Институт Sans

Читальный зал библиотеки безопасности

Основы реверсивной инженерии с отладчиком Immunity

Автор: Роберто Нарделла
Перевод: Lemma Works Studio

Оглавление

Основы реверсивной инженерии с отладчиком Immunity

Отступление

Реверсивная инженерия – достаточно интригующее искусство, но в то же время – это одна из самых сложных тем в области безопасности и анализа вредоносных программ. Опытные инженеры имеют глубочайшие знания языка программирования Ассемблер, архитектур процессоров и хорошо знакомы с наиболее важными отладчиками. Тем не менее, есть множество информации, которая может быть собрана и изучена людьми с базовым уровнем знаний об отладчиках и Ассемблере. В этой статье я опишу некоторые очень простые, но полезные моменты реверсивной инженерии, выполняемые с помощью изумительного отладчика Immunity.

Вступление

Целью данного документа не является углубление в технические особенности языка программирования Ассемблер (*далее – просто Ассемблер, прим. переводчика*), хотя это и необходимо для углублённой работы с реверсивной инженерией. В этой же статье, я хочу раскрыть основы и простые действия, выполняемые с помощью отладчика Immunity (*далее – просто Immunity, прим. переводчика*). Эти действия могут дать отклик в виде множества полезных результатов одновременно. Чтобы хорошо понять информацию, содержащуюся в этом документе, необходимо иметь базовые знания языка программирования C для Windows (WinAPI).

Примеры фрагментов кода, которые я привожу в этой статье, будут максимально приближены к «реальным» фрагментам кода. Они помогут полноценно понять, какие действия выполняет вредоносное ПО, к примеру: подгрузка файлов из сети прямо с целевой машины, переименование расширений, запись файлов на диск и добавление определённых механизмов сохранения посредством изменения или создания ключей реестра.

Основные моменты

Все исходные коды, показанные в этой работе, были скомпилированы в бесплатном компиляторе «Pelles C». Те параметры, которые были использованы для компиляции исполняемых файлов, вы можете изучить в Приложении A. Весь исходный код компилировался в 32-разрядной среде (x86).

Несмотря на то, что цель этого документа – раскрыть потенциал и возможности базовой отладки и здесь не будут рассматриваться технические особенности Ассемблера, нам не удастся избежать описания основных регистров процессора Intel. Чтобы свести к минимуму трудность чтения и восприятия этой статьи, в следующих двух таблицах будет кратко расписано назначение основных регистров общего назначения процессора x8086 и индексных регистров. Таблицу, которая расписывает сегментированные регистры, я намеренно опускаю, так как подобные регистры не встретятся нам в примерах этой статьи. Буква «E» в начале акронимов (EAX, EBX и так далее) означает «Расширенная (Extended)» или «Улучшенная (Enhanced)».

Прим. переводчика: таблицы приведу страницей ниже, чтобы не рвать их на части.

Таблица 1: Регистры общего назначения	
Название	Описание
EAX	Накопительный Регистр. Этот регистр обычно используется для хранения временных данных (таких, как отклик функции) или для хранения значений, используемых в математических операциях.
EBX	Базовый регистр. У него нет каких-то конкретных целей: он используется как для хранения временных данных, так и для индексной адресации.
ECX	Регистр-счётчик. Он используется как счётчик циклов.
EDX	Регистр данных. Используется для операций ввода-вывода и как дополнительный регистр общего назначения.

Таблица 2: Индексные регистры	
Название	Описание
ESI	Исходный индекс. Этот регистр используется для операций, проводимых с массивами и строками, в основном в «режиме чтения».
EDI	Индекс назначения. То же, что и ESI, но преимущественно для «режима записи».
EIP	«Инструктор», указатель инструкций. Это регистр, используемый только для чтения, содержащий адрес инструкции, которая должна быть выполнена далее.
EBP	Базовый указатель. Содержит переменные параметров, передаваемый в подпрограмму. Также используется для передачи аргументов в структуры данных.
ESP	Указатель стека. Содержит адрес вершины стека памяти.

Окружение отладчика Immunity

Immunity Debugger – это замечательный и бесплатный отладчик. Основы работы с ним расписаны в понятной и полезной статье Игоря Новковича. Хотя следующие данные уже упоминались в его статье, чтобы не нагружать вас лишними действиями, я напомню, что представляют собой четыре основные панели отладчика Immunity и какие данные содержатся в них после открытия исполняемого файла или присоединения процесса.

Прим. переводчика: оригиналы названия статей я вынес справа от текста в виде примечаний, и добавил гиперссылки – так удобнее, чем постоянно мотаться в конец документа.

На рисунке 1 показан интерфейс отладчика, разбитый на нумерованные панели для наилучшего понимания.

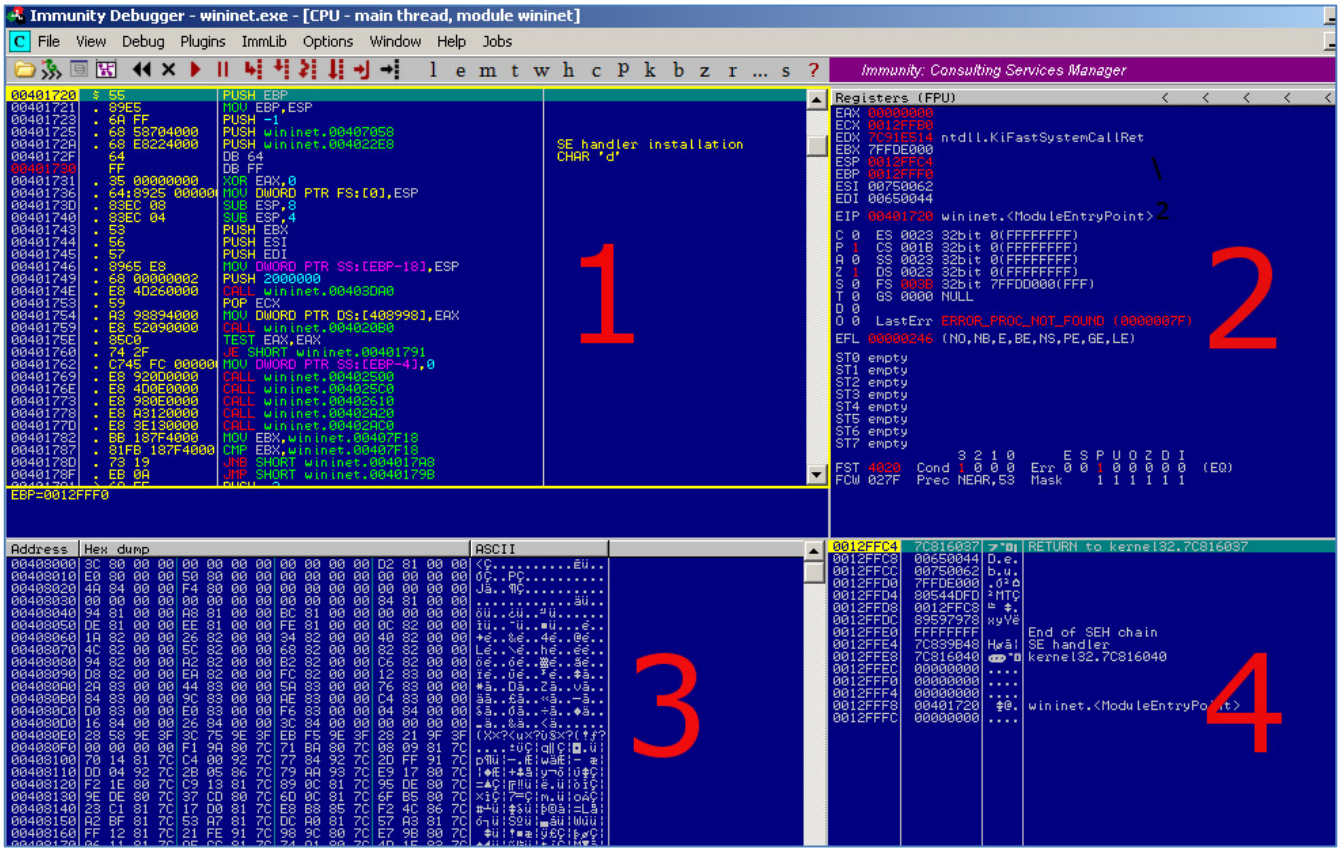


Рисунок 1. Панели Immunity

- Панель 1: на данной панели будут отображены содержать инструкции по сборке (третий столбец), исходный машинный код (второй столбец) и смещение для каждой инструкции (первый столбец). В последнем столбце (четвёртом слева) будут выводиться комментарии, добавленные либо самим отладчиком, либо же пользователем во время анализа;

- Панель 2: эта панель содержит регистры центрального процессора. Некоторые регистры, которые будут упоминаться в примерах ниже я уже кратко описал в предыдущих таблицах.
- Панель 3: здесь мы можем увидеть шестнадцатеричный дамп анализируемого исполняемого файла.
- Панель 4: панель содержит обзор стека памяти (смещения и содержимое). Во время анализа исполняемого файла или процесса, данные на этой панели изменяются динамически, к примеру, когда новые элементы помещаются в стек либо удаляются из него.

Под основным меню Immunity расположен ряд значков. Непосредственное внимание я уделю лишь трём из них, несущих основной функционал (см. рисунок 2):

- Display graph
- Step into
- Step over

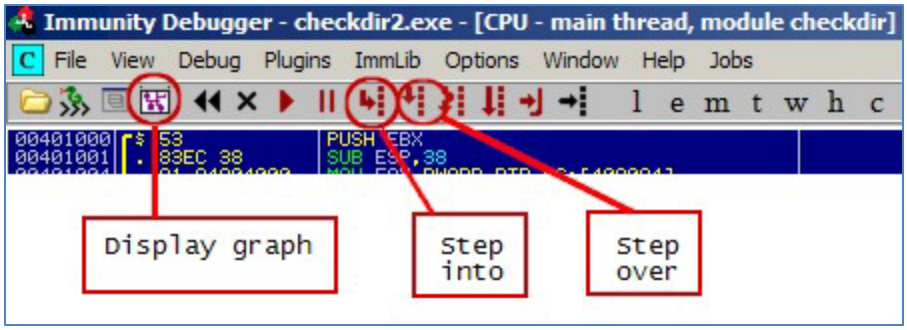


Рисунок 2. Функциональные значки

Первый значок (Display Graph) отрисует график выполнения программы, как показано на рисунке 3. Каждая программа имеет свою логику, которая, опираясь на определённое условие (новое значение, присвоенное переменной, новое событие, получение результата сравнения и так далее), может принять решение выполнить один блок инструкций вместо другого. Чтобы упростить отслеживание этих потоков данных и используется данный график. В отличие от других отладчиков, блок-схема, выводимая Immunity, отображает не цельный поток программы, а поток на более ограниченном уровне рекурсии.

Другими словами, как мы видим в примере 2, выбор какой-либо функции точкой отсчёта может быть полезен для генерации графика потока для изучения. На рисунке 3 мы можем увидеть отображение таких инструкций «JUMP» для Ассемблера, как «JMP» (Jump), «JE» (Jump if equal), «JA» (Jump if above):

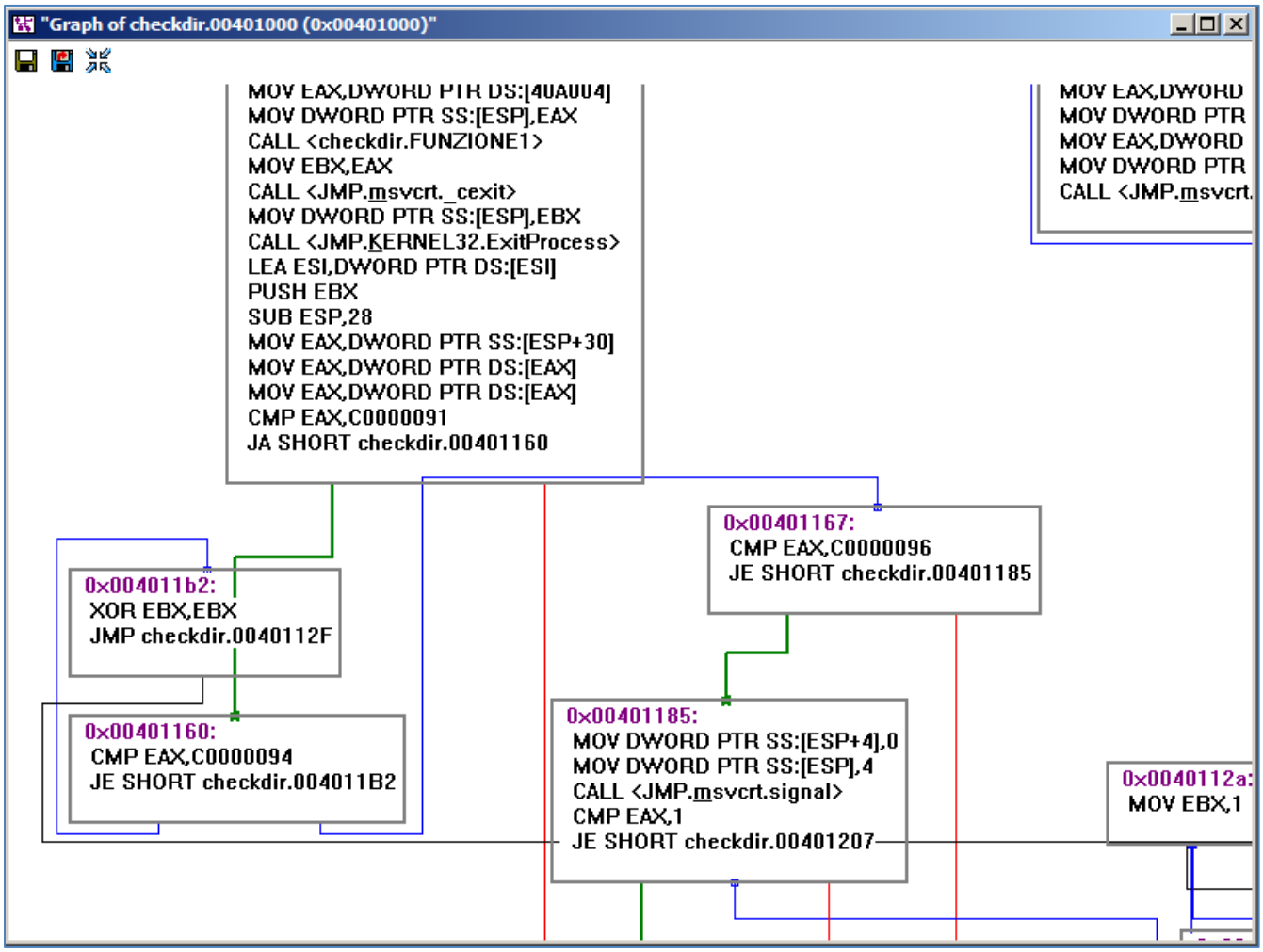


Рисунок 3. Пример графика потока, сгенерированный Immunity

Для получения более подробной информации о Jump-инструкциях Ассемблера, стоит обратиться к этому справочнику.

