Норсеев Сергей Norseev@gmail.com

РУЧНАЯ РЕАНИМАЦИЯ ДАМПА ПАМЯТИ

Руководство по ручному восстановлению памяти

Для автоматизации распаковки программ создано немало различных утилит. Но ни одна из них не дает стопроцентную гарантию решения поставленной перед ней задачи. Поэтому полагаться приходится только на себя. Если снятие дампа памяти, как правило, не вызывает проблем, то при реанимации этого дампа (придании ему работоспособного состояния) мы полагаемся на программы, которые могут и не сработать. Как же вернуть дамп к жизни в этом случае?

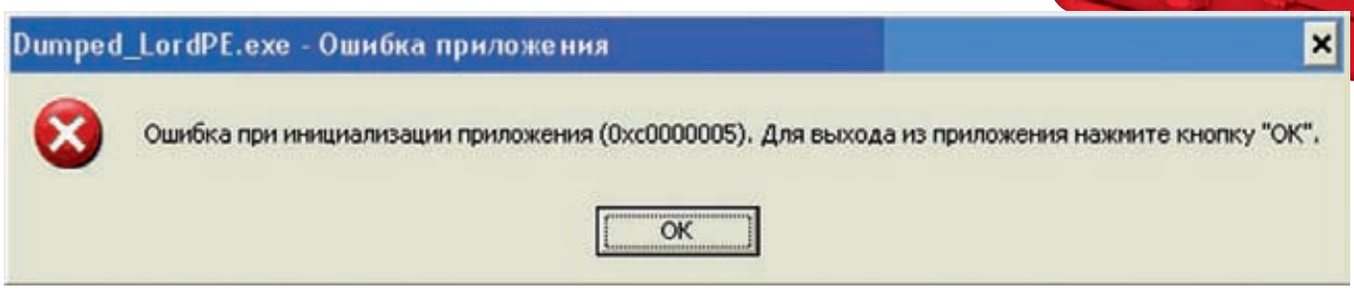
Введение, или зачем все это надо

Представь себе такую ситуацию (наверняка каждый был в ней, и не раз): решил ты вручную распаковать какую-то программу, нашел OEP, зациклил программу, снял дамп памяти и ... дамп не работает! Причина здесь вполне очевидна — накрылась таблица импорта. В принципе, попробовать восстановить ее можно и с помощью ImpRec, очень неплохой программы, восстанавливающей импорт (точнее, пытающейся сделать это). Но бывают случаи, когда ImpRec восстанавливает (если, конечно, вообще что-то восстанавливает) не всю таблицу импорта, а, в лучшем случае, только ее часть. При таком раскладе мы оказываемся один на один со снятым дампом. И что делать? Как быть? На самом деле, восстановление таблицы импорта — не такая уж и сложная задача (в большинстве случаев), как кажется. Сейчас я расскажу о том, как это сделать с помощью подручных средств (отладчика, Hex-редактора и редактора заголовка PE-файлов).

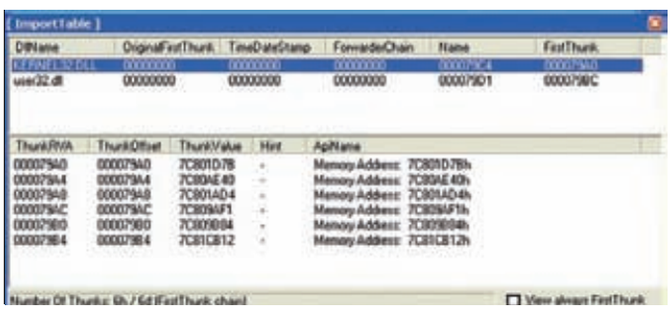
Строение таблицы импорта

Таблицу импорта описывает первый (считая от нуля) элемент массива DataDirectory. Ее адрес (здесь и далее под словом «адрес» мы будем подразумевать RVA-адрес) хранится по смещению 80h от начала PE-заголовка файла. Сама таблица представляет собой массив структур IMAGE_IMPORT_DESCRIPTOR, вот ее прототип:

