КОДИНГ

Тайный WinAPI. Как обфусцировать вызовы WinAPI в своем приложении

Nik Zerof , 06.12.2018

Образцы серьезной малвари и вымогателей часто содержат интересные методики заражения, скрытия активности и нестандартные отладочные приемы. В вирусах типа Potato или вымогателях вроде SynAsk используется простая, но мощная техника скрытия вызовов WinAPI. Об этом мы и поговорим, а заодно напишем рабочий пример скрытия WinAPI в приложении.

Итак, есть несколько способов скрытия вызовов WinAPI.

1. Виртуализация. Важный код скрывается внутри самодельной виртуальной машины.
2. Прыжок в тело функции WinAPI после ее пролога. Для этого нужен дизассемблер длин инструкций.
3. Вызов функций по их хеш-значениям.

Все остальные техники — это разные вариации или развитие трех этих атак. Первые две встречаются нечасто — слишком громоздкие. Как минимум приходится всюду таскать с собой дизассемблер длин и прологи функций, рассчитанные на две разные архитектуры. Вызов функций по хеш-именам прост и часто используется в более-менее видной малвари (даже кибершпионской).

Наша задача — написать легко масштабируемый мотор для реализации скрытия вызовов WinAPI. Они не должны читаться в таблице импорта и не должны бросаться в глаза в дизассемблере. Давай напишем короткую программу для экспериментов и откомпилируем ее для x64.

```
#include <Windows.h>

int main() {
  HANDLE hFile = CreateFileA("C:\\test\\text.txt",
    GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL);
  Sleep(5000);
  return 0;
}
```

Как видишь, здесь используются две функции WinAPI — `CreateFileA` и `Sleep`. Функцию `CreateFileA` я решил привести в качестве примера не случайно — по ее аргументу `"C:\\test\\text.txt"` мы ее легко и найдем в уже обфусцированном виде.

Давай глянем на дизассемблированный код этого приложения. Чтобы листинг на ASM был выразительнее, программу необходимо откомпилировать, избавившись от всего лишнего в коде. Откажемся от некоторых проверок безопасности и библиотеки CRT. Для оптимизации приложения необходимо выполнить следующие настройки компилятора:

- предпочитать краткость кода (`/Os`),
- отключить проверку безопасности (`/Gs-`),
- отключить отладочную информацию,
- в настройках компоновщика отключить внесение случайности в базовый адрес (`/DYNAMICBASE:NO`),
- включить фиксированный базовый адрес (`/FIXED`),
- обозначить самостоятельно точку входа (в нашем случае это main),
- игнорировать все стандартные библиотеки (`/NODEFAULTLIB`),
- отказаться от манифеста (`/MANIFEST:NO`).

Эти действия помогут уменьшить размер программы и избавить ее от вставок неявного кода. В моем случае получилось, что программа занимает 3 Кбайт. Ниже — ее полный листинг.