Два других функциональных значка, «Step into» и «Step over», полезны на стадии изучения программы: с их помощью можно выполнять по одной инструкции за раз. Разница между ними состоит в следующем: при выполнении инструкции «CALL», «Step into» переместит курсор к вызываемой функции, а «Step over» просто выполнит функцию, не перемещая курсор. При выполнении инструкции JUMP, и «Step Into» и «Step over» будут вести себя одинаково, а курсор будет просто перемещён к указанному смещению.

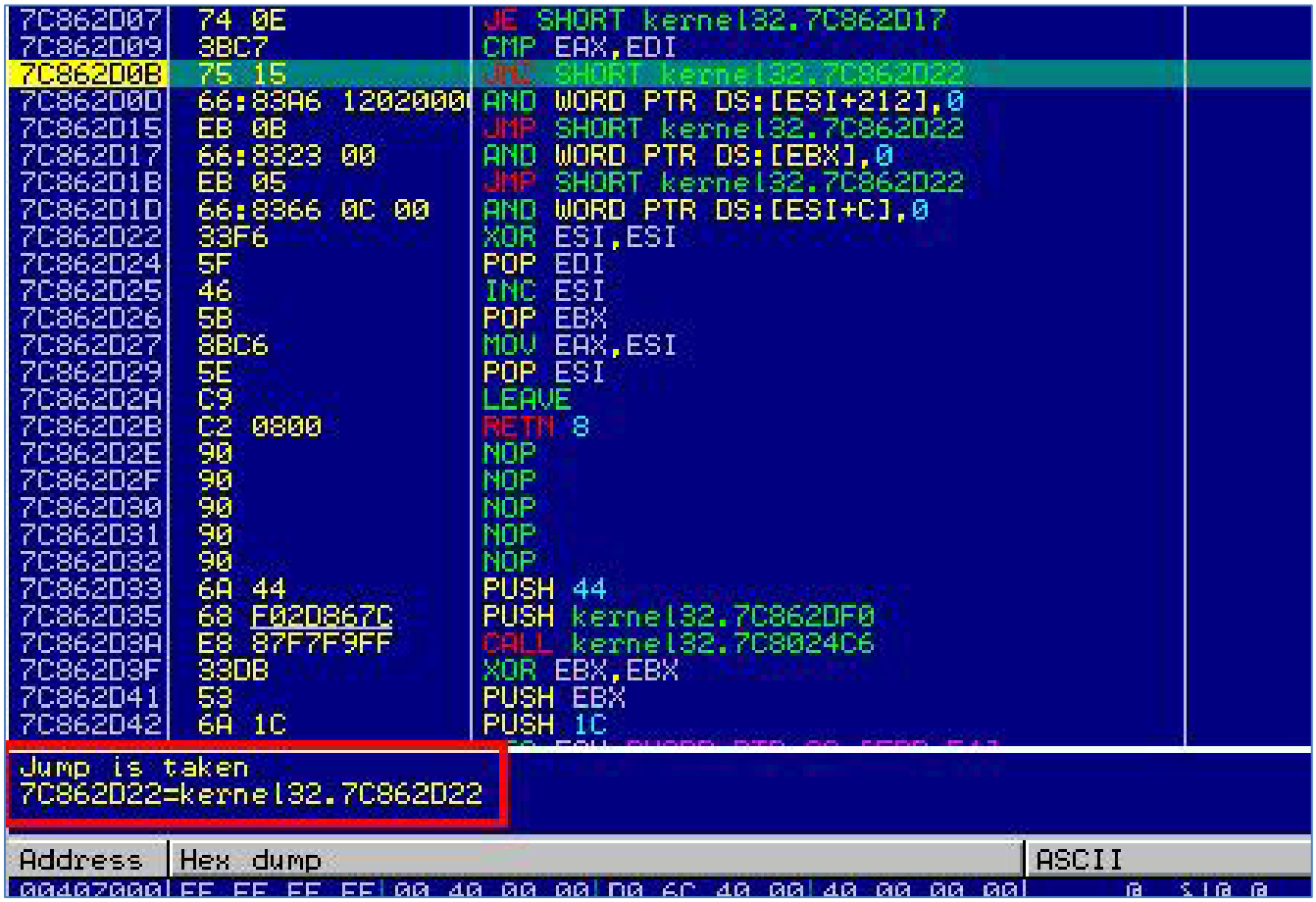


Рисунок 4. Оценка Jump-инструкции

Курсор (это зеленая линия на интерфейсе Immunity), который мы видим на рисунке 4, размещен на смещении 0x7C862D0B, на инструкции JNZ («Jump if not zero»). В небольшой подпанели, которая расположена под панелью инструкций по сборке, Immunity оповещает нас, что будет выполнена инструкция JUMP (красная рамка на рисунке 4), так как инструкция, непосредственно предшествующая JNZ (CMP EAX, EDI), полностью удовлетворяет требованиям инструкции JUMP. При нажатии на кнопки «Step into» или «Step Over», курсор переместится на смещение 0x7C862D22, которое также видно на рисунке 4 (инструкция: XOR ESI, ESI).

На рисунке 5, курсор находится на ещё не выполненной инструкции CALL. Подпанель показывает, что её выполнение вызовет блок инструкций со смещением 0x77C0537C:

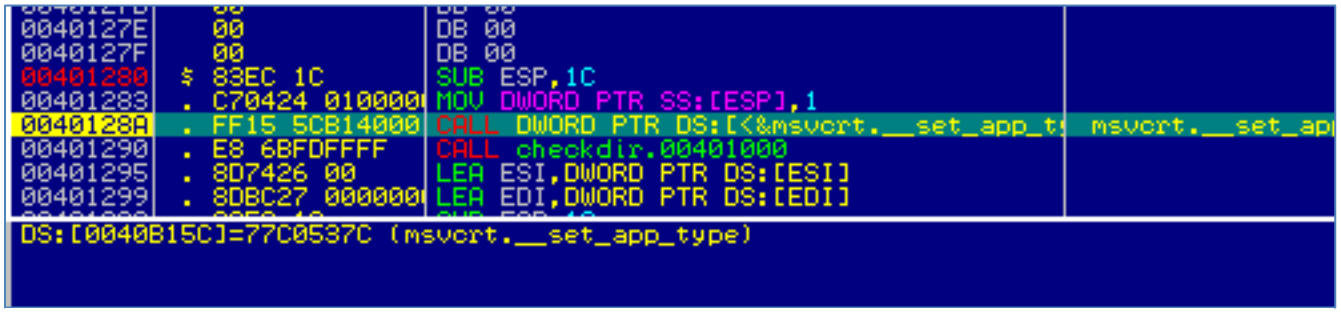


Рисунок 5. Зелёная линия курсора Immunity

При нажатии на «Step into», курсор переместится на смещение 0x77C0537C, как можно понять из расшифровки Jump-инструкций. При нажатии на «Step over», курсор останется на месте, но все инструкции, расположенные между смещением 0x77C0537C и следующими RETN-инструкциями, будут выполнены; соответственно, все значения регистров ЦП будут регулярно обновляться. После выполнения инструкции RETN курсор вновь продолжит следовать за шагами выполнения программы.

Еще пара фактов об окружении Immunity, о которых следует сказать: во-первых, функции программы автоматически выделяются желтыми скобками. Это видно на рисунке 6 (между столбцом смещений и столбцом инструкций машинного языка):

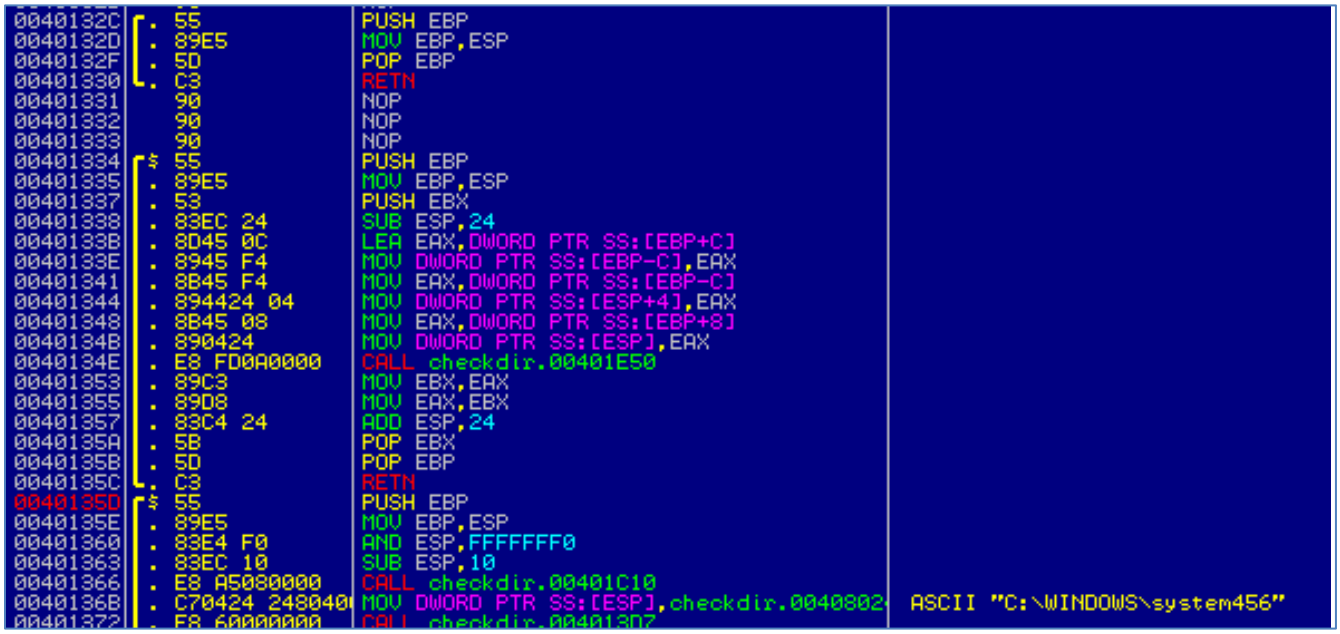


Рисунок 6. Подсветка функций в Immunity

Как можно увидеть, функции в Ассемблере обычно начинаются с инструкций «PUSH EBP - MOV EBP, ESP» и заканчиваются инструкцией RETN. Immunity автоматически находит эти инструкции и выделяет их желтой скобкой.

Во-вторых, Immunity умеет отображать «циклы» (loops), к примеру, «FOR» или «WHILE». Эквивалентный код Ассемблера может заметно отличаться в зависимости от структуры и потока цикла. Самый простой из них показан на рисунке 7:

Рисунок 7. Цикл, подсвеченный в интерфейсе пользователя

Цикл начинается со смещения 0x004010CE и обозначается символом «больше чем» («>»).

Инструкция Ассемблера в строке, помеченной символом «>» (первая строка на рисунке 7), представляет собой код операции «INC» («increment»), и увеличивает значение,

хранящееся в регистре EAX (Накопительный регистр). Сразу после выполнения этого кода операции, будет выполнена следующая инструкция ассемблера - инструкция «CMP» («Compare»). В процессе её выполнения, значение, полученное в результате вычисления выражения из смещения 0x004010CF, будет сравнено с нулём. Последняя строка (смещение 0x00400D3), несёт в себе инструкцию ассемблера «JNZ» («Jump if not zero»); следовательно, если значение, полученное в результате вычисления выражения, не равно нулю, то произойдёт переход на смещение 0x004010CE. Данная операция прописана маленьким «символом стрелки», указывающим вверх («^»). Эти два символа – > и ^ – являются графическим индикатором цикла.

Пример 1

Фрагмент кода, который я приведу чуть ниже – это простой отрывок кода C, который выполняет следующие действия:

- Переименовывает файл «NOTEPAD.TXT» в файл «NOTEPAD.EXE», либо же помещает его в папку «TEMP» и переименовывает его в файл «NOTEPAD.EXE»;
- Если операция переименования прошла успешно, то запускается файл «NOTEPAD.EXE».