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



К такому сообщению приводит попытка запуска дампа



Так выглядит сдампенная таблица импорта с перезаписанным массивом FirstThunk

Сам массив заканчивается структурой IMAGE_IMPORT_DESCRIPTOR, все поля которой равны нулю.
Из всех полей этой структуры нас интересует интересуют только два:
• Name — указывает на строку с именем библиотеки;
• FirstThunk — указывает на массив структур IMAGE_THUNK_DATA32.
Остальные поля могут быть пустыми.
Структура IMAGE_THUNK_DATA32 имеет следующий прототип:

```
struct IMAGE_THUNK_DATA32 {
    union {
        DWORD ForwarderString;
        DWORD Function;
        DWORD Ordinal;
        DWORD AddressOfData;
    } u1;
}
```

Единственное поле этой структуры указывает на строку с именем импортируемой функции за вычетом двух байт (в них хранится ординал функции). Заканчивается массив нулевым элементом.

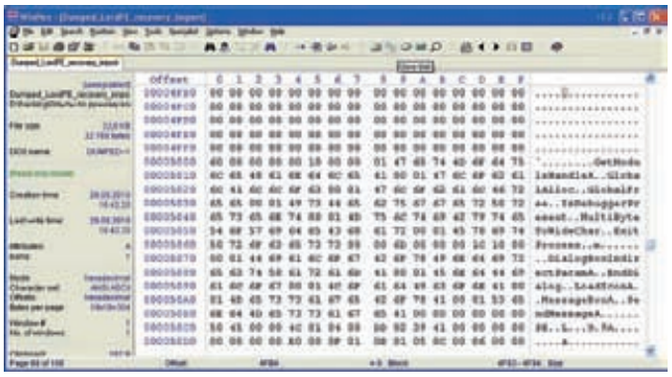
Когда происходит загрузка PE-файла, загрузчик разбирает массив структур IMAGE_IMPORT_DESCRIPTOR, загружает в память процесса соответствующие библиотеки (находит их по полю Name) и перезаполняет массив FirstThunk (по этой причине он должен располагаться в области памяти, доступной как на чтение, так и на запись). В этом массиве вместо имен функций (или их ординалов, в случае импортирования по ординалу) оказываются записанными адреса соответствующих функций. Именно через массив FirstThunk и происходит вызов API-функций.

Почему сдампенный импорт не работает?

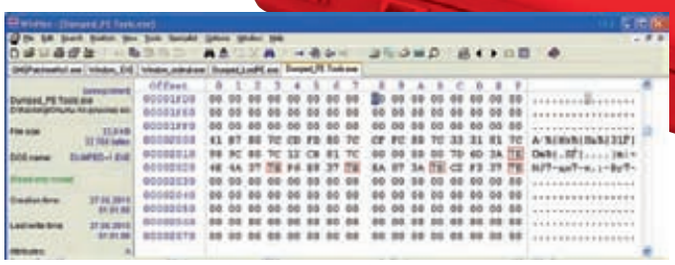
Предположим, что у нас есть дамп памяти некоего процесса с уже восстановленной точкой входа, который наотрез отказывается запускаться как самостоятельный exe-файл. В чем причина? В дампе таблица импорта выглядит примерно так, как показано на рисунке. Тебя не смущает значение поля ThunkValue? Оно ведь вроде как должно указывать на имя импортируемой функции. На самом деле произошло следующее: когда мы запустили зацикленную упакованную программу с целью снять с нее дамп памяти, загрузчик переписал содержимое массива FirstThunk адресами импортируемых функций, а дампер снял дамп памяти, как он есть. То есть сейчас содержимое массива FirstThunk указывает не на имя (или ординал) импортируемой функции, а на ее непосредственный адрес. Что происходит при попытке запустить этот файл на исполнение?
Происходит следующее:
1) Загрузчик по массиву DataDirectory находит таблицу импорта;.
2) Из структуры IMAGE_IMPORT_DESCRIPTOR извлекает имя загружаемой библиотеки и адрес массива FirstThunk;.
3) Анализируя массив FirstThunk, загрузчик, в надежде найти имя импортируемой функции, обращается по адресу, указанному в этом массиве, и находит там... невыделенную область памяти (сама библиотека загружается позже), обращение к которой вызывает исключение access violation, с последующим обламыванием всего процесса загрузки и выводом соответствующего ругательного сообщения.
Выходит, что восстановление таблицы импорта в большинстве случаев сводится к восстановлению массива FirstThunk. Для того, чтобы его восстановить, нужно знать имена (ординалы) импортируемых функций, а также адреса, по которым должны быть записаны адреса этих функций.