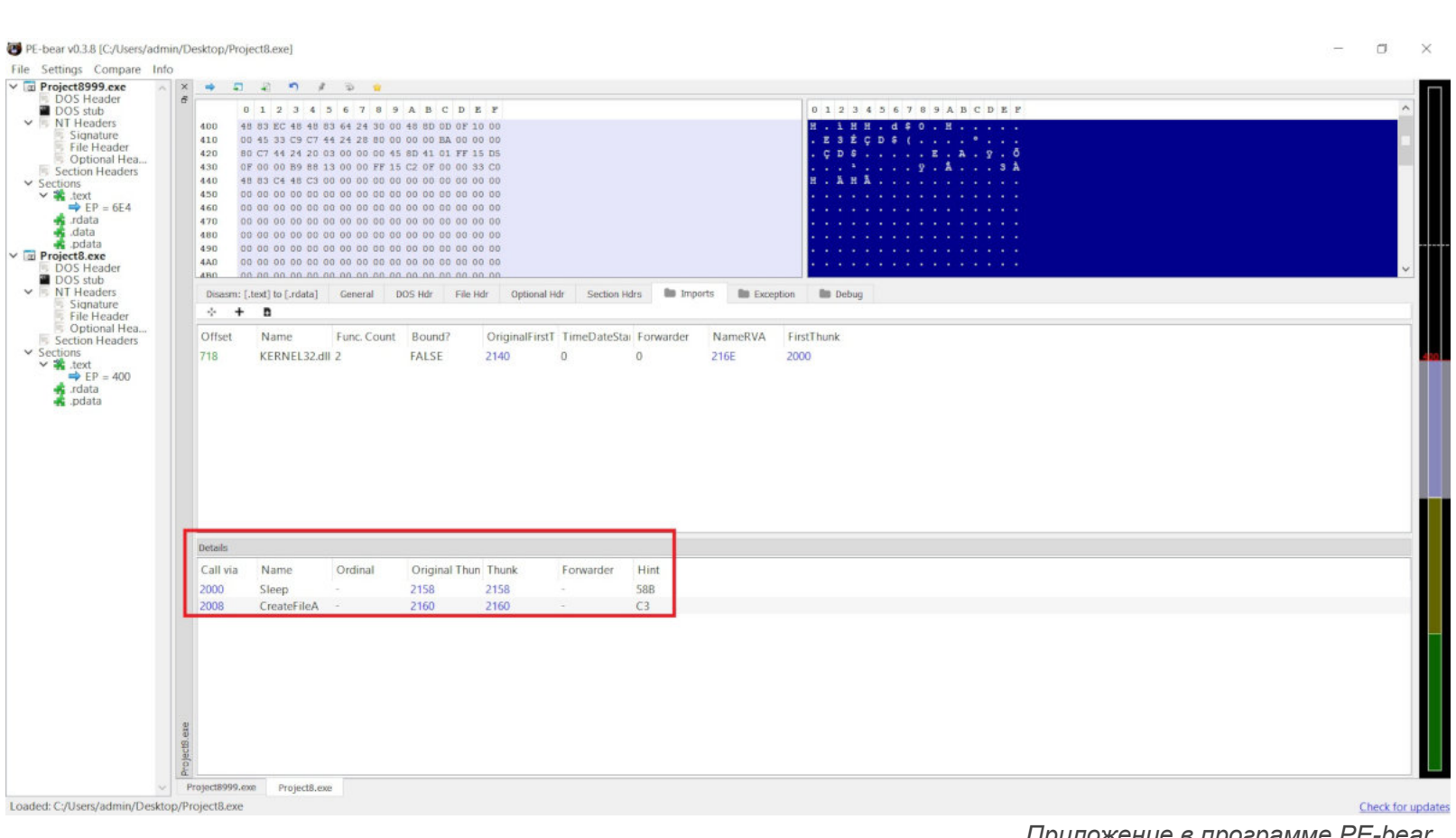
```
public start
start proc near

dwCreationDisposition= dword ptr -28h
dwFlagsAndAttributes= dword ptr -20h
var_18= qword ptr -18h

sub   rsp, 48h
and   [rsp+48h+var_18], 0
lea   rcx, FileName        ; "C:\\test\\text.txt"
xor   r9d, r9d             ; lpSecurityAttributes
mov   [rsp+48h+dwFlagsAndAttributes], 80h ; dwFlagsAndAttributes
mov   edx, 80000000h       ; dwDesiredAccess
mov   [rsp+48h+dwCreationDisposition], 3 ; dwCreationDisposition
lea   r8d, [r9+1]          ; dwShareMode
call  cs:CreateFileA
mov   ecx, 1388h           ; dwMilliseconds
call  cs:Sleep
xor   eax, eax
add   rsp, 48h
retn

start endp
```

Как видишь, функции WinAPI явно читаются в коде и видны в таблице импорта приложения.



Приложение в программе PE-bear

Теперь давай создадим модуль, который поможет скрывать от любопытных глаз используемые нами функции WinAPI. Напишем таблицу хешей функций.

```
static DWORD hash_api_table[] = {
  0xe976c80c, // CreateFileA
  0xb233e4a5, // Sleep
}
```

Как хешировать

В статье нет смысла приводить алгоритм хеширования — их десятки, и они доступны в Сети, даже в Википедии. Могу посоветовать алгоритмы, с возможностью выставления вектора начальной инициализации (seed), чтобы хеши функций были уникальными. Например, подойдет алгоритм MurmurHash.

Давай условимся, что у нас макрос хеширования будет иметь прототип `HASH_API(name, name_len, seed)`, где `name` — имя функции, `name_len` — длина имени, `seed` — вектор начальной инициализации. Так что все значения хеш-функций у тебя будут другими, не как в статье!