Старые малвари (к примеру, Unitrix), ранее загружались на атакуемое устройство в другом формате файла (например, с расширением .JPEG), а затем переименовывались в .EXE. Это делалось для обхода правил антивирусных программ, которые могут блокировать загрузку подозрительных .EXE-файлов. Новые малвари, в частности – вымогатели, такие как Cryptowall и так далее, изменяют расширения документов на атакуемом устройстве и зашифровывают эти файлы. Пример такого кода очень прост для понимания:

```
#include <windows.h>
#include <stdio.h>
int main(void){
if (MoveFile ( "c:\\temp\\notepad.txt", "c:\\temp\\notepad.exe" )) {
    ShellExecute( NULL, "open", "c:\\temp\\notepad.exe", "", NULL, SW_SHOW );
} else {
  // printf("Errato %d\n",GetLastError());
                        return 0;
                }
            }
```

Сниппет кода 1. Код C примера 1

«MoveFile» и «ShellExecute» - это две функции из API Windows, поэтому для каждой из этих функций требуется файл хэдера «windows.h». «MoveFile» используется для перемещения или переименования файлов (включая расширения); «ShellExecute» же используется для выполнения операций над файлом или каталогом (открытие каталога, поиск файла по каталогу, запуск файла и так далее).

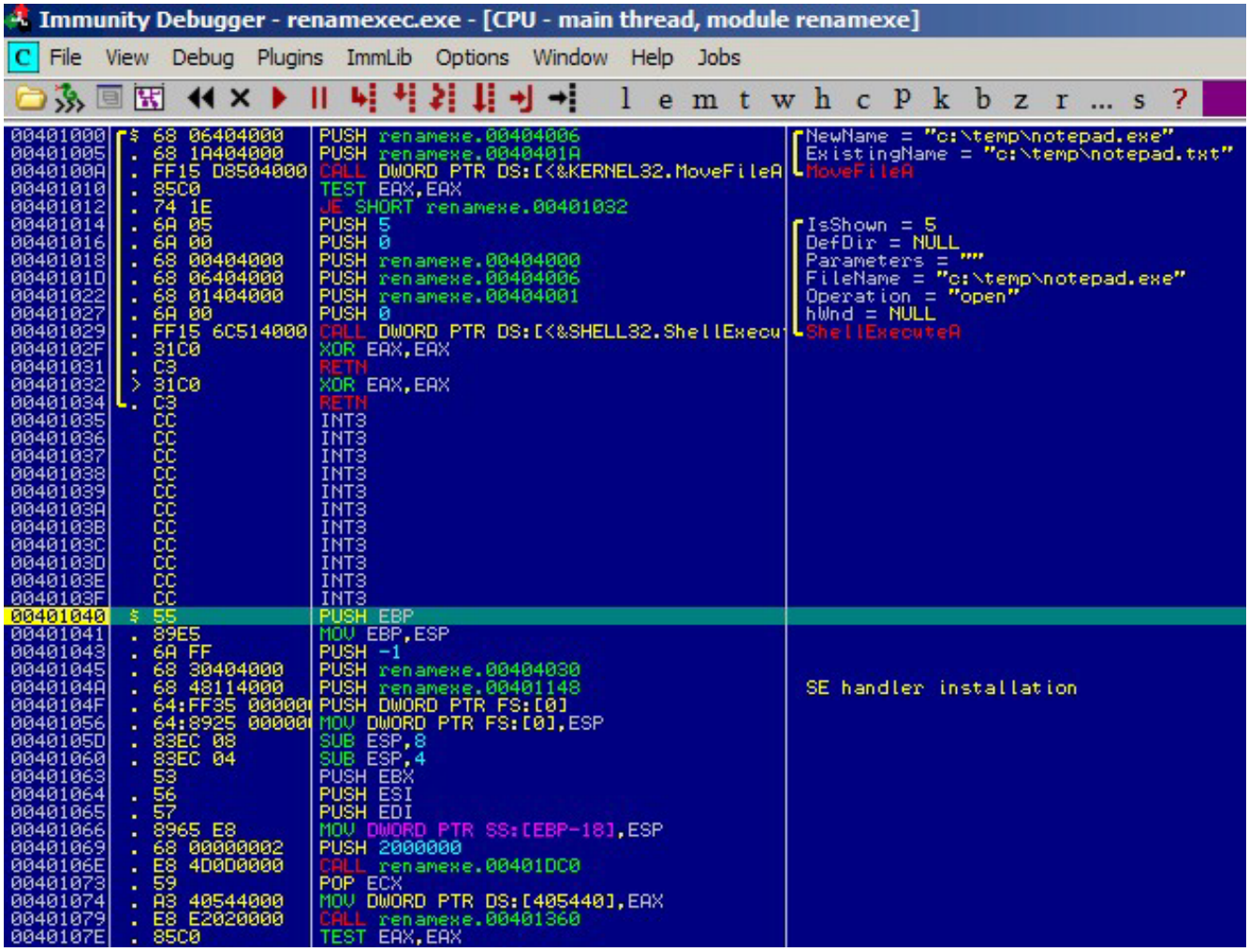


Рисунок 8. Пример 1, открытый в Immunity

Процесс наблюдения за скомпилированным и запущенным исполняемым файлом в Immunity показан на рисунке 8. Самое время рассмотреть его подробнее.

Подсвеченная строка со смещением «0x00401040» - это не Main-функция программы, а точка входа для исполняемого файла. Перед непосредственным запуском кода, написанного программистом, исполняемый файл выполняет ещё множество операций (проверка архитектуры ЦП, настройка обработчиков исключений и так далее), и лишь потом приступает к запуску кода.

Код, представленный на рисунке выше, получен из исполняемого файла, скомпилированного с помощью Pelles C. Pelles помещает код (часть кода), созданный

программистом, в самый верх (строка со смещением 0x00401000), поэтому таким образом очень легко найти результат действий программиста. Но следует помнить, что другие компиляторы могут создавать совершенно другой код на Ассемблере – не всегда поиск будет столь простым.

Одним из способов поиска «созданных программистом» инструкций является поиск «текстовых строк, на которые есть ссылки». Доступ к данной функции поиска можно получить из всплывающего меню, которое появляется при щелчке правой кнопкой мыши на панели Ассемблера, как показано на рисунке 9:

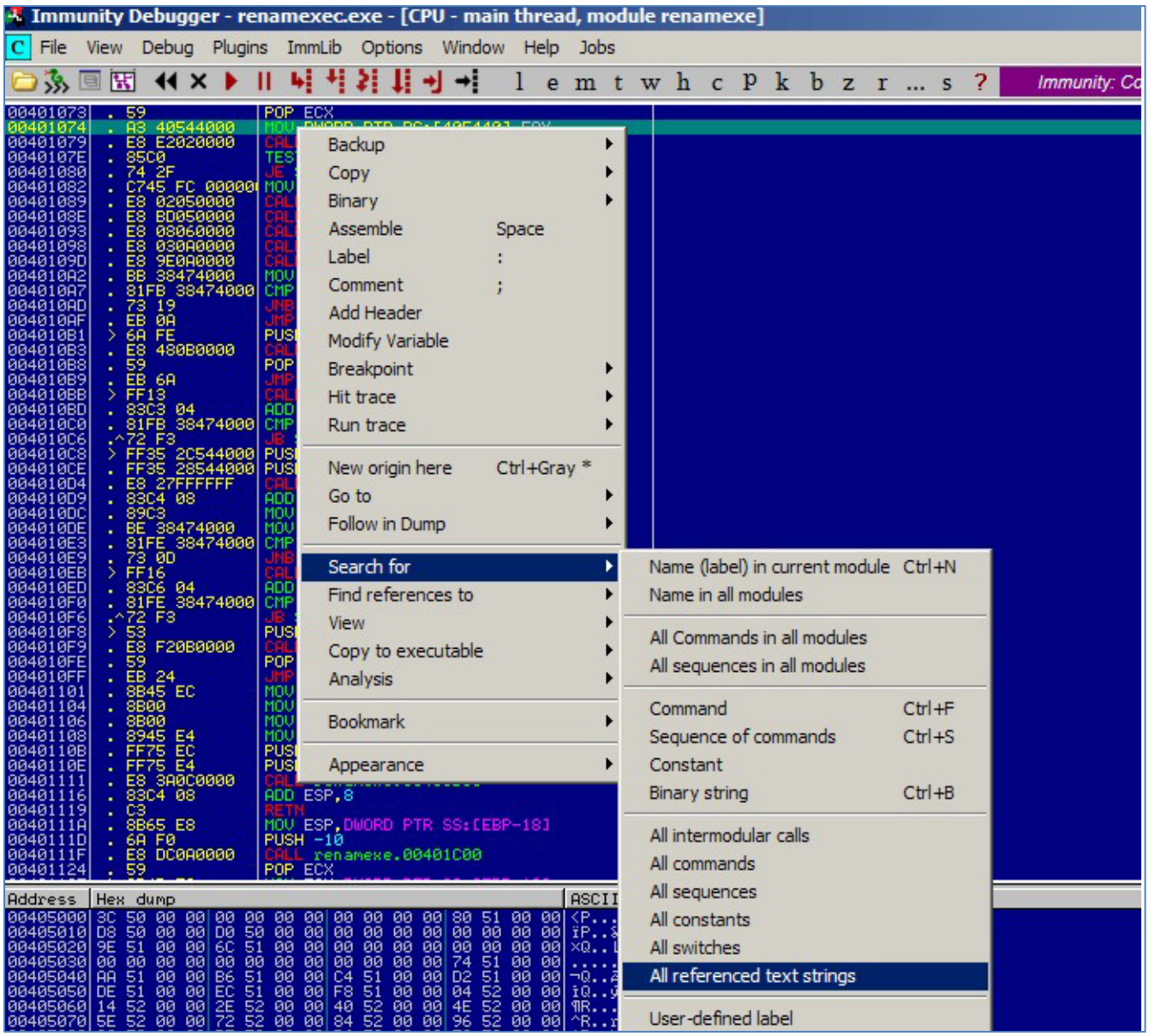


Рисунок 9. Функционал поиска строк, на которые есть ссылки

В результате этого действия мы получим список всех строк, обычно хранимых в разделе «.DATA» исполняемого файла. Эти строки отображаются в Immunity на отдельной панели.

Каждая из строк является гиперссылкой, которая ведёт к соответствующей инструкции в главном окне (панель «Инструкции по сборке»), которая использует эту строку, либо управляет ею каким-то образом.

Как мы видим на рисунке 10, некоторые строки (конкретно в этом примере – это строки, содержащие «ASCII C:\TEMP\NOTEPAD.EXE») могут представлять интерес, поскольку обычно этот исполняемый файл не находится в каталоге «TEMP», а текстовый файл «NOTEPAD.TXT» вообще отсутствует в Windows по умолчанию:

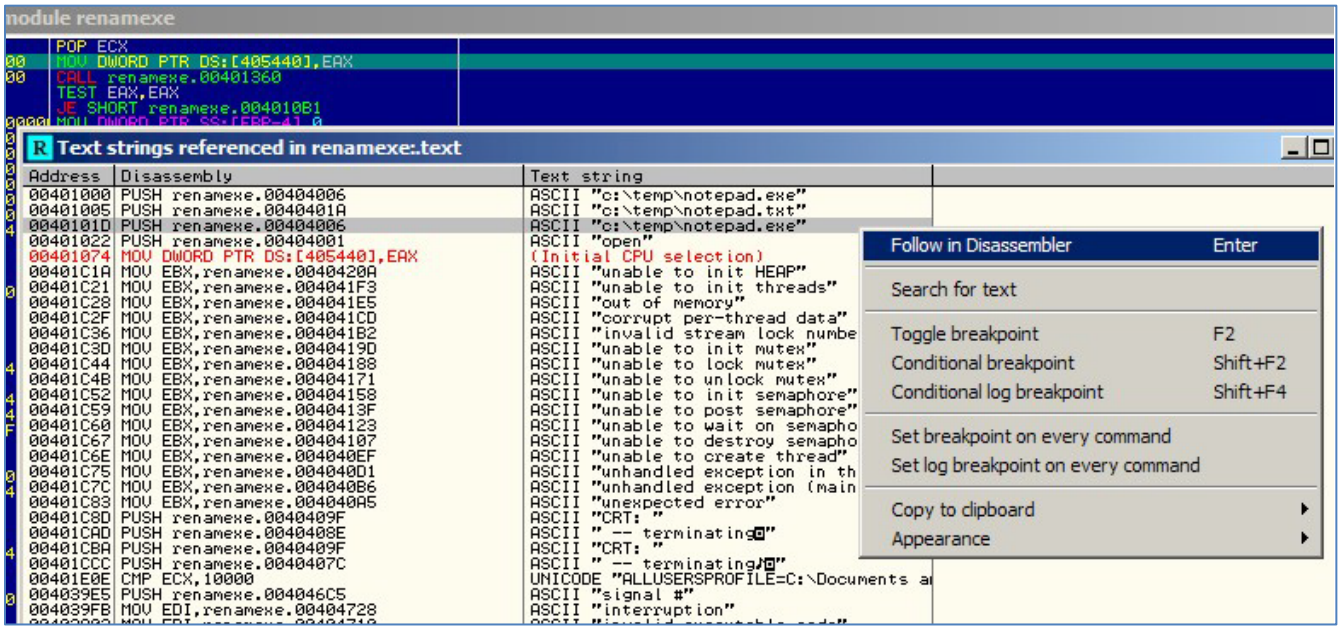


Рисунок 10. Строки, на которые есть ссылки в анализируемом файле

Сразу после того, как подобным образом (Follow in Disassembler) будет выбрана текстовая строка, в основной панели Immunity курсор будет перемещён точно на неё:

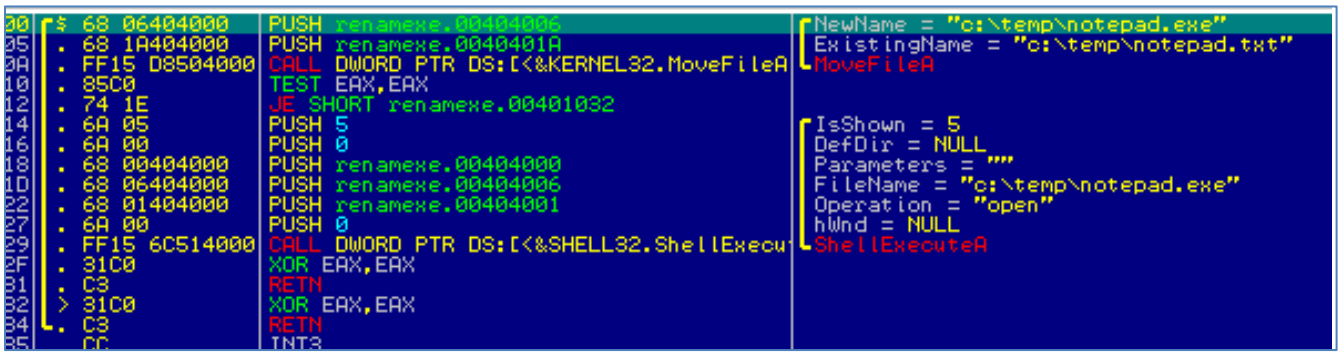


Рисунок 11. Аргумент для инструкции MoveFileA

Первый пример кода состоит из одной уникальной функции (Main-функции): уже из MAIN() вызываются ещё две упомянутые функции («MOVEFILE» и «SHELLEXECUTE»). Кроме того, что эти функции помещены в группы «желтыми скобками», упомянутыми в предыдущем абзаце, мы видим, что порядок, в котором аргументы предоставлены в

функции, противоположен порядку, используемому в исходном коде. Порядок перевёрнут по той причине, что для компиляции этого исполняемого файла (и следующих файлов в этой статье) применяется политика вызовов «_CDECL». Чтобы получить представление о различных используемых опциях компилятора, обратитесь к приложению А. В Microsoft MSDN мы можем найти информацию, что политика вызовов «CDECL» - это порядок предоставления аргументов «справа налево».

