мало интересует, хотя в нем можно найти информацию о количестве секций и для какой архитектуры процессора (в моем случае AMD64) предназначен данный PE файл. Нас же интересует его размер - он фиксирован и составляет 20 байт (0x14 байт), и размер опционального заголовка (он идет следующим). Размер опционального заголовка находится по смещению 0x10.

Отступив от начала NT заголовка на 0x18 байт (4 байта сигнатуры + 0x14 байт файлового заголовка) мы получим адрес опционального заголовка (Optional header). Я уж не знаю кто назвал его опциональным, но мне кажется это название не отражает и даже противоречит его сути (ИМХО). В нем содержится большая часть информации о PE файле. Здесь и информация о том, какого формата файл (PE32 или PE32+(PE64)), и адрес EntryPoint (точки входа — адрес, куда передается управление, после того, как PE loader ОС загрузит

исполняемый файл в ОП), и адрес куда загрузится секция кода, и интересовавший нас ImageBase (адрес по которому загрузить образ в ОП, по этому адресу образ и загрузится, если б не ASLR) и много еще полезного. Размер этого заголовка зависит от того, какого формата у нас файл PE32 или PE32+(PE64) и указан в файловом заголовке. Тут стоит отметить, размер опционального заголовка указан с учетом того, что в конце заголовка располагается массив структур Data Directories. Размер же опционального заголовка без учета Data Directories составляет 0x60 и 0x70 байт для форматов PE32 и PE32+ соответственно. Количество элементов массива Data Directories хоть и указано в поле NumberOfRvaAndSizes опционального заголовка, но в настоящее время в этом массиве всегда 16 (0x10) элементов. Сама структура Data Directory проста, занимает 8 байт и выглядит так:

```
struct DATA_DIRECTORY {
  DWORD VirtualAddress;
  DWORD Size;
};
```

Для нас наибольший интерес представляют 0-й и 1-й элементы массива — таблица экспорта и таблица импорта соответственно.
Сейчас нам нужна таблица импорта, а таблица экспорта понадобится когда мы будет работать с динамической библиотекой (Dll), тем более, что в нашем файле таблицы экспорта нет вовсе (в этом случае в структуре DATA_DIRECTORY будут нулевые значения).
Итак, чтобы получить адрес таблицы экспорта (то есть начала массива Data Directories) нам нужно отступить от NT заголовка на:
4 + 0x14 + 0x60 = 0x78 (4 байта сигнатуры + 0x14 байт размер файлового заголовка + 0x60 байт опционального заголовка (до Data Directories)) для формата PE32. Для формата PE32+ этот отступ будет равен 0x88 байт. Смещения до адреса таблицы импорта равны 0x78 + 8 = 0x80 и 0x88 + 8 = 0x90 байт для форматов PE32 и PE32+ соответственно.
Адреса таблиц экспорта и импорта, указанные в соответствующих полях, являются относительными (RVA адреса), т. е. рассчитаны относительно ImageBase.

```
41      char offset_to_opt_header = 0x18;   //+0x18 - смещение до OptionalHeader,
42                                          //в начале которого инфа о формате файла
43                                          //(4б сигнатуры + 0x14б sizeof(File Header))
44      short fmt = *(short *)(nt_header_addr + offset_to_opt_header);
45      //std::cout << fmt;
46      short opt_header_size;              //размер опционального заголовка без учета Data Directories
47      if (fmt == 0x10B) opt_header_size = 0x60;       //0x10B соответствует формату PE32
48      else if (fmt == 0x20B) opt_header_size = 0x70;  //0x20B соответствует формату PE32+
49      else {
50          std::cout << "Error!";
51          return 1;
52      }
53      //std::cout << opt_header_size;
54      struct DATA_DIRECTORY {
55          DWORD RVA;
56          DWORD Size;
57      };
58      DATA_DIRECTORY* data_directory = (DATA_DIRECTORY*)(nt_header_addr + offset_to_opt_header + opt_header_size);
59      //std::cout << data_directory[1].RVA;
60      QWORD import_table_addr = image_base + data_directory[1].RVA;
```

Коммитим.

Начинается самое интересное.
В документации MSDN написано, что таблица импорта начинается с массива структур IMAGE_IMPORT_DESCRIPTOR, по одному элементу массива на каждую подключаемую dll библиотеку. Последний элемент массива заполняется нулями (своеобразный признак окончания массива). Структура имеет следующий вид:

```
struct IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;
    };
    DWORD   TimeDateStamp;
    DWORD   ForwarderChain;
    DWORD   Name;
    DWORD   FirstThunk;
};
```

Наибольший интерес представляют поля OriginalFirstThunk/Characteristics, Name и FirstThunk.