Поскольку мы договорились писать легко масштабируемый модуль, определимся, что функция получения WinAPI у нас будет вида

```
LPVOID get_api(DWORD api_hash, LPCSTR module);
```

Но до этого еще нужно дойти, а сейчас напишем универсальную функцию, которая будет разбирать экспортируемые функции WinAPI передаваемой в нее системной библиотеки.

```
LPVOID parse_export_table(HMODULE module, DWORD api_hash) {
  PIMAGE_DOS_HEADER     img_dos_header;
  PIMAGE_NT_HEADERS     img_nt_header;
  PIMAGE_EXPORT_DIRECTORY     in_export;

  img_dos_header  = (PIMAGE_DOS_HEADER)module;
  img_nt_header = (PIMAGE_NT_HEADERS)((DWORD_PTR)img_dos_header + img_dos_header->e_lfar
  in_export = (PIMAGE_EXPORT_DIRECTORY)((DWORD_PTR)img_dos_header + img_nt_header->Optic
```

По ходу написания этой функции я буду пояснять, что к чему, потому что путешествие по заголовку PE-файла — дело непростое (у динамической библиотеки будет именно такой заголовок). Сначала мы объявили используемые переменные, с этим не должно было возникнуть проблем. 🙂 Далее, в первой строчке кода, мы получаем из переданного в нашу функцию модуля DLL ее `IMAGE_DOS_HEADER`. Вот его структура:

```
typedef struct _IMAGE_DOS_HEADER {
  WORD e_magic;
  WORD e_cblp;
  WORD e_cp;
  WORD e_crlc;
  WORD e_cparhdr;
  WORD e_minalloc;
  WORD e_maxalloc;
  WORD e_ss;
  WORD e_sp;
  WORD e_csum;
  WORD e_ip;
  WORD e_cs;
  WORD e_lfarlc;
  WORD e_ovno;
  WORD e_res[4];
  WORD e_oemid;
  WORD e_oeminfo;
  WORD e_res2[10];
  LONG e_lfanew;
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Здесь нас интересует поле `e_lfanew` — это RVA (Relative Virtual Address, смещение) до заголовка `IMAGE_NT_HEADERS`, который, в свою очередь, имеет такую структуру:

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

Нужное нам поле `OptionalHeader` указывает на еще одну структуру — `IMAGE_OPTIONAL_HEADER`. Она громоздкая, и я ее сократил до нужных нам полей, точнее до элемента `DataDirectory`, который содержит 16 полей. Нужное нам поле называется `IMAGE_DIRECTORY_ENTRY_EXPORT`. Оно описывает символы экспорта, а поле `VirtualAddress` указывает смещение секции экспорта.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  ...
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Итак, мы в секции экспорта, в `IMAGE_EXPORT_DIRECTORY`. Продолжаем ее читать:

```
PDWORD rva_name;
UINT rva_ordinal;
rva_name = (PDWORD)((DWORD_PTR)img_dos_header + in_export->AddressOfNames);
rva_ordinal = (PWORD)((DWORD_PTR)img_dos_header + in_export->AddressOfNameOrdinals);
```

Чтобы было понятнее, структура `IMAGE_EXPORT_DIRECTORY`:

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

Наконец-то мы пробрались сквозь дебри заголовка PE к нужным нам данным. Остальное — дело техники. Как ты уже понял по коду, здесь нас интересуют два поля: `AddressOfNames` и `AddressOfNameOrdinals`. Первое содержит имена функций, второе — их индекс (читай: номер). Суть дальнейших действий проста: в цикле будем просматривать и сверять переданный в нашу функцию хеш с хешами функций в таблице экспорта и, как найдем совпадение, выходим из цикла.

```
UINT ord = -1;
for (i = 0; i < in_export->NumberOfNames; i++) {
  api_name = (PCHAR)((DWORD_PTR)img_dos_header + rva_name[i]);
  get_hash = HASH_API(api_name, name_len, seed);
  if (api_hash == get_hash) {
    ord = (UINT)rva_ordinal[i];
    break;
  }
}
```

Нашли! Теперь получаем ее адрес и возвращаем его:

```
  func_addr = (PDWORD)((DWORD_PTR)img_dos_header + in_export->AddressOfFunctions);
  func_find = (LPVOID)((DWORD_PTR)img_dos_header + func_addr[ord]);
  return func_find;
}
```

INFO

В коде отсутствуют проверки корректности обрабатываемых и поступающих в функции данных. Это сделано умышленно, чтобы не засорять код и не отвлекать читателя от сути статьи.

Функция получилась весьма короткая и понятная. Теперь перейдем к написанию основной функции. Помнишь, мы ее обозначили как `LPVOID get_api`? Она будет, по сути, оберткой над `parse_export_table`, но сделает ее универсальной.

Дело в том, что наша функция `parse_export_table` слишком «сырая» — она просматривает таблицы импортов передаваемых в нее библиотек, но не читает эти библиотеки в память (если их там нет). Для этого мы используем функцию `LoadLibrary`, точнее ее хешированный вариант. Заодно посмотрим на работоспособность `parse_export_table`. 🙂

Функция экспортируется библиотекой `Kernel32.dll`. Чтобы начать с ней работать, мы должны найти эту библиотеку в адресном пространстве нашего процесса через PEB. Я буду писать сразу универсальный код, который подойдет для обеих архитектур.