В заключение, из рисунка 11 видно, что рассматриваемая программа сначала выполняет инструкцию «MoveFile», переименовывая файл «NOTEPAD.TXT» в «NOTEPAD.EXE» (полные пути каталога передаются функции в качестве аргументов). Затем вновь созданный «NOTEPAD.EXE» запускается функцией «ShellExecute».

Пример 2

Код, приведенный ниже – это ещё один пример кода C, немного более сложный, чем в Примере 1. Этот код проверяет наличие каталога «C:\\WINDOWS\\system456»: если такой каталог существует, то программа ничего не предпринимает и завершает работу. Если же такого каталога нет, то будут выполнены другие инструкции, которые создадут ключ реестра, запишут файл и откроют сокет.

В данном случае код делится на несколько функций, вызываемых из функции «Main()». Во втором сниппете кода я разделил все функции разделителем зеленого цвета, чтобы облегчить чтение, восприятие и понимание кода. В свою очередь, каждая функция будет вызывать несколько функций WinAPI, на этот раз не только из файла хэдера «WINDOWS.H», но и из файлов хэдера «winsock.h» (для сокета) и «aclapi.h» (поскольку создание раздела реестра требует более высокого уровня доступа, который можно получить путём использования структуры данных «SECURITY_ATTRIBUTES»).

Данный код эмулирует те действия, которые обычно совершаются огромным количеством вредоносных программ. Перед запуском кода многие малвари проверяют атакуемое устройство на наличие:

1. Конкретного антивируса (который уже может иметь в базе сигнатуру той вредоносной программы, которая будет загружена);
2. Прочих средств обнаружения вредоносных программ, анализаторов и инструментов анализа.

Это проверка производится посредством поиска определенных установочных файлов и каталогов на устройстве жертвы. Если эти файлы отсутствуют, и вредоносный код может быть «безопасно» загружен и запущен на зараженной машине, то вредоносная программа может приступить к выполнению других действий, к примеру, добавить механизм сохранения, прописав значение реестра в общих ключах автозапуска (например, «HKEY_CURRENT_USER\Microsoft\Windows\CurrentVersion\Run»). Другие распространенные действия, выполняемые вредоносными программами – это загрузка

файлов, обращение к серверам C&C, открытие сокетов, HTTP-сеансы, запись файлов на диск и так далее.

Приведенный ниже код начинается с проверки наличия каталога «C:\WINDOWS\SYSTEM456» с помощью функции «CONTROLLA». Если каталог будет найден, то выполнение кода будет остановлено. Но, поскольку этот каталог не создаётся при установке Windows, обычно происходит следующее:

- Создаётся ключ реестра с именем “PROVADISCRITTURA” (функцией «SCRIVIREGISTRO»);
- Создаётся текстовый документ в каталоге C:\ (функцией «SCRIVIFILE»);
- Открывается сокет к Google на порте 80 (функцией «CONNETTI»).

В отличие от предыдущего сниппета исходного кода (где были вызваны только два API, и присутствовала только одна функция «MAIN»), в этом примере код имеет функцию MAIN(), несколько дополнительных функций, и разделен на большее количество функций: поэтому программный поток здесь более сложный и динамичный, что особенно хорошо отслеживается в отладчике.

Примечание переводчика: код привел в том виде, в каком он приведён в исходном мануале, включая разрывы страниц – поэтому извиняюсь за мелкий шрифт и пробел в конце страницы.

.

```
#include <stdio.h>
#include <windows.h>
#include <strings.h>
#include <stdbool.h>
#include <aclapi.h>
#include <winsock.h>
#pragma comment(lib,"ws2_32.lib")

int controlla(const char *);
void scrivifile(void);
void scriviregistro(void);
void connetti(void);

int valore;
HANDLE hFile;
BOOL bRet = FALSE;
char* bBuffer;
DWORD bytesdascrivere;
DWORD dwWritten;
```

```
//**********************************************************************

int main(void){
    valore = controlla("C:\\WINDOWS\\system456");
            if (valore == 1){
            printf(" valore trovato : %d\n", valore);
    exit(0);
}
    If (valore == 0) {
            scrivifile();
            scriviregistro();
            connetti();
            exit(0);
            }
            else
            exit(0);
}
//**********************************************************************

int controlla(const char* percorso){
  DWORD var = GetFileAttributesA(percorso);
  if (var == INVALID_FILE_ATTRIBUTES)
    return false;
  if (var & FILE_ATTRIBUTE_DIRECTORY)
    return true;
  return false;
}
//**********************************************************************

void scrivifile(void){
  hFile = CreateFile ("C:\\txtdiprova.txt", GENERIC_WRITE, 0, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL);

    bBuffer = "Scrivi qualcosa.";
    bytesdascrivere = (DWORD)strlen(bBuffer);
    bRet = WriteFile (hFile, bBuffer, bytesdascrivere, &dwWritten, NULL);
}
//**********************************************************************

void scriviregistro(void){
            PSECURITY_DESCRIPTOR secdesc = NULL;
            DWORD dwDisposition;
            SECURITY_ATTRIBUTES sa;
              LONG lRes;
              HKEY hkSub = NULL;

  sa.nLength = sizeof(SECURITY_ATTRIBUTES);
  sa.lpSecurityDescriptor = secdesc;
  sa.bInheritHandle = FALSE;

  char cName[] = "Provadiscrittura";
  HKEY hKey = HKEY_CURRENT_USER;
  lRes = RegCreateKeyEx(hKey, cName, 0, "", 0, KEY_ALL_ACCESS, &sa, &hkSub, &dwDisposition);

}
```

```
//**********************************************************************
void connetti(void){
  WSADATA wsa;
  SOCKET s;
  struct sockaddr_in server;
  WSAStartup(MAKEWORD(2,2),&wsa);
  s = socket(AF_INET , SOCK_STREAM , 0 );

  server.sin_addr.s_addr = inet_addr("74.125.235.20");
  server.sin_family = AF_INET;
  server.sin_port = htons( 80 );
  connect(s , (struct sockaddr *)&server , sizeof(server));
        exit(0);
}
```

Сниппет кода 2. Код C примера 2

В этот раз, первым шагом после открытия скомпилированного .EXE-файла в Immunity стала проверка всех вызовов WinAPI-функций, выполняемых программой. Чтобы произвести такую проверку, нужно щёлкнуть правой кнопкой мыши на главной панели и выбрать «SEARCH FOR – ALL INTERMODULAR CALLS», как показано на рисунке 12.

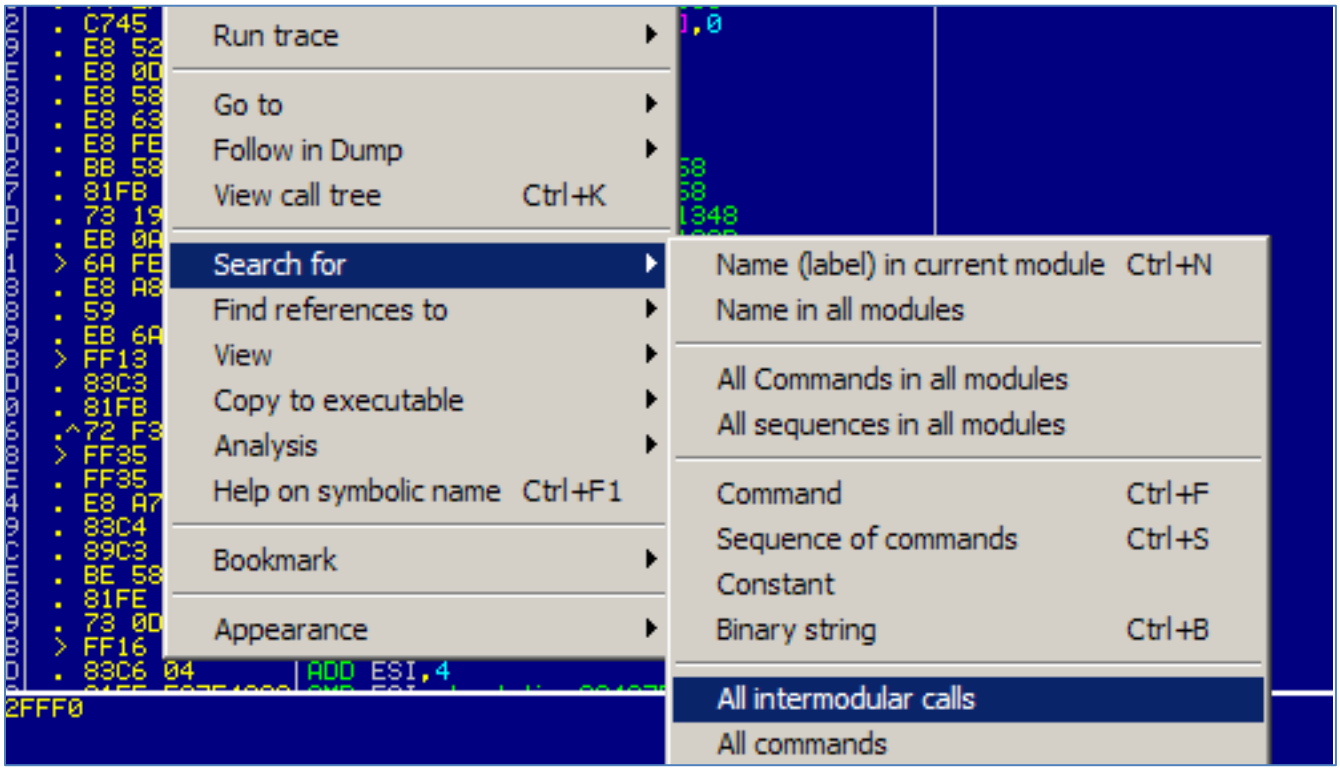


Рисунок 12. Функционал "Search for all intermodular calls"

В результате мы увидим список всех вызовов API для программы. Обратите внимание на важный момент: для каждого из вызываемых API, Immunity указывает соответствующую DLL-библиотеку. Это показано на рисунке 13.

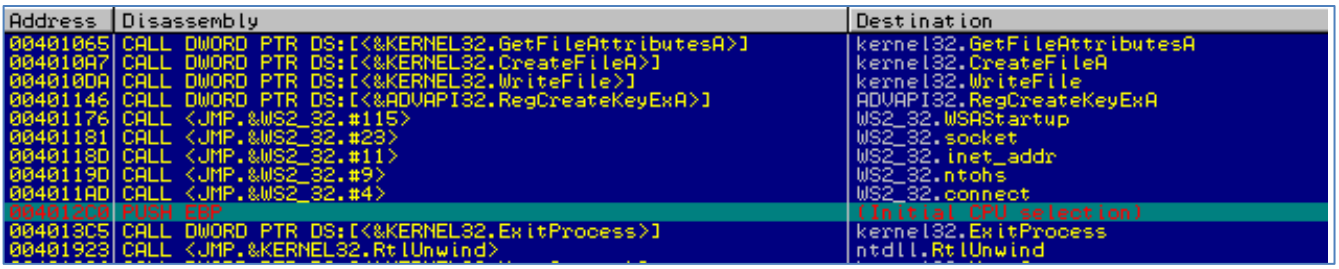


Рисунок 13. Список DLL-библиотек и соответствующих вызываемых функций.

На рисунке 13 мы видим, что над точкой входа в программу (смещение 0x004012C0, выделенное красным цветом) находятся несколько потенциально интересных WinAPI, импортированных из «KERNEL32.DLL»: «GETFILEATTRIBUTESA», «CREATEFILEA» и «WRITEFILE». Затем следуют WinAPI «REGCREATEKEYEXA», импортированный из «ADVAPI32.DLL», и несколько функций для создания сокетов (конкретно тут – сокета «BERKELEY»), импортированные из «WS2_32.DLL». Из этой можно понять, что умеет программа: вытянуть информацию о файле, создать файл, записать содержимое в файл, создать ключ реестра и запустить сокет.

В приведённом примере, Immunity смог распознать все функции и API-интерфейсы, которые были импортированы из библиотек файлов «KERNEL32.DLL», «ADVAPI32.DLL» и «WS2_32.DLL», поскольку это фундаментальные и широко используемые библиотеки DLL. Итак, запомните: имя задействованных WinAPI автоматически определяется и отображается в столбце «DESTINATION» (это видно на рисунке выше) и в первом столбце справа от главной панели (красным шрифтом).