Начнем с того, что проще. С имени. В этом поле хранится RVA (то есть относительный) адрес, где хранится имя соответствующей библиотеки. Тогда, чтобы получить список всех импортируемых библиотек нужно написать такой код.

```
61      struct IMAGE_IMPORT_DESCRIPTOR {
62          union {
63              DWORD   Characteristics;
64              DWORD   OriginalFirstThunk;
65          };
66          DWORD   TimeDateStamp;
67          DWORD   ForwarderChain;
68          DWORD   Name;
69          DWORD   FirstThunk;
70      };
71      //создаем и инициализируем указатель на массив структур IMAGE_IMPORT_DESCRIPTOR
72      IMAGE_IMPORT_DESCRIPTOR* img_import_desc = (IMAGE_IMPORT_DESCRIPTOR*)import_table_addr;
73      int i = 0;
74      std::cout << "Get imported Dll's names:\n";
75      //проходим по всем элементам массива, пока не встретим нулевой
76      while (img_import_desc[i].Name != 0){
77          printf("%s\n", (image_base + img_import_desc[i].Name));
78          i++;
79      }
```

Вывод программы:

```
Get imported Dll's names:
KERNEL32.dll
msvcrt.dll
libstdc++-6.dll
```

Теперь давайте остановимся на библиотеке KERNEL32.dll и попробуем получить список импортируемых из нее функций и использовать одну из них.

Для этого нам понадобятся поля OriginalFirstThunk/Characteristics и FirstThunk структуры IMAGE_IMPORT_DESCRIPTOR.

Для наглядности (да и мне так удобней) я буду использовать HIEW в качестве HEX редактора вместо HxD. В этом случае в HIEW есть переключение между RAW (физическим адресом в файле) и VA (виртуальным адресом, как если бы программа находилась в ОП) адресами. Можно конечно пользоваться и HxD, по постоянно пересчитывать RAW адреса в VA, тот еще геморрой.

Начало таблицы импорта:

```
0040E000:  50 E0 00 00-00 00 00 00-00 00 00 00-34 E9 00 00  Pр          4щ
0040E010:  68 E2 00 00-28 E1 00 00-00 00 00 00-00 00 00 00  hт  (с
0040E020:  CC E9 00 00-40 E3 00 00-40 E2 00 00-00 00 00 00  Ьщ  @у  @т
0040E030:  00 00 00 00-E8 E9 00 00-58 E4 00 00-00 00 00 00      шщ  Xф
0040E040:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
0040E050:  80 E4 00 00-00 00 00 00-98 E4 00 00-00 00 00 00  Аф      Шф
```

Выделенная область — 0-й элемент массива структур IMAGE_IMPORT_DESCRIPTOR.

Мы видим

OriginalFirstThunk/Characteristics = 0x0000E050;

Name = 0x0000E934;

FirstThunk = 0x0000E268
Перейдя по адресу 0x0000E934 увидим имя первой загружаемой dll.

```
0040E930:  00 E0 00 00-4B 45 52 4E-45 4C 33 32-2E 64 6C 6C   p  KERNEL32.dll
0040E940:  00 00 00 00-14 E0 00 00-14 E0 00 00-14 E0 00 00      ¶p  ¶p  ¶p
```

Но сейчас нас больше интересуют другие поля.
Оговорюсь сразу, что рассматривать мы будем импорт функций по имени. Существуют и другие способы, но рассматривать их утомительно (тем более что такие РЕшники еще нужно найти).
Посмотрим куда же указывают поля OriginalFirstThunk/Characteristics и FirstThunk.
OriginalFirstThunk/Characteristics (0x0000E050):

```
0040E050:  80 E4 00 00-00 00 00 00-98 E4 00 00-00 00 00 00   Аф      Шф
0040E060:  B0 E4 00 00-00 00 00 00-C4 E4 00 00-00 00 00 00   ▒ф      ─ф
0040E070:  DA E4 00 00-00 00 00 00-F0 E4 00 00-00 00 00 00   ┌ф      Ёф
0040E080:  00 E5 00 00-00 00 00 00-12 E5 00 00-00 00 00 00    x      ↕x
0040E090:  2C E5 00 00-00 00 00 00-3C E5 00 00-00 00 00 00   ,x      <x
0040E0A0:  58 E5 00 00-00 00 00 00-6C E5 00 00-00 00 00 00   Xx      lx
0040E0B0:  84 E5 00 00-00 00 00 00-9A E5 00 00-00 00 00 00   Дx      Ъx
0040E0C0:  B4 E5 00 00-00 00 00 00-CA E5 00 00-00 00 00 00   ┤x      ╩x
0040E0D0:  DE E5 00 00-00 00 00 00-F8 E5 00 00-00 00 00 00   ▐x      °x
0040E0E0:  0C E6 00 00-00 00 00 00-2A E6 00 00-00 00 00 00   ♀ц      *ц
0040E0F0:  32 E6 00 00-00 00 00 00-46 E6 00 00-00 00 00 00   2ц      Fц
0040E100:  54 E6 00 00-00 00 00 00-70 E6 00 00-00 00 00 00   Tц      pц
0040E110:  82 E6 00 00-00 00 00 00-92 E6 00 00-00 00 00 00   Вц      Тц
0040E120:  00 00 00 00-00 00 00 00-A8 E6 00 00-00 00 00 00            иц
```

