Анализ таблицы экспорта

Вредоносные файлы с экспортируемыми функциями — редкое явление, однако иногда на просторах вирусных полей они встречаются. Бывают в том числе и вредоносные DLL.

Таблицу экспорта, в которой перечислены экспортируемые функции, можно так же, как и таблицу импорта, посмотреть в CFF Explorer или Hiew.