Полезно знать, что Immunity, как и отладчик WinDBG от Microsoft, имеет возможность загрузки таблицы символов отладки Microsoft. Это можно сделать, выбрав «DEBUG – DEBUGGING SYMBOLS OPTIONS» в раскрывающемся меню в верхней части интерфейса.

В отличие от отладчика GDB, Immunity не имеет возможности идентификации функции Main(), каковая выполняется в GDB командой «DISAS MAIN» (*примечание переводчика: не уверен в правильности написания, вполне может быть и «DISASMAIN» - команда разбита по двум строчкам*). Однако, в мануале «Finding Main() – Compiler Code vs. Developer Code» расписан способ найти Main-функцию, следуя логике самой программы.

Во время анализа программы мы столкнёмся с первой серьёзной проблемой: точка входа в программу (обозначенная на основной панели инструкций по сборке как «INITIAL CPU SELECTION», и отображённая на рисунке 13) не соответствует функции MAIN() программы. Поэтому нам предстоит определить отправную точку программы с точки зрения программиста. Очевидно, что определение разницы между кодом, сгенерированным программистом, и кодом, сгенерированным компилятором, может быть достаточно сложным.

Чтобы понять логику программы, необходимо следовать процедуре, описанной в руководстве от Хеффнера (Heffner), комбинируя приведённый в нём modus operandi и

использование функционала Immunity «LABEL» в сочетании с «DISPLAY GRAPH».

Когда одна из функций определена с помощью «SEARCH FOR ALL INTERMODULAR CALLS» или «SEARCH FOR ALL REFERENCED TEXT STRINGS», мы можем присвоить этой функции произвольное имя, установив «метку» (для этого, щелкните правой кнопкой мыши на первой строке инструкции функции и выберите «LABEL»:

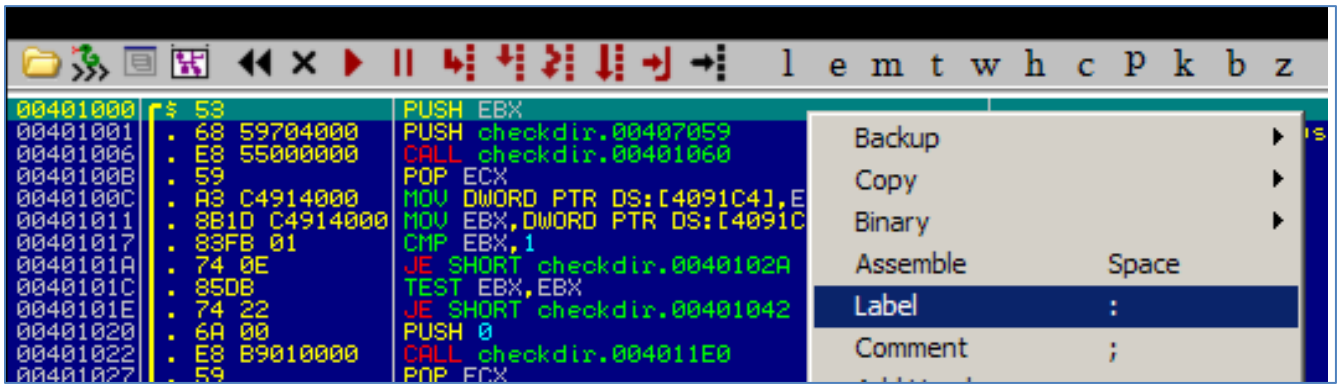


Рисунок 14. Добавление метки к строке инструкции

В данном случае, именем метки назначено «CHECK». Каждую интересующую строку или интермодульного вызов, обнаруженные с помощью этого функционала, можно пометить как «GET FILE ATTRIBUTE», «WRITEFILE», «REGISTRYKEY» и «SOCKET». Затем, активировав функцию «GRAPH DISPLAY», мы увидим, что точка входа программы вызывает блок инструкций на смещении 0x00401302, содержащий четыре инструкции CALL:

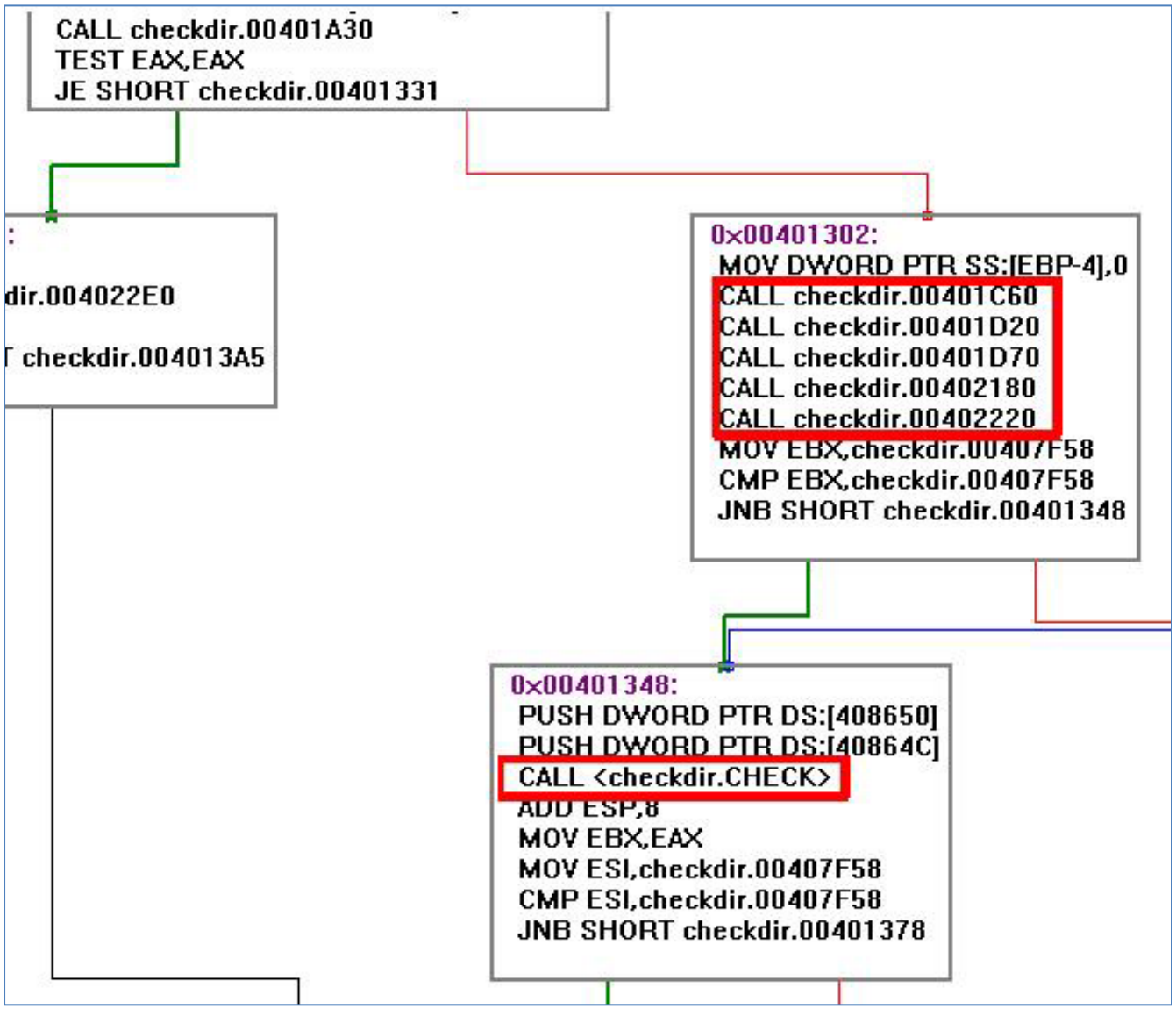


Рисунок 15. График потока для примера 2, точка входа в программу

Изучив код, предоставленный четырьмя кодами операций CALL, мы можем сделать вывод, что ни одна из них не является MAIN() программы. Этому есть несколько причин:

- Смещение 0x00401560 содержит инструкцию CPUID, которая является кодом операции сборки, собирающем дополнительную информацию о процессоре;
- Смещение 0x00401D20 содержит код «GETFILESYSTEMASTIME» (это API другого рода, обычно используемый для инициализации исполняемого файла);
- Смещение 0x00401D70 несёт в себе коды операций, которые соответствуют API-интерфейсам «GETSTARTUPINFOA», «GETFILETYPE», «GETSTDHANDLE» (нам оно не подходит по тем же соображения, что и предыдущее смещение);
- Смещение 0x00402180 содержит ссылку на API «GETCOMMANDLINEA», поэтому, скорее всего, это возможный эквивалент C-инструкции «INT MAIN(VOID)»;

- В смещении 0x00402220 содержится ссылка на API «GetEnvironmentStrings», поэтому, отметаем его по той же причине, что и предыдущее.

На схеме визуализации потока, которая отображена на рисунке 15 можно увидеть, что только что просмотренный блок выводит нас на следующий блок инструкций, в котором есть ещё один потенциально интересный вызов (на рисунке я обозначил его красной рамкой). Смещению этого блока было присвоено имя «<CALL checkdir.CHECK>»). Таким образом, блок-схема становится более удобочитаемой и указывает, что мы близки к коду, созданному программистом.

В том случае, когда вместо точки входа в программу («Initial CPU Selection») подсветкой будет выделена начальная строка инструкции «checkdir.CHECK» (зелёный курсор на главной панели), мы получим другой график, который будет отталкиваться от выделенной линии:

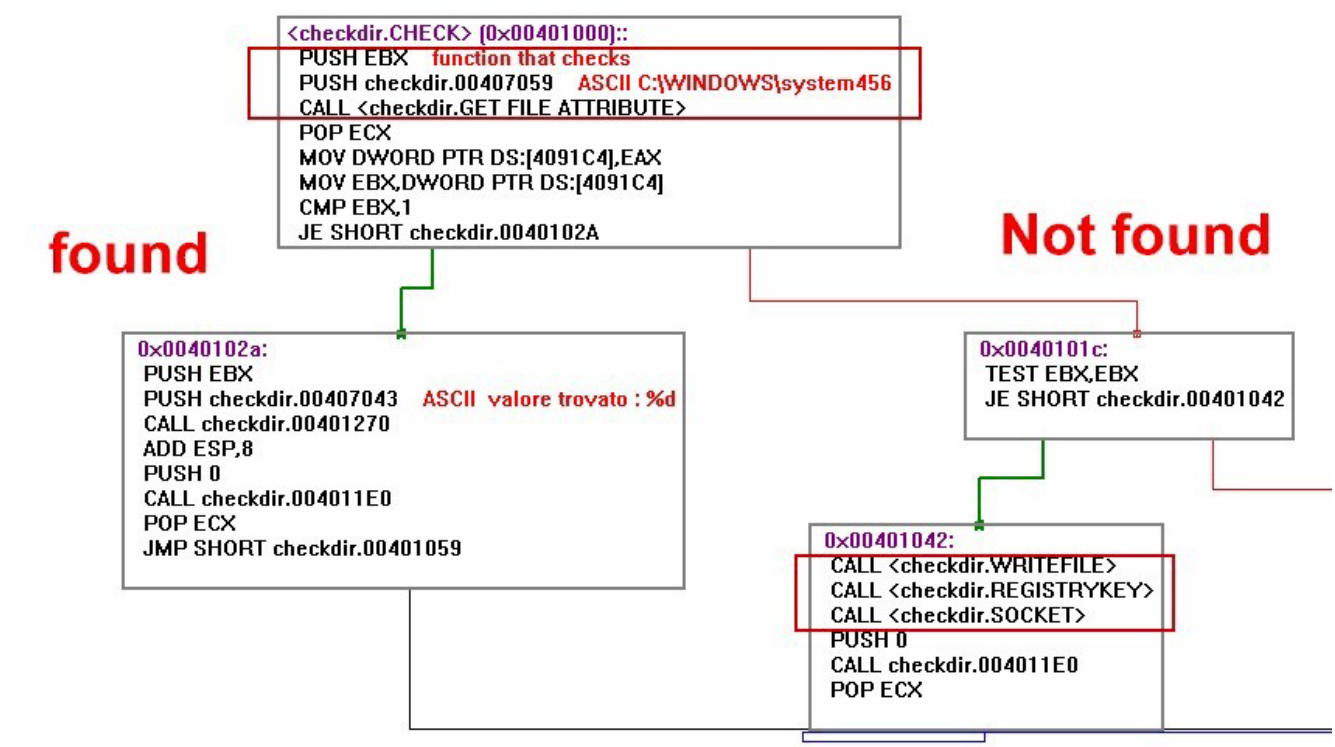


Рисунок 16. График потока примера 2 от точки "checkdir.CHECK"

В таком случае, логика программы отслеживается куда более чётко.

На этом этапе анализа будет нелишним разобраться, как аргументы, включенные в блок инструкций «Socket», передаются функциям. В приведённой структуре данных сокета, мы с помощью функции «htons» («host to network short») указали порт 80 с. В отладчике же мы видим, что этот данный параметр передается через функцию «ntohs» («network to host short») с аргументом «50» («50» в нашем случае – это число 80 в шестнадцатеричной системе). Подробнее о причинах подобной замены функции, можно прочитать в [этой статье](#).