FirstThunk (0x0000E268):

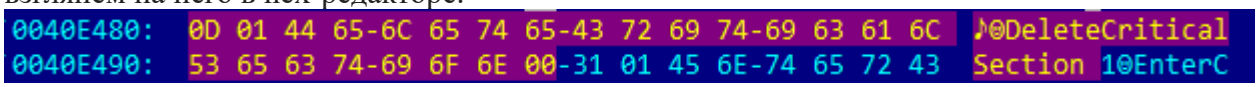
```
0040E260:  00 00 00 00-00 00 00 00-80 E4 00 00-00 00 00 00            Аф
0040E270:  98 E4 00 00-00 00 00 00-B0 E4 00 00-00 00 00 00   Шф      ▒ф
0040E280:  C4 E4 00 00-00 00 00 00-DA E4 00 00-00 00 00 00   ─ф      ┌ф
0040E290:  F0 E4 00 00-00 00 00 00-00 E5 00 00-00 00 00 00   Ёф       x
0040E2A0:  12 E5 00 00-00 00 00 00-2C E5 00 00-00 00 00 00   ↕x      ,x
0040E2B0:  3C E5 00 00-00 00 00 00-58 E5 00 00-00 00 00 00   <x      Xx
0040E2C0:  6C E5 00 00-00 00 00 00-84 E5 00 00-00 00 00 00   lx      Дx
0040E2D0:  9A E5 00 00-00 00 00 00-B4 E5 00 00-00 00 00 00   Ъx      ┤x
0040E2E0:  CA E5 00 00-00 00 00 00-DE E5 00 00-00 00 00 00   ╩x      ▐x
0040E2F0:  F8 E5 00 00-00 00 00 00-0C E6 00 00-00 00 00 00   °x      ♀ц
0040E300:  2A E6 00 00-00 00 00 00-32 E6 00 00-00 00 00 00   *ц      2ц
0040E310:  46 E6 00 00-00 00 00 00-54 E6 00 00-00 00 00 00   Fц      Тц
0040E320:  70 E6 00 00-00 00 00 00-82 E6 00 00-00 00 00 00   pц      Вц
0040E330:  92 E6 00 00-00 00 00 00-00 00 00 00-00 00 00 00   Тц
```

Видим что оба этих поля указывают на одни и те же данные, хоть и расположенные в разных участках памяти. Тут может возникнуть вопрос, а для чего так сделано? Ответ мы узнаем, когда снимем дамп из ОП с образа исполняемого файла. Пока же внимательнее приглядимся к этим данным. Можно заметить что эти данные похожи на массив, содержащий 8-ми байтные элементы (8 байт в случае PE32+, а для PE32 будут 4-х байтные), последний элемент которого равен 0 (опять как признак окончания массива). В качестве элементов похоже выступают какие-то RVA адреса. MSDN говорит что это массив указателей на структуры IMAGE_IMPORT_BY_NAME. Структура очень проста и выглядит так:

```
struct IMAGE_IMPORT_BY_NAME{
        WORD    Hint;
        BYTE    Name[?];
}
```

Сам массив вроде бы носит имя IMAGE_THUNK_DATA.
В поле Hint записано значение ординала (что такое ординал, становится понятно, после изучения структуры таблицы экспорта) для импортируемой функции. Однако даже в MSDN написано, что это значение не обязательно должно быть корректным. Так что верить этому