Реконструкция импорта

Прежде чем начать непосредственное восстановление таблицы импорта, нам нужно собрать всю необходимую информацию о составе и месторасположении массива FirstThunk. Обращаю внимание на то, что в сдампенном приложении можно обнаружить следы двух таблиц импорта: первая используется распаковщиком (и, как правило, именно на нее первоначально указывает массив DataDirectory), вторая используется самим упакованным приложением (она-то нас и интересует).
Как их различить? Во-первых, таблица импорта распаковщика не отличается большим разнообразием; она, как правило, значительно меньше импорта обычного приложения и не содержит в себе никаких функций работы с окнами и сообщениями, типа GetMessage, DispatchMessage, CreateWindow и т.д. (распаковщику они просто не нужны). Во-вторых, таблица импорта приложения размещена значительно ближе к самому приложению и дальше от кода распаковщика. Гораздо сложнее, когда программа упакована несколько раз. В этом случае перечень функций упакованной программы (которая представляет собой второй упаковщик) и распаковщика очень похож или вообще полностью идентичен.
В этом случае порядок действий такой: встаем отладчиком в начало распаковщика и смотрим, какая из двух таблиц присутствует в памяти (таблица импорта конечной программы еще не



Список импортируемых API-функций виден невооруженным глазом



Цепочка байтов 7Eh с шагом 4h демаскирует FirstThunk

распакована, поэтому в памяти ее нет). В 99,9 % случаев список импортируемых функций виден, что называется, невооруженным глазом в любом HEX-редакторе. Если же функции импортируются по ординалам, то получить полный список импортируемых функций не так просто. Проблема усугубляется тем фактом, что имеющиеся ординалы оказываются затертыми адресами функций. В этом случае единственным способом обнаружения функций является нахождение массива FirstThunk по записанным в нем адресам. Как ты знаешь, все PE-файлы имеют так называемый базовый адрес загрузки. Виртуальный адрес функции рассчитывается путем сложения этого базового адреса и относительного виртуального адреса функции. Так вот, на системах без ASLR этот базовый адрес постоянен, а при ASLR постоянен только его старший байт. Поскольку RVA адреса меньше 01000000h, то старший байт постоянен для данной библиотеки. Этим мы и воспользуемся для поиска импортируемых функций. Последовательность действий при этом такая:

1) Определяем используемые библиотеки (их имена хранятся в дампе открытым текстом и видны невооруженным глазом).

2) Находим для них старший байт базового адреса загрузки (например, для библиотеки user32 — 7Eh).

3) Ищем в снятом дампе вхождения этого дампа; цепочка таких байтов с шагом 4h и будет основным признаком массива FirstThunk.

Резонно возникает вопрос: «А почему для поиска функций не воспользоваться API-монитором?». Дело в том, что при анализе дампов многократно упакованной программы мы можем «промахнуться». Поясню на примере: пусть у нас есть какая-то программа, ее упаковывают пакером A, затем пакером B, ну и на погоны — пакером C. Предположим, что мы снимаем пакер C, соответственно, нам нужны функции, используемые B. API-монитор не различает, кто вызывает функцию — упакованная программа или кто-то из распаковщиков. Поэтому мы вполне можем впасть в заблуждение и начать восстанавливать таблицу импорта основной программы, не сняв всех пакеров, что ни к чему хорошему нас не приведет.