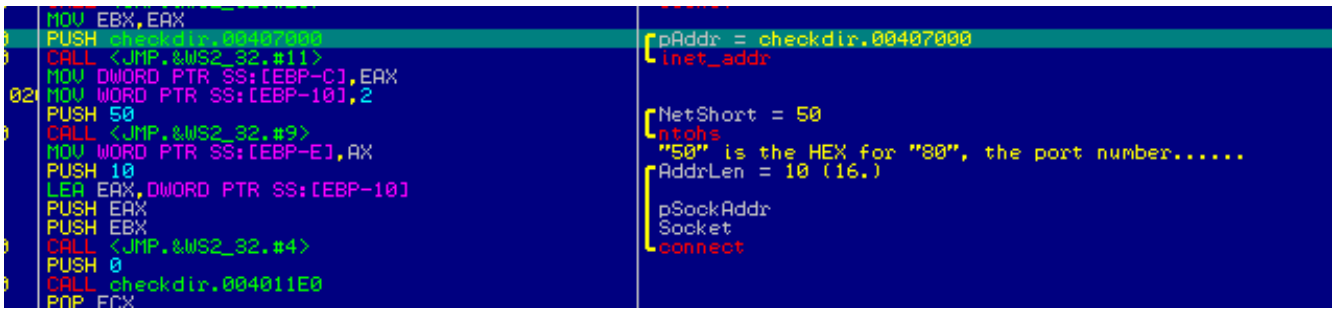


Рисунок 17. Передача аргумента функции "Socket"

Соответственно, в зависимости от процессора, на котором выполняется программа, «данные функции преобразуют изначальный порядок байтов в порядок байтов для сети и обратно». Другой важный момент заключается в том, что, несмотря на то, что IP-адрес передается как аргумент «ASCII STRING», он не отображается в столбце комментариев по неизвестным причинам. Но – его достаточно легко найти с помощью функционала «SEARCH FOR ALL REFERENCED TEXT STRINGS». Тем не менее, мы видим смещение 0x00407000, на котором находится эта строка. Immunity не отображает содержимое раздела «.DATA» ни на одной из четырех основных панелей графического интерфейса: он лишь перечисляет коды операций, которые содержатся в разделе «.TEXT» исполняемого файла (для этой конкретной программы – это смещения от 0x00401000 до 0x00406FFF). Также, отладчик копирует данные ссылок из раздела «.DATA» в столбец комментариев.

Чтобы ясно увидеть момент передачи аргумента, есть возможность использовать триггер остановки разбора для каждой инструкции, содержащейся в блоке «SOCKET» (клавиша «F2»). После запуска программы в отладчике, теперь видно, что IP-адрес отображается на панели стека памяти (смещение 0x0012FDD8). Также, отображена ссылка на его смещение 0x00407000:

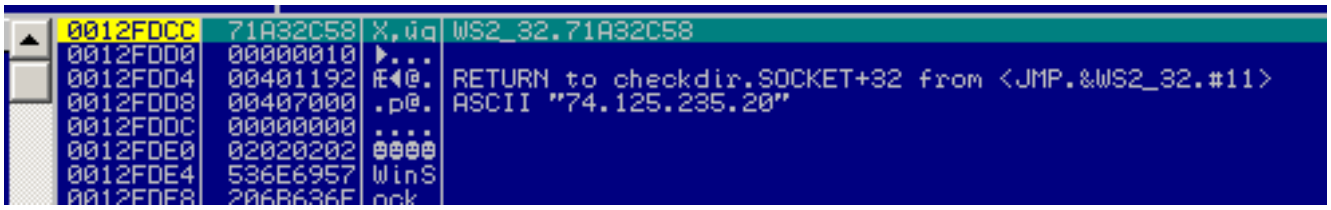


Рисунок 18. IP-адрес, загруженный в стек памяти

После этого IP-адрес перемещается в EAX-регистр (накопительный регистр), как показано на рисунке 19: данный регистр – регистр общего назначения, и при необходимости он может функционировать как своего рода регистр «временного хранения данных».

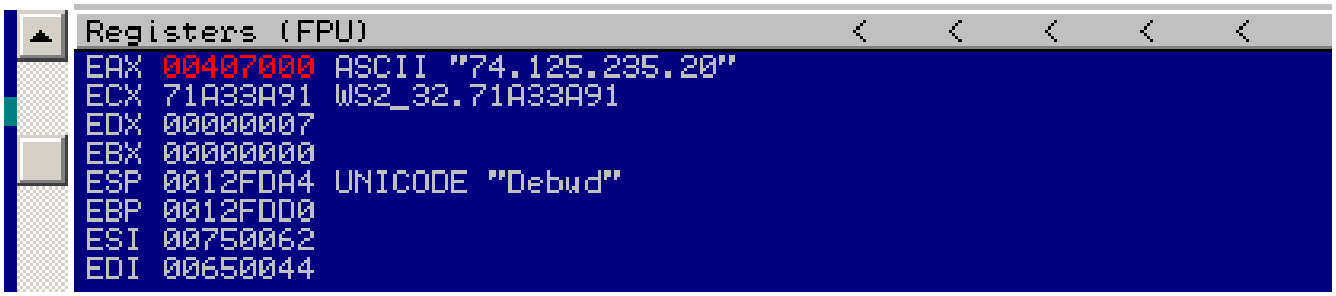


Рисунок 19. Ссылка на ASCII-строку, перемещённую в EAX

EAX-регистр использовался для хранения IP-адреса из стека, потому что за эту операцию отвечал один из модулей, вызванный исполняемым файлом (библиотека, связанная с исполняемым файлом, конкретнее – «WS32_2.DLL»). Если присмотреться повнимательнее к рисунку 20, можно выделить три момента:

1. Заголовок основного экрана графического интерфейса изменился на «CPU – Main thread, module WS2_32.DLL»;
2. Изменение диапазона смещения. Как уже говорилось, диапазон смещения для секции .text этой программы – от 0x00401000 до 0x00406FFF; но на рисунке 20 чётко видно, что текущее смещение это 0x71A32C00;
3. Также, на рисунке 20 показано (смещение, подсвеченное зелёным курсором), что инструкция MOV копирует значение, содержащееся в сегменте стека (SS, Stack Segment), начиная со строки, равной значению Base Pointer (0x0012FDD0), и до смещения, увеличенного на 8 (0x0012FDD8).

Если мы вновь вернёмся к рисунку 18 (панель стека памяти), то увидим, что по факту, IP-адрес находится точно в смещении стека памяти, 0x0012FDD8.

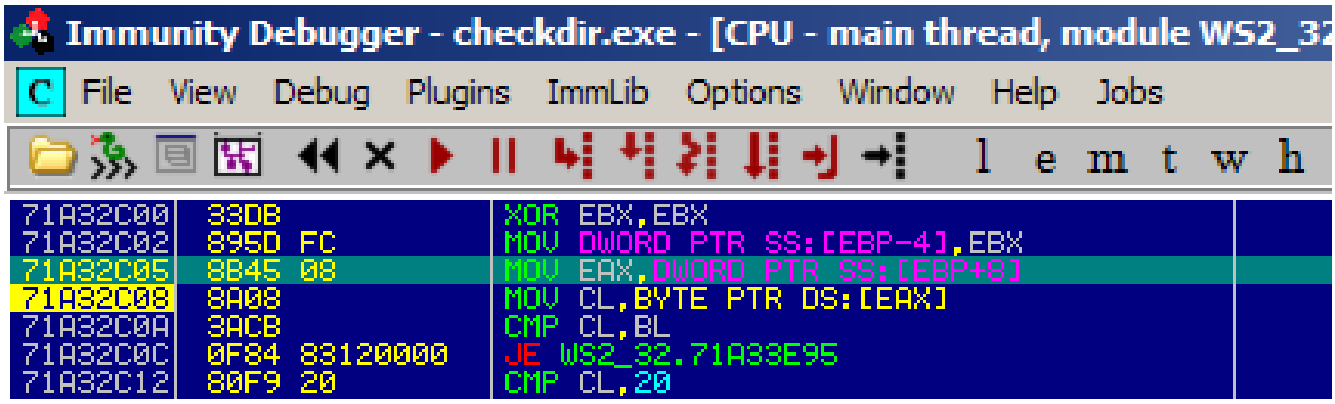


Рисунок 20. Перемещение ASCII-строки в EAX-регистр

Чуть позже, непосредственно перед успешным созданием сокет-соединения, становится видно, что для передачи этого IP-адреса функции «RtlIpv4StringToAddress» (импортированной из модуля «NTDLL.DLL»), использовался EAX-регистр. Более наглядно это показано на рисунке 21.

Рисунок 21. Аргументы, передаваемые функции "RtlIpv4StringToAddress"

Данная функция была автоматически добавлена в программу компилятором, а не программистом; она преобразует строчный IPV4-адрес в двоичный формат. В соответствии с документацией MSDN, эта функция может принимать четыре аргумента. Ссылку на полную документацию по этой функции можно найти в разделе «Ссылки» или по гиперссылке выше.

Четыре аргумента, передаваемые данной функции отображены на рисунке 22 (с панелью стека памяти):

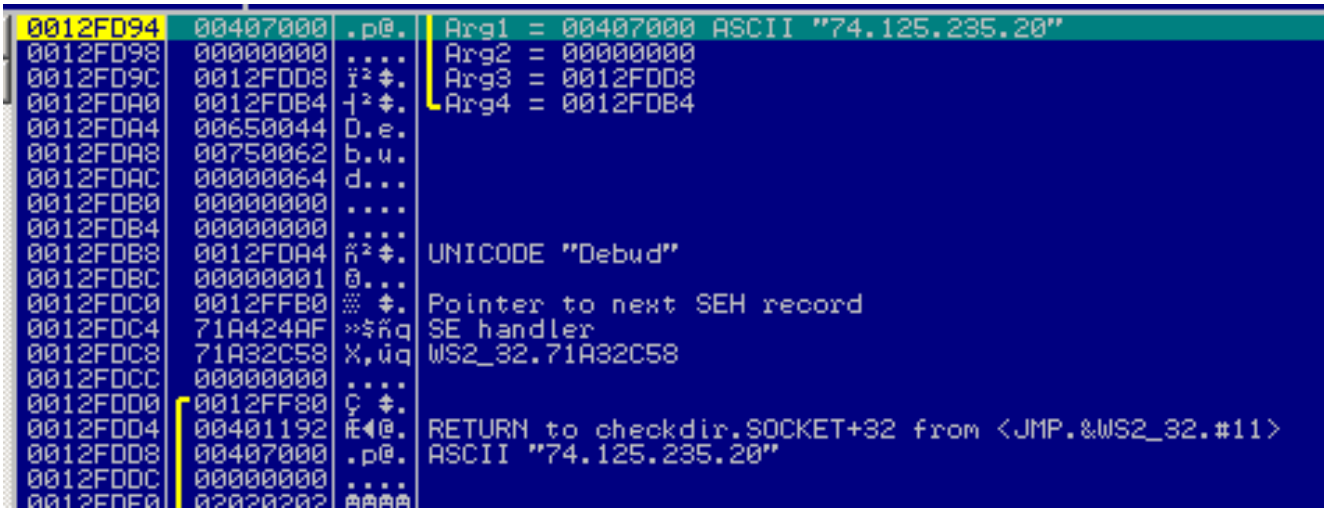


Рисунок 22. Аргументы, передаваемые функции "RtlIpv4StringToAddress"

Пример 3

В примере 2 мы разобрали вызов сокета и установку соединения с внешней инфраструктурой (Berkeley-сокет был выбран для демонстрации передачи аргументов функций сокета). Однако, большинство вредоносных программ, написанных для Windows, используют «родной» виндовский API и обычно обращаются не к Berkeley-сокету, а к API-интерфейсам «Wininet», прописанным в файле хэдера WININET.H.

Кроме того, URL-адрес, получаемый сокетом в качестве аргумента, может быть закодирован (например, чтобы избежать обнаружения или просто усложнить возможный анализ).

Пример кода C для этого примера:

```
#include <windows.h>
#include <wininet.h>
#include <stdbool.h>
#include <stdio.h>

LPSTR decodifica(LPSTR);
BOOL scarica (LPSTR);

int main (void) {
            BOOL bRet;
            LPSTR percorso = "uggc122jjj4rivy3jrofvgr4vg2znva4ugzy";
            LPSTR finale;
            finale = decodifica(percorso);
//          printf("%s\n", finale);
            bRet = scarica (finale);
return 0;
}

//************************************************************************

 BOOL scarica (LPSTR lpszUrl) {
               HINTERNET hInternet;
               HINTERNET hInternetUrl;
               BYTE bBuffer[1024];
               DWORD dwRead;
               BOOL bRet = FALSE;

 hInternet = InternetOpen ("MyAgent/1.0", INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);

 if (hInternet != NULL) {
   hInternetUrl = InternetOpenUrl (hInternet, lpszUrl, NULL, 0, 0, 0);

 if (hInternetUrl != NULL) {
             do {
             bRet = InternetReadFile (hInternetUrl, bBuffer, sizeof (bBuffer), &dwRead);
                                          printf(bBuffer);
                                          printf("\n");
             } while (bRet && dwRead == sizeof (bBuffer));

                          InternetCloseHandle (hInternetUrl);
                          }
             InternetCloseHandle (hInternet);
             }
 return bRet;
 }

 //************************************************************************

LPSTR decodifica(LPSTR input) {
     int cont;
char *buffertemp;
buffertemp = (char*)malloc(sizeof(char) * (strlen(input)+1));

for (cont = 0; cont<strlen(input); cont++) {
               //    else
     buffertemp[cont] = (((input[cont]-97)+13)%26+97);
            if (buffertemp[cont] == 'X') buffertemp[cont] = 58;
            if (buffertemp[cont] == 'Y') buffertemp[cont] = 47;
            if (buffertemp[cont] == 'Z') buffertemp[cont] = 45;
            if (buffertemp[cont] == '[') buffertemp[cont] = 46;

}
buffertemp[strlen(input)] = '\0';
            return buffertemp;
}
```

Сниппет кода 3. Код С примера 3.