полю не стоит. *(а еще возникает желание попробовать написать какой-нибудь шелкодец(или воспользоваться metasploit'ом) и распихать его в эти hint'ы))), ведь даже 100 импортируемых функций дадут 200 байт места, для шеллкода хватит. Вряд ли я первый кому пришла в голову эта идея, возможно есть ограничения/проверки о которых я не знаю, но думаю попробовать надо будет)*.
Сразу за полем hint следует имя импортируемой функции.
Возьмем 0-й элемент массива IMAGE_THUNK_DATA (т .е. 0X000000000000E480) и взглянем на него в hex-редакторе:



Как и ожидалось. Поле hint = 0x01D0, в поле Name лежит нуль-терминированная строка.
Теперь можем попробовать получить список всех импортируемых функций определенной библиотеки. Мне нравится KERNEL32.dll. Для этого чуть подправим код:

```
74      IMAGE_IMPORT_DESCRIPTOR* img_import_desc = (IMAGE_IMPORT_DESCRIPTOR*)import_table_addr;
75      int i = 0;  //счетчик
76      int n = 0;  //порядковый номер элемента в массиве IMAGE_IMPORT_DESCRIPTOR,
77                  //соответствующего нужной библиотеке
78      std::cout << "Get imported Dll's names:\n";
79      char dll_name[] = "KERNEL32.dll";
80      //проходим по всем элементам массива, пока не встретим нулевой
81      while (img_import_desc[i].Name != 0){
82          printf("%s\n", (image_base + img_import_desc[i].Name));
83          //сравниваем массивы char'ов (имена библиотек)
84          if(strcmp((char*)(image_base + img_import_desc[i].Name), dll_name) == 0) n = i;
85          i++;
86      }
87      //получаем адрес(указатель) массива IMAGE_THUNK_DATA,
88      //где каждый элемент является указателем на struct{short Hint;char func_Name[]}
89      QWORD* OriginalFirstThunk_arr = (QWORD*)(image_base + img_import_desc[n].OriginalFirstThunk);
90      char* import_func_name;
91      i = 0;
92      std::cout << "\nGet imported functions names:\n";
93      while (OriginalFirstThunk_arr[i] != 0){
94          import_func_name = (char*)(image_base + OriginalFirstThunk_arr[i] + 2); //+ 2 байта (размер Hint)
95          printf("%s\n", import_func_name);
96          i++;
97      }
```

Вывод программы:

```
Get imported Dll's names:
KERNEL32.dll
msvcrt.dll
libstdc++-6.dll

Get imported functions names:
DeleteCriticalSection
EnterCriticalSection
GetCurrentProcess
GetCurrentProcessId
GetCurrentThreadId
GetLastError
GetStartupInfoA
GetSystemTimeAsFileTime
GetTickCount
InitializeCriticalSection
IsDBCSLeadByteEx
LeaveCriticalSection
MultiByteToWideChar
QueryPerformanceCounter
RtlAddFunctionTable
RtlCaptureContext
RtlLookupFunctionEntry
RtlVirtualUnwind
SetUnhandledExceptionFilter
Sleep
TerminateProcess
TlsGetValue
UnhandledExceptionFilter
VirtualProtect
VirtualQuery
WideCharToMultiByte
```

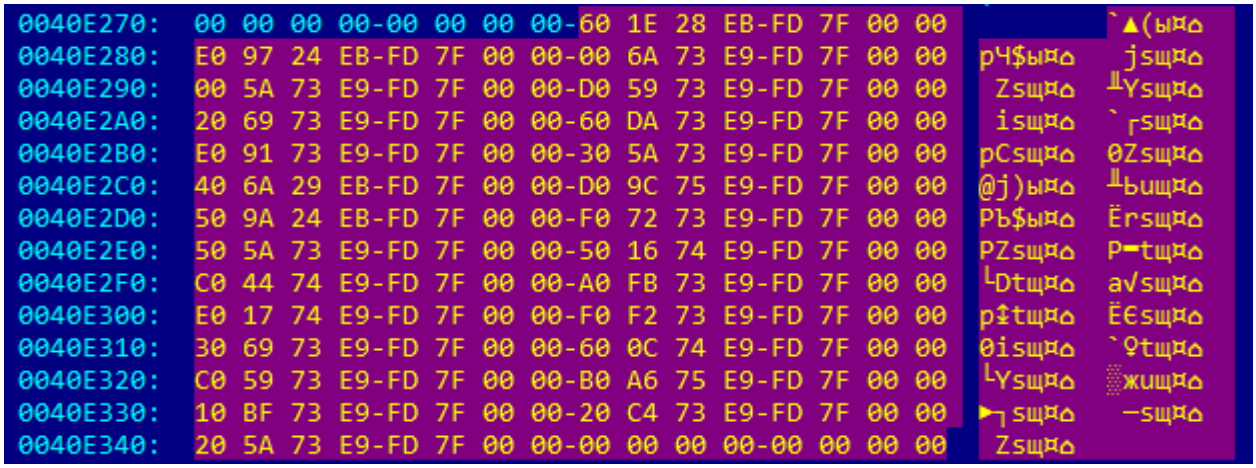
Коммитим.