```
LPVOID get_api(DWORD api_hash, LPCSTR module) {
  HMODULE krnl32, hDll;
  LPVOID  api_func;

  #ifdef _WIN64
    int ModuleList = 0x18;
    int ModuleListFlink = 0x18;
    int KernelBaseAddr = 0x10;
    INT_PTR peb = __readgsqword(0x60);
  #else
    int ModuleList = 0x0C;
    int ModuleListFlink = 0x10;
    int KernelBaseAddr = 0x10;
    INT_PTR peb = __readfsdword(0x30);
  #endif

  // Теперь получим адрес kernel32.dll

  INT_PTR mod_list    = *(INT_PTR*)(peb + ModuleList);
  INT_PTR list_flink  = *(INT_PTR*)(mod_list + ModuleListFlink);
  LDR_MODULE *ldr_mod = (LDR_MODULE*)list_flink;

  for (; list_flink != (INT_PTR)ldr_mod ;) {
    ldr_mod = (LDR_MODULE*)ldr_mod->e[0].Flink;
    if (!lstrcmpiW(ldr_mod->dllname.Buffer, L"kernel32.dll"))
      break;
  }

  krnl32 = (HMODULE)ldr_mod->base;
```

Далее нам необходимо объявить прототип нашей функции `LoadLibraryA`. Это нужно сделать в начале файла. Вот прототип:

```
HMODULE (WINAPI *temp_LoadLibraryA)(__in LPCSTR file_name) = NULL;
HMODULE hash_LoadLibraryA(__in LPCSTR file_name) {
  return temp_LoadLibraryA(file_name);
}
```

Кроме того, объявим прототипы наших функций из тестового приложения, которое мы писали в самом начале:

```
HANDLE (WINAPI *temp_CreateFileA)(__in LPCSTR file_name,
  __in DWORD access,
  __in DWORD share,
  __inopt LPSECURITY_ATTRIBUTES security,
  __in DWORD creation_disposition,
  __in DWORD flags,
  __inopt HANDLE template_file) = NULL;

HANDLE hash_CreateFileA(__in    LPCSTR      file_name,
  __in    DWORD     access,
  __in    DWORD     share_mode,
  __inopt           LPSECURITY_ATTRIBUTES security,
  __in    DWORD     creation_disposition,
  __in    DWORD     flags,
  __inopt HANDLE    template_file) {
  temp_CreateFileA = (HANDLE (WINAPI *)(LPCSTR,
    DWORD,
    DWORD,
    LPSECURITY_ATTRIBUTES,
    DWORD,
    DWORD,
    HANDLE))get_api(hash_api_table[0], "Kernel32.dll");
  return temp_CreateFileA(file_name, access, share_mode, security, creation_disposition,
}