Для анализа этого исполняемого файла, были выполнены шаги, уже рассмотренные в примерах 1 и 2 («SEARCH FOR ALL REFERENCED TEXT STRINGS» и «SEARCH FOR ALL INTERMODULAR CALLS»). С их помощью, а также с помощью функционала «DISPLAY GRAPH», мы вычленили несколько потенциально интересных API и функций. Далее, мы «разграничили» блоки инструкций с помощью меток, комментариев и использования триггеров остановки разбора (напомню, это «F2»). И хотя в результате у нас появилось более чёткое представление о действиях, выполняемых исполняемым файлом, проблема «закодированного URL» никуда не делась.

Во время анализа и наблюдения за выполнением программы, мы видим, что на смещении 0x004010CE происходит интересный цикл, который разбирает каждый символ кодированной строки (рисунок 23).

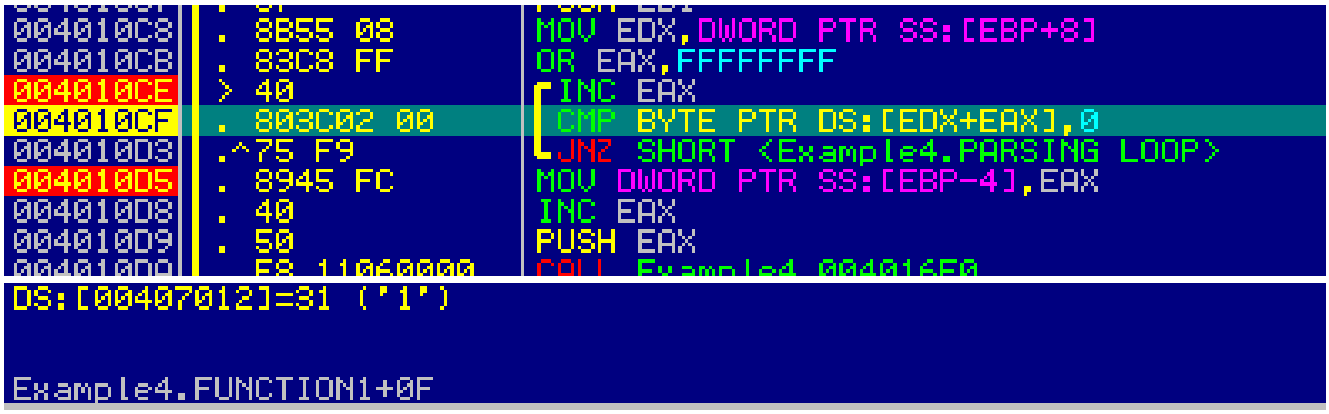


Рисунок 23. Цикл, разбирающий каждый символ кодированной строки

Если мы обратим внимание на подпанель, то увидим, что инструкция «COMPARE» («CMP»), выделенная зеленым курсором, сравнивает текущее значение из сегмента данных («DS») в смещении 0x00407012, с нулем.

В данном случае тут отображён пятая петля цикла, соответствующая символу «1». Это пятый символ зашифрованной строки «UGGC122JJJ4RIVY3JROFVGR4VG2ZNVA4UGZY».

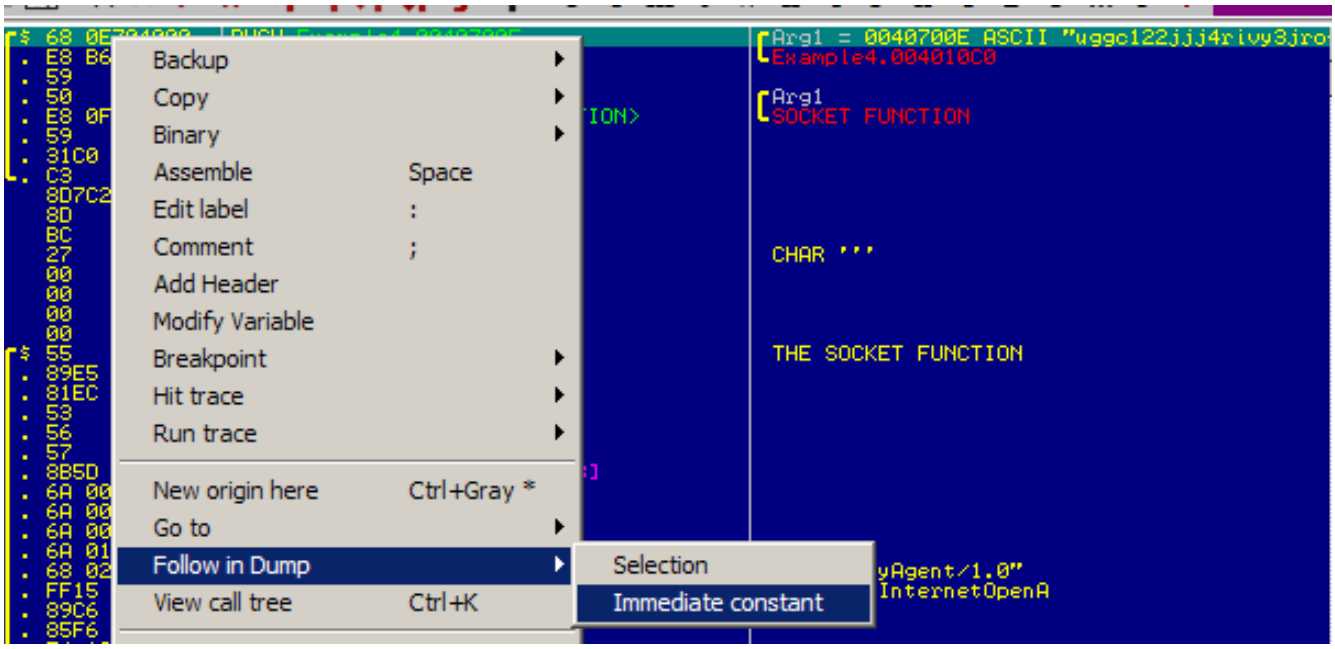


Рисунок 24. Определение положения зашифрованной строки в разделе .data

Здесь происходит обращение к регистру «DS», поскольку рассматриваемая строка находится в разделе .DATA исполняемого файла. В поиске этой строку в разделе .data, нам может помочь функция «FOLLOW IN DUMP» (для её вызова нужно щелкнуть правой кнопкой мыши на панели инструкций по сборке). Опция «IMMEDIATE CONSTANT» приведёт нас напрямую к строке, а опция «SELECTION» приведет нас к инструкции «PUSH» (в разделе .TEXT исполняемого файла). После того, как выбрана опция «IMMEDIATE CONSTANT» и изучен отобразившийся дамп, мы можем отследить фактическое смещение символа «1» – 0x00407012 (нужные данные подсвечены зеленым на рисунке 25):

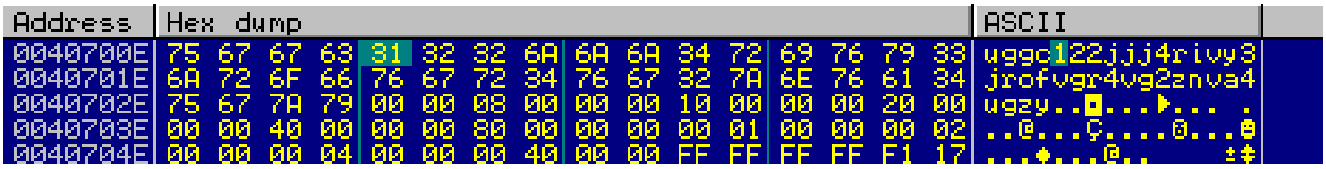


Рисунок 25. Определение зашифрованной строки в дампе раздела .data

Теперь стоит уделить отдельное внимание инструкции «CMP BYTE PTR DS: [EDX+EAX], 0».

Эта инструкция производит сравнение с нулём (обозначается символом «0»), потому что этот символ, представляемый ANSI C как «\0») является символом конца строки. Другими словами, эта инструкция сравнивает каждый символ строки со значением NULL. Если результат «не равен нулю», то цикл будет продолжаться, поскольку присутствует инструкция «JNZ» («JUMP IF NOT ZERO»).

Почему же использовалось выражение «[EDX+EAX]»?

Я уже упоминал, что EAX – это разновидность временного накопителя, в котором может содержаться множество разных типов данных. В данном случае, он используется для

хранения номера элемента исследуемого массива (пятый символ, потому что первый элемент массива равен 0). EDX же - другой регистр общего назначения – хранит в себе смещение первого символа строки, то есть 0x0070700E. Если мы добавим «4» к этому смещению, то получим 0x00407012. За каждую петлю цикла, EAX увеличивается на единицу. Это видно на рисунке 26:

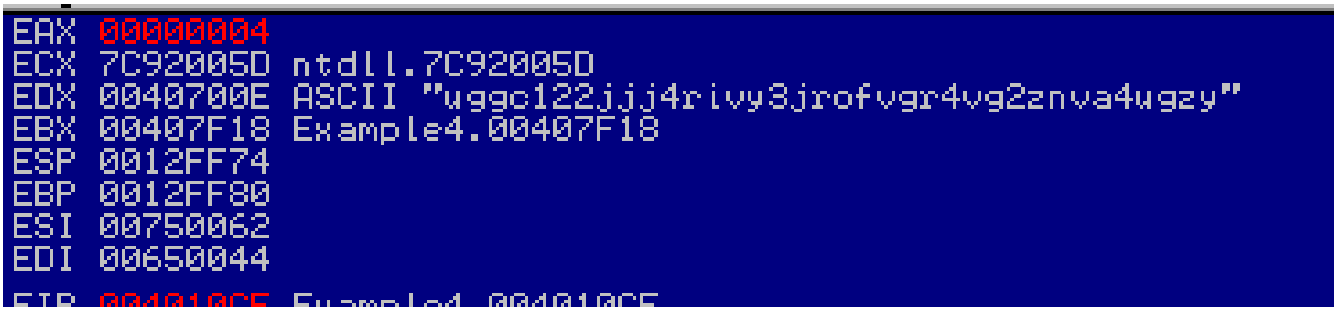


Рисунок 26. Текущие значения EAX и EDX

На второй таблице видно, что указатели ESI и EDI – это индексные регистры, которые, как мы помним, используются для манипулирования массивами и строками (ESI – для хранения строк и их последующего копирования в EDI). Кроме того, мы все прекрасно понимаем, что строка – это массив символов.

После того, как мы закончим все рутинные операции декодирования и тщательного проследим за выполнением программы с использованием функции «STEP INTO», мы наконец сможем увидеть в регистре ESI декодированную строку, «символ за символом», удерживая нажатой клавишу F7 (рисунок 27):

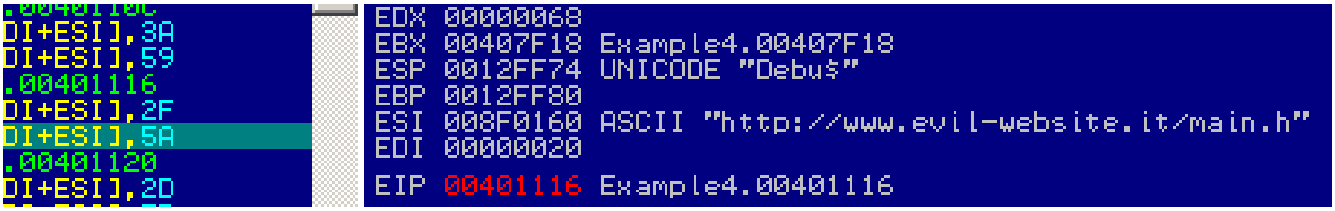


Рисунок 27. Расшифрованный URL-адрес, созданный в регистре ESI

Далее мы увидим, что инструкция «MOV EAX, ESI» (выделена зеленым цветом) скопировала расшифрованный URL из ESI в накопительный регистр EAX (рисунок 28). Причина копирования этого URL-адреса в EAX аналогична той, с которой мы уже сталкивались в примере 2 (абзацы, где мы рассматривали передачу аргументов в структуру данных сокета): после этих операций, расшифрованный URL-адрес будет обработан API-интерфейсами модуля WININET.DLL и, с помощью регистра EAX, будет передан этому модулю в качестве аргумента:

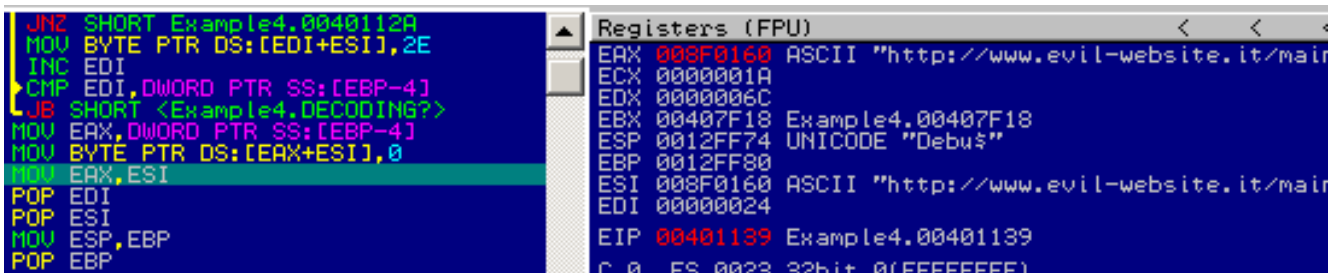


Рисунок 28. Строка регистра ESI, скопированная в регистр EAX

Заключение

Итак, подведём итоги. Хотя эти три примера кода безопасны для выполнения, они имитируют некоторые действия, выполняемые вредоносными программами. Это – реальная симуляция основ реверсивной инженерии. Основная цель этой статьи – произвести общий обзор отладчика Immunity и рассказать об очень простых и практичных операциях, с помощью которых мы можем получить полезную информацию об анализируемом исполняемом файле. Когда код C в отладчике преобразуется в Ассемблер, то мы получаем неимоверное количество строк инструкций: думаю, лишним будет говорить, что изучение всех строк потребует огромных (и очень часто бесполезных) усилий. Ведь при анализе реальных вредоносных программ именно время играет решающую роль. Из этого можно сделать вывод, что настоящее искусство реверсивной инженерии – это способность определить ключевые действия или исполняемого файла, или DLL-библиотеки, или присоединенного процесса. Конечно, очевидно, что глубокие познания в языках Ассемблера и C – это очень полезные навыки. Тематические исследования, предоставленные в этой статье, являются лишь начальной точкой удивительной, но очень сложной области – реверсивной инженерии. И, будьте уверены, кривая обучения здесь очень крутая.