Так. Для решения первой поставленной задачи остается только ответить на вопрос «Для чего же используется 2 одинаковых массива?» получить адрес какой-нибудь функции и вызвать ее. Для ответа на вопрос нужно снять образ файла, когда он будет загружен в оперативную память. Можно к примеру воспользоваться отладчиком x64dbg или же Process Hacker'ом.
Я снял дамп только таблицы импорта, вот ее начало:

```
0040E000:  50 E0 00 00-00 00 00 00-00 00 00 00-68 E9 00 00  Pp          hщ
0040E010:  78 E2 00 00-28 E1 00 00-00 00 00 00-00 00 00 00  xт  (c
```

Замечание: по сравнению с прошлыми скринами есть изменения, вызванные добавлением кода.
Выделенная область — 0-й элемент массива структур IMAGE_IMPORT_DESCRIPTOR.
Мы видим
OriginalFirstThunk/Characteristics = 0x0000E050;
FirstThunk = 0x0000E278;
Как мы помним это были указатели на два одинаковых массива.
Взглянем на них теперь.
OriginalFirstThunk/Characteristics:

То есть этот массив не изменился.
FirstThunk:



А этот массив заполнился другими адресами. Эти адреса PE loader операционной системы заменил действительными адресами функций. То есть PE loader загрузил в ОП необходимую библиотеку, нашел в ней нужные нам функции и записал их адреса в этот массив.
Теперь, чтобы получить адрес определенной функции нужно пройти по элементам массива IMAGE_THUNK_DATA, проверить имя функции и в случае, если это интересующая нас функция, то запомнить индекс элемента массива. По этому индексу взять адрес из массива адресов.
Немного изменим код:

```
89      QWORD* OriginalFirstThunk_arr = (QWORD*)(image_base + img_import_desc[n].OriginalFirstThunk);
90      QWORD *FirstThunk_arr = (QWORD *)(image_base + img_import_desc[n].FirstThunk);
91      char* import_func_name;
92      i = 0;
93      n = 0;
94      std::cout << "\nGet imported functions names:\n";
95      char our_func_name[] = "GetCurrentProcessId";
96      while (OriginalFirstThunk_arr[i] != 0){
97          import_func_name = (char*)(image_base + OriginalFirstThunk_arr[i] + 2); //+ 2 байта (размер Hint)
98          printf("%s\n", import_func_name);
99          if(strcmp(import_func_name, our_func_name) == 0) n = i;
100         i++;
101     }
102     QWORD our_func_addr = FirstThunk_arr[n];
```

Добавились стоки с получением указателя на массив содержащий адреса функций (FirstThunk_arr), сравнение имени функции и запоминание соответствующего индекса, а также непосредственно получение адреса.
Можно заметить, что искал я функцию «GetCurrentProcessId».
И теперь нам известен ее адрес.
Я не знал способа, как зная адрес функции запустить ее средствами языка C++. Однако немного погуглив решение обнаружилось.

Нужно объявить указатель на функцию, а потом присвоить указателю адрес нужной функции. Функция GetCurrentProcessId не имеет параметров и возвращает значение типа DWORD, тогда для нее нужно объявить такой указатель:
DWORD (*name)();
Скобки вокруг *name необходимы иначе запись DWORD* name(); будет являться просто объявлением функции name без параметров и возвращающая значение типа DWORD*.
Дальше нужно только присвоить указателю нужный адрес, приведя к нужному типу.
Теперь код выглядит так:

```
102     QWORD our_func_addr = FirstThunk_arr[n];
103     DWORD (*get_process_id)();                      //объявляем указатель на функцию
104     get_process_id = (DWORD(*)())our_func_addr;     //приводим адрес к нужному типу
105     DWORD process_id = get_process_id();            //и вызываем функцию
106     std::cout << process_id;
107     getch();
108     return 0;
```

Добавил также заголовочный файл <conio.h> и функцию getch(), чтобы программа не завершалась сразу.
Закоммитил.
Основная задача выполнена.

Доп. задача
coming soon...