Другой вариант поиска массива FirstThunk основан на так называемом «переходнике». Дело в том, что в большинстве программ API-функции вызываются не напрямую, а через так называемый «переходник», который представляет из себя простую команду jmp на вызываемую функцию (при этом направление, куда «прыгать», берется из массива FirstThunk). Пример такого «переходника» представлен на рисунке.

Данный «переходник» выдает нам массив FirstThunk со всем его содержимым.

Хуже, когда такого «переходника» нет, то есть библиотечные функции вызываются напрямую. В этом случае его приходится искать самим. Порядок примерно следующий:

1) Находим место вызова любой библиотечной функции. Наиболее разумный для этого способ — установка точки останова на функцию и эксплуатация программы до тех пор, пока отладчик не всплывет на ней;.

2) Определяем, откуда берется адрес вызываемой функции. Если речь идет не о неявном вызове API-функций через ручной расчет их адресов (что встречается крайне редко), то браться он будет из массива FirstThunk. Выглядеть он будет примерно так, как показано на рисунке. При этом здесь перед нами будут адреса всех импортируемых функций из разных библиотек. То есть перед нами не один, а несколько массивов FirstThunk (каждый из них соответствует своей библиотеке), разделенных между собой нулевым двойным словом.

3) Осталось лишь сопоставить адреса, записанные в этот массив, с соответствующими им функциям. Сделать это можно прямо в отладчике.

Создание таблицы импорта

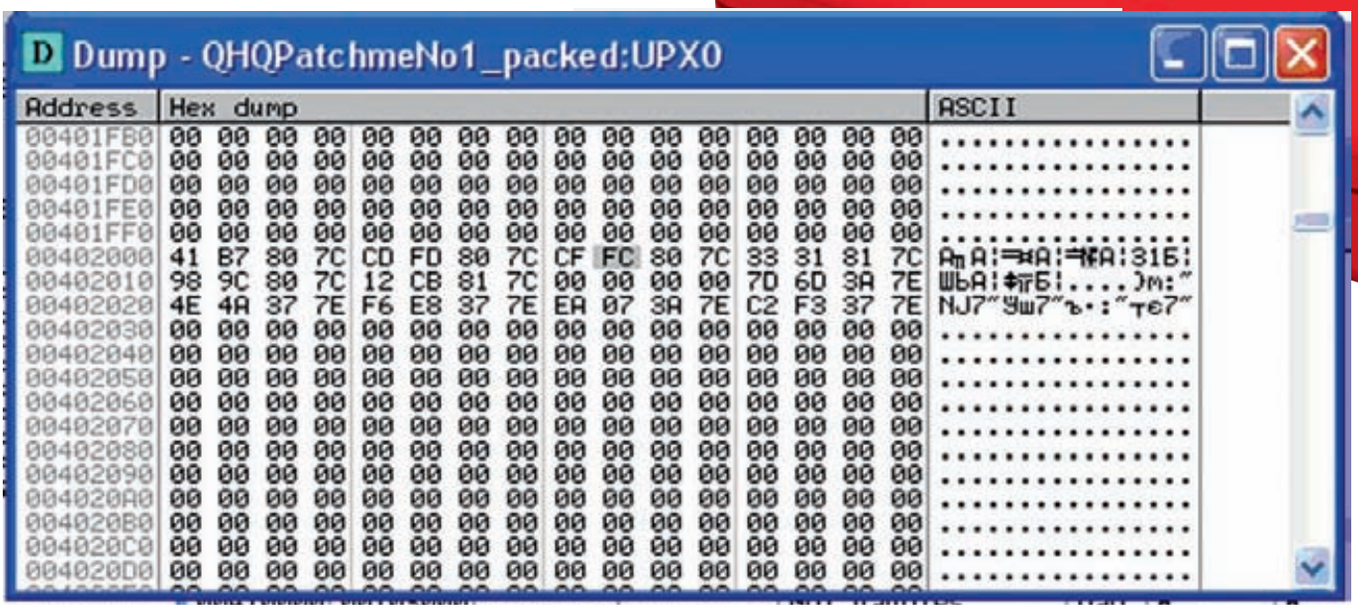
После того, как у нас на руках окажется вся необходимая информация, мы можем приступить к созданию (именно «созданию», потому что после распаковки массив структур IMAGE_IMPORT_DECRIPTOR исходного приложения полностью отсутствует) таблицы импорта. Сам этот массив может быть размещен в любом месте программы, доступном на чтение. Разместим его по адресу 2040h (за массивами FirstThunk). Пропускаем первые Ch байт (в них идут первые три поля IMAGE_IMPORT_DESCRIPTOR, которые мы не будем заполнять). По адресу 204Ch пишем адрес строки «kernel32.dll» (находим ее в дампе или создаем самостоятельно). В следующие 4 байта записываем найденный адрес массива FirstThunk с функциями из библиотеки kernel32 (у нас 2000h). Теперь перезаполняем массив FirstThunk (напоминаю, что каждый элемент этого массива представляет собой двойное слово, содержащее адрес строки с именем функции, за вычетом двойки). В нашем примере первый элемент содержит адрес функции GetModuleHandleA, строка с ее именем размещена по адресу 5009h, вычитаем 2, получаем 5007h. Это значение и пишем в массив FirstThunk.