Приложение A

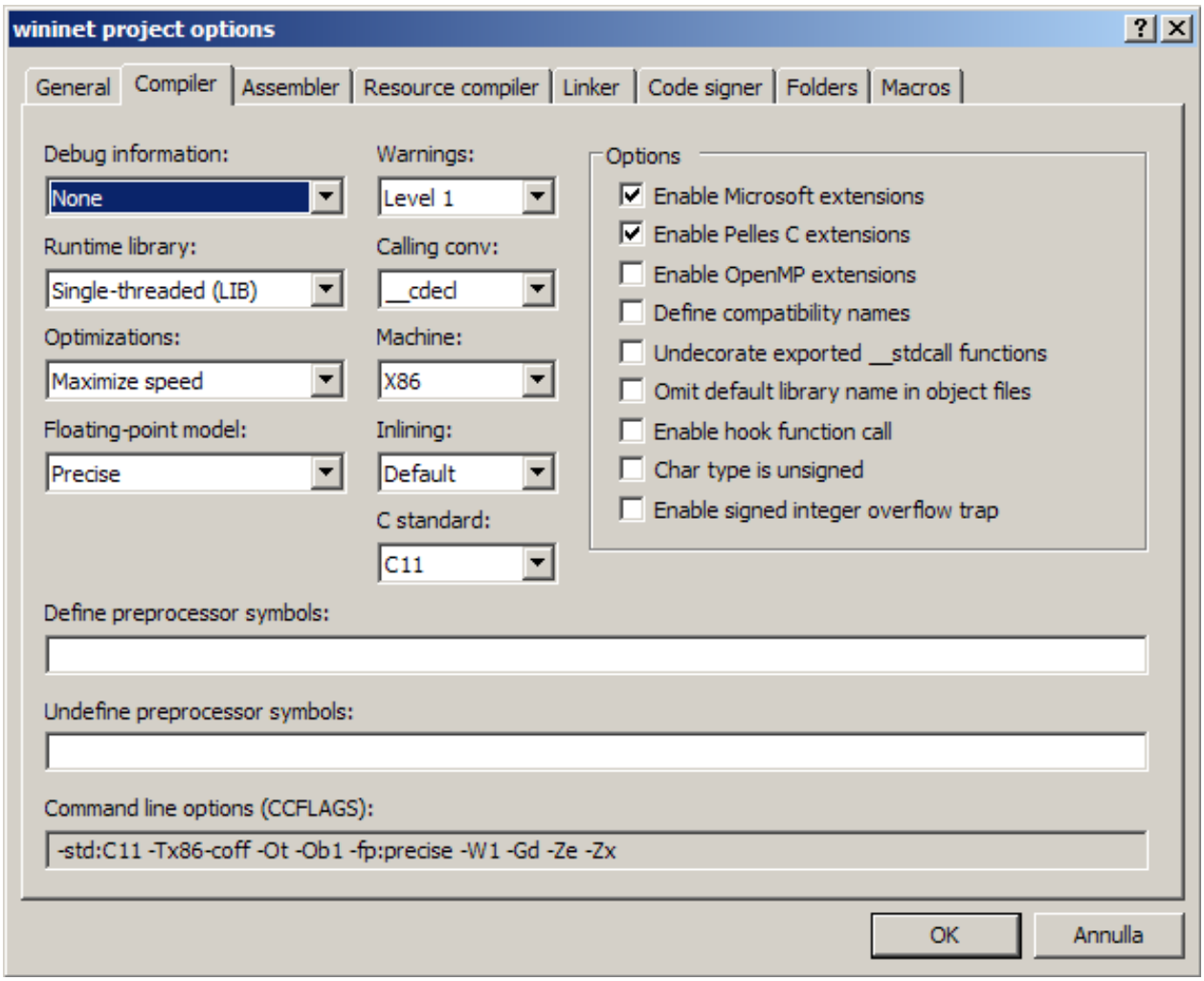
Все исходные коды, предоставленные в примерах выше, были скомпилированы с помощью компилятора Pelles C версии 8.00.11 для 32-битных систем.

Опции Pelles Projects

Используемые опции компилятора Pelles представлены на следующих рисунках:

Компилирующие опции:

Выбранный стандарт C – «C 2011», политика вызова – «cdecl».

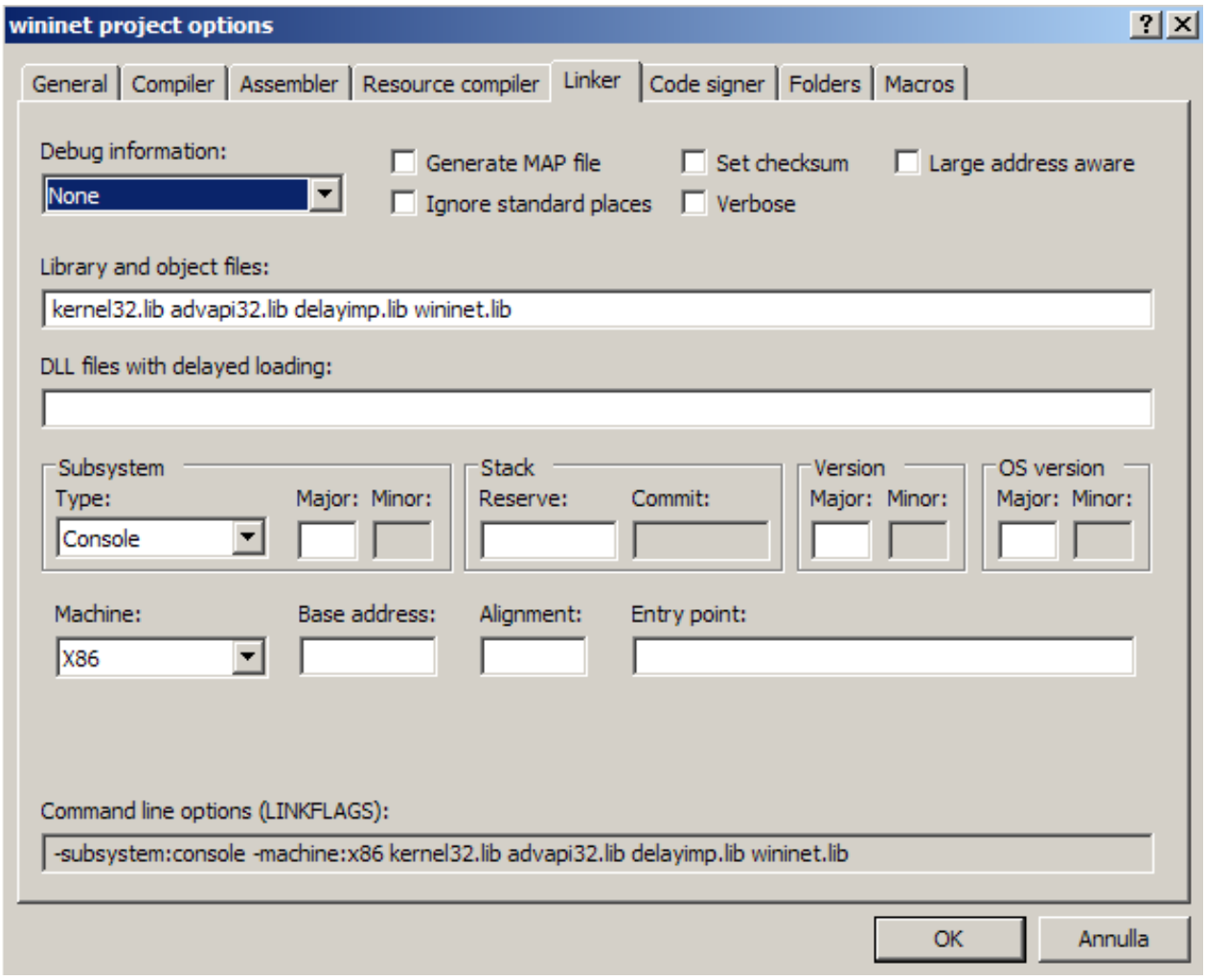


Опции компоновщика

На рисунке ниже показаны настройки компоновщика. В частности, тут мы видим прописанные опции для «LIBRARY AND OBJECT FILES» (для кода из примера 3).

Обратите внимание, что для правильного воссоздания примеров из данной статьи, для компиляции исполняемых файлов нужно использовать следующие библиотеки и объектные файлы:

- Пример 1: kernel32.lib, advapi32.lib, delayimp.lib, shell32.lib
- Пример 2: kernel32.lib, advapi32.lib, delayimp.lib, WS32_2.lib
- Пример 3: kernel32.lib, advapi32.lib, delayimp.lib, wininet.lib


wininet project options
General
Compiler
Assembler
Resource compiler
Linker
Code signer
Folders
Macros
Debug information:
None
Generate MAP file
Set checksum
Large address aware
Ignore standard places
Verbose
Library and object files:
kernel32.lib advapi32.lib delayimp.lib wininet.lib
DLL files with delayed loading:
Subsystem
Type:
Console
Major:
Minor:
Stack
Reserve:
Commit:
Version
Major:
Minor:
OS version
Major:
Minor:
Machine:
X86
Base address:
Alignment:
Entry point:
Command line options (LINKFLAGS):
-subsystem:console -machine:x86 kernel32.lib advapi32.lib delayimp.lib wininet.lib
OK
Annulla

Ссылки

Примечание переводчика: здесь я приведу все ссылки из статьи на отдельной странице, мало ли что ☺

1) Novkovic, Igor. Immunity Debugger basics, part 1. Blog. Student blog sgrosstudents.blogspot.ca. May 22, 2014. Retrieved Feb 15th, 2015.

http://sgros-students.blogspot.ca/2014/05/immunity-debugger-basics-part-1.html

2) Intel x86 JUMP quick reference, (n.d.), retrieved 04/25/2016, from the Steve Friedl's Unixwiz.Net Tech Tips.

http://unixwiz.net/techtips/x86-jumps.html

3) Frink, Lyle, "Unpacking" the Unitrix Malware, Avast! Blog, Sept. 7th, 2011, Retrieved Feb. 9th, 2016

https://blog.avast.com/2011/09/07/unpacking-the-unitrix-malware/

4) CryptoWall .aaa Extension Ransomware Removal Guide , Blog. Deletemalware.blogspot.co.uk, Aug. 6th, 2015, Retrieved Mar. 1st, 2016,

http://deletemalware.blogspot.co.uk/2015/08/cryptowall-aaa-extension-ransomware.html

5) Steane, Andrew M, Quick introduction to Windows API, Exeter College, Oxford University and Centre for Quantum Computing , 2009, Retrieved Mar. 2nd, 2016

https://users.physics.ox.ac.uk/~Steane/cpp_help/winapi_intro.htm

6) MoveFile Function, (n.d.), retrieved 04/25/2016, from Microsoft MSDN

https://msdn.microsoft.com/it-it/library/windows/desktop/aa365239%28v=vs.85%29.aspx

7) ShellExecute Function, (n.d.), retrieved 04/25/2016, from Microsoft MSDN

https://msdn.microsoft.com/en-us/library/windows/desktop/bb762153%28v=vs.85%29.aspx

8) Crt0, Wikipedia, (n.d.), retrieved Mar 4th, 2016

https://en.wikipedia.org/wiki/Crt0

9) Cdecl calling convention, (n.d.), retrieved 04/25/2016, from Microsoft MSDN

https://msdn.microsoft.com/en-us/library/zkwh89ks.aspx

10) Creating a Basic Winsock Application, (n.d.), retrieved 04/25/2016, from Microsoft MSDN

https://msdn.microsoft.com/it-it/library/windows/desktop/ms737629(v=vs.85).aspx

11) Taking Object Ownership in C++, retrieved 04/25/2016, from Microsoft MSDN

https://msdn.microsoft.com/it-it/library/windows/desktop/aa379620(v=vs.85).aspx

12) Load IE Symbols in Immunity Debugger, (May 28th, 2015), retrieved 04/25/2016, from ReverseEngineering – Stackexchange.com

http://reverseengineering.stackexchange.com/questions/9006/load-ie-symbols-in-immunity-debugger

13) Finding Main() – Compiler Code vs. Developer Code, (October 18th, 2007), retrieved 04/15/2016, from the Ethical Hacker Network

https://www.ethicalhacker.net/columns/heffner/intro-to-reverse-engineering-part-2#findingmain

14) Htons() function description, (n.d.), retrieved 04/25/2016, from the Beej's Guide to Network Programming

http://beej.us/guide/bgnet/output/html/multipage/htonsman.html

15) RtlIpv4StringToAddress Function, (n.d.), retrieved 04/25/2016, from Microsoft MSDN

https://msdn.microsoft.com/it-it/library/windows/desktop/aa814458%28v=vs.85%29.aspx

16) Wininet Reference, (n.d.), retrieved 04/25/2016, from Microsoft MSDN

https://msdn.microsoft.com/en-us/library/windows/desktop/aa385483(v=vs.85).aspx

Использованная литература

- Petzold, Charles (2011*), Programming Windows – 5th Edition, The Definitive Guide to programming Windows API*, Microsoft Press
- Eilam, Eldad (2005), *Reversing – Secrets of Reverse Engineering*, Wiley Publishing
- M. Sikorski, A. Honig (2012), *Practical Malware Analysis*, No Starch Press