VOID (WINAPI *temp_Sleep)(DWORD time) = NULL;
VOID hash_Sleep(__in DWORD time) {
  temp_Sleep = (VOID (WINAPI *)(DWORD))get_api(hash_api_table[1], "Kernel32.dll");
  return temp_Sleep(time);
}
```

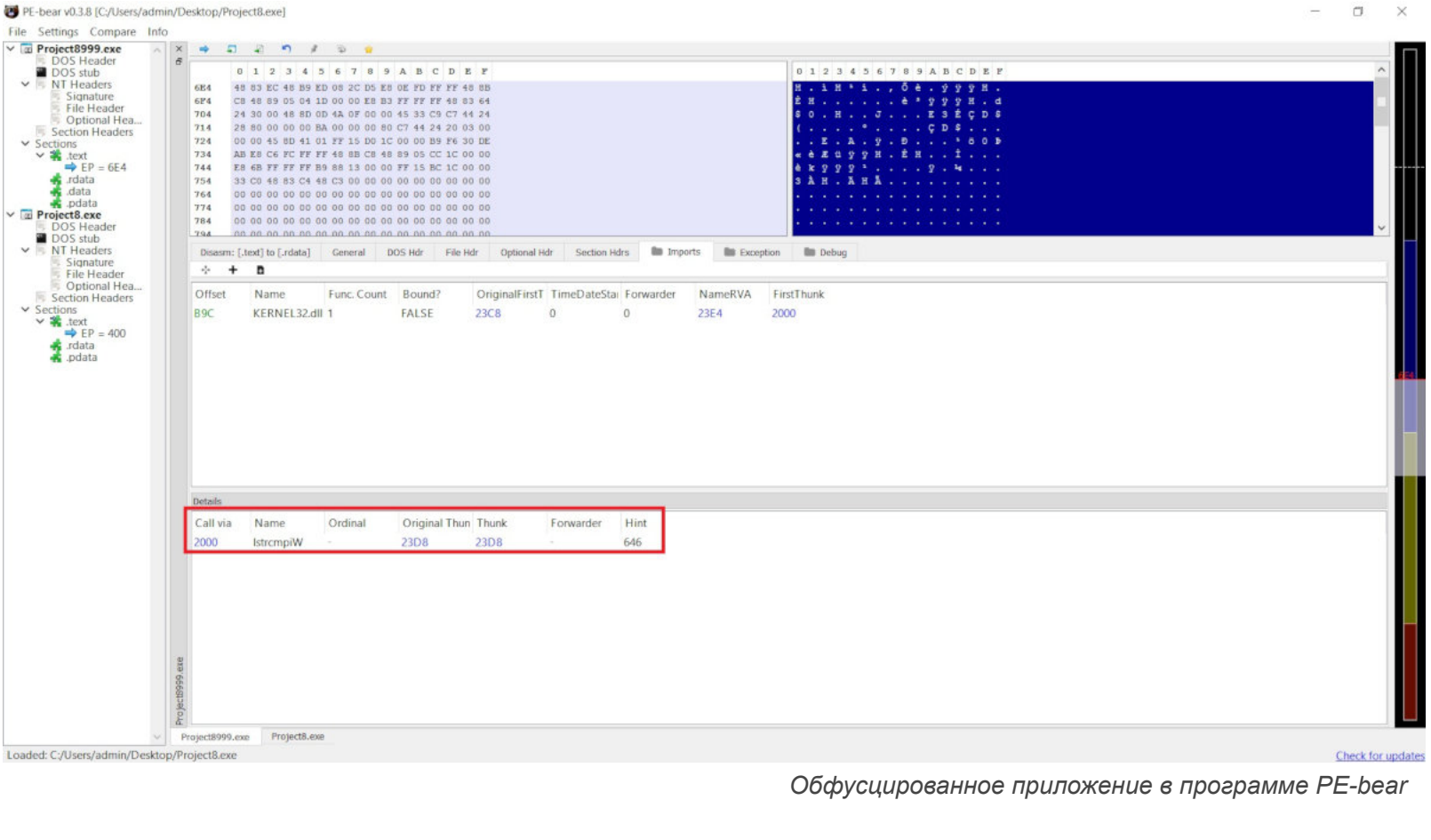
Прототип для `LoadLibraryA` — упрощенный. Мы здесь не используем нашу таблицу хешей `hash_api_table[]`, потому что хеш `LoadLibraryA` мы захардкодим дальше. Хеш будет у каждого свой, в зависимости от алгоритма хеширования.