Именно по такому принципу мы и восстановим всю таблицу импорта. Самое главное при этом — не потереть важные для программы данные (строки, ресурсы, переменные). Поэтому располагать таблицу импорта лучше всего в области, состоящей из нулей. Вообще, для размещения массива IMAGE_IMPORT_DESCRIPTOR (массив FirstThunk и имя библиотеки у нас уже есть), описывающего N библиотек, нужно (N+1)*14h байт памяти.

Нам осталось только изменить адрес таблицы импорта в массиве DataDirectory и наслаждаться приложением с восстановленной таблицей.

Ресурсы

А вот с ресурсами не все так однозначно. Дело в том, что восстанавливать их не требуется, так как они уже исправны (содержатся в дампе в своем первозданном виде). Но почему тогда их нельзя отредактировать или хотя бы просмотреть ни одним редактором ресурсов? Происходит это, потому что ни один из известных мне редакторов не умеет работать с ресурсами, расположенными в



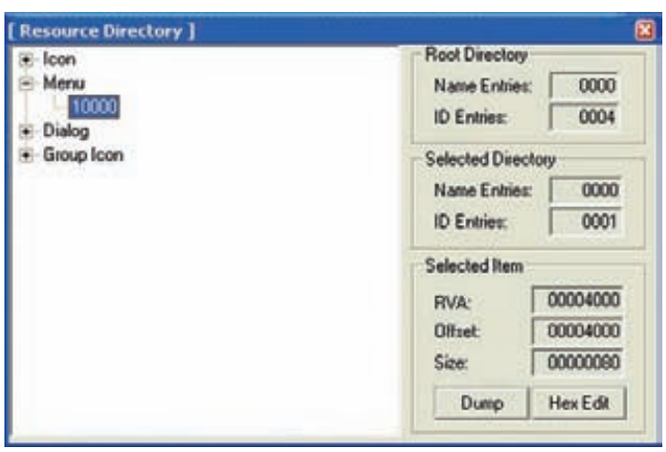
Массив FirstThunk в модифицированном загрузчиком виде

Переходник, через который и осуществляется вызов API-функций

двух или более секциях (кстати, хороший способ спрятать ресурсы от любопытных), а в снятом дампе они расположены именно так. На работоспособность снятого дампа это никак не влияет, так как таблица ресурсов полностью исправна и указывает на имеющиеся ресурсы. Убедиться в этом можно с помощью LordPE. Если же тебе кровь из носу нужна возможность редактирования ресурсов, то воспользуйся программой Resource Binder, которая создает новую секцию и помещает в нее все найденные ресурсы.

Заключение

Как видно ручная реанимация дампа памяти — не такая уж и сложная задача, вполне осуществимая при наличии внимательности, головы и прямых рук. Надеюсь, теперь ручная распаковка программ станет для тебя еще проще.



LordPE показывает нам структуру ресурсов и их RVA-адреса

Определяем имя функции по ее адресу