```
  temp_LoadLibraryA = (HMODULE (WINAPI *)(LPCSTR))parse_export_table(krnl32, 0x731faae5)
  hDll = hash_LoadLibraryA(module);
  api_func = (LPVOID)parse_export_table(hDll, api_hash);
  return api_func;
}
```

Итак, все готово. Этот мотор для вызова функций по хешу можно вынести в отдельный файл и расширять, добавляя новые прототипы и хеши. Теперь, после всех манипуляций, изменим наш тестовый файл и откомпилируем его.

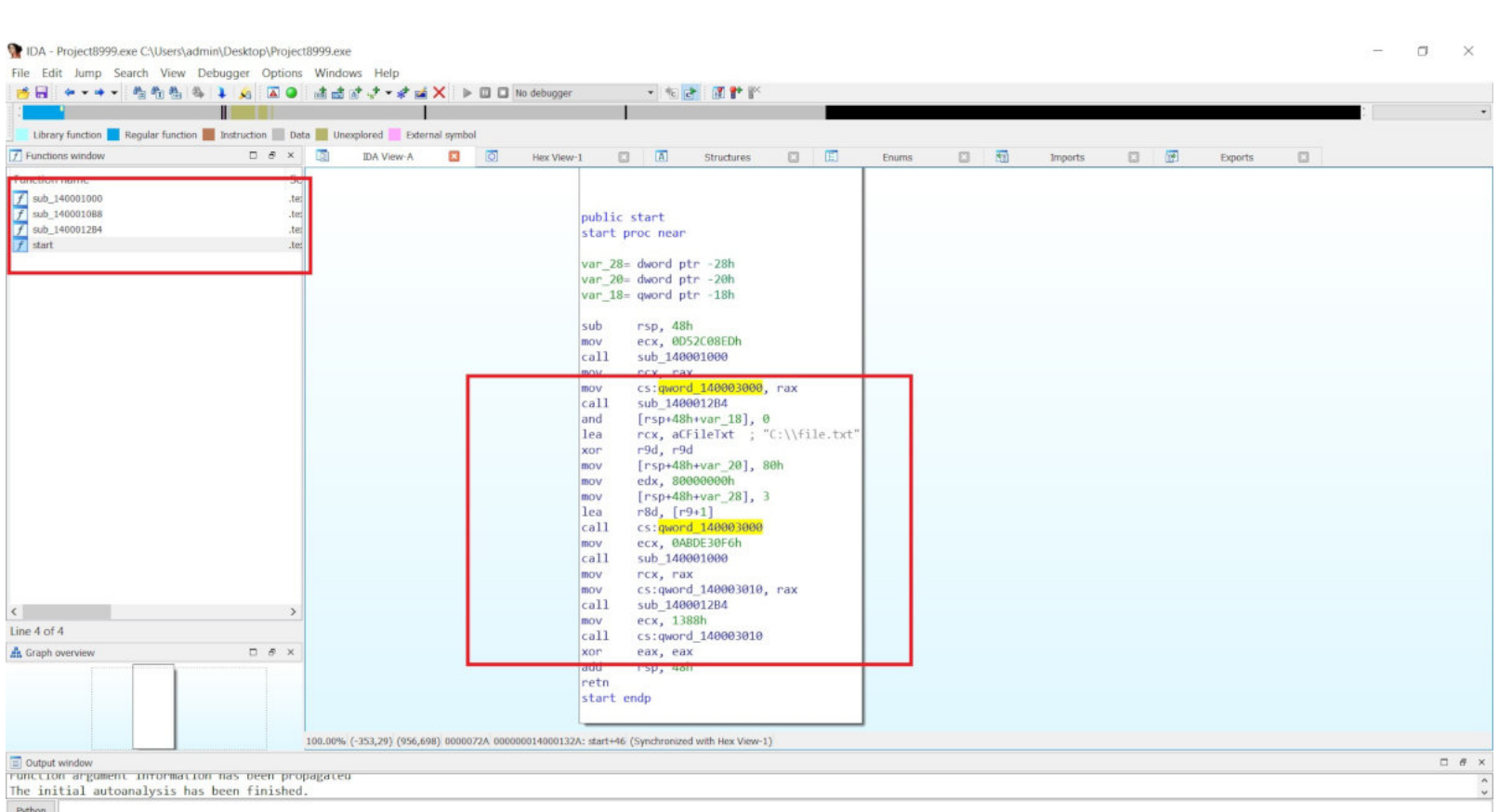
```
int main() {
  HANDLE hFile = hash_CreateFileA("C:\\test\\text.txt",
    GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL);
  hash_Sleep(5000);
  return 0;
}
```

Первое, что бросается в глаза, — наш файл работает! 🙂 Текстовый файл создается, программа засыпает на пять секунд и закрывается. Теперь давай посмотрим на таблицу импорта...



Обфусцированное приложение в программе PE-bear

...и увидим там только функцию `lstrcmpiW`. Ты ведь помнишь, что мы ее использовали для сравнения строк? Больше никаких функций нет! Теперь заглянем в дизассемблер.



Обфусцированное приложение в программе IDA

Здесь мы тоже не видим никаких вызовов. Если углубиться в исследование программы, мы, разумеется, обнаружим наши хеши, строки типа `kernel32.dll` и прочее. Но это просто учебный пример, демонстрирующий базу, которую можно развивать.

Хеши можно защитить различными математическими операциями, а строки зашифровать. Для закрепления знаний попробуй скрыть функцию `lstrcmpiW` по аналогии с другими WinAPI. Даю подсказку: эта функция экспортируется библиотекой `kernel32.dll`. 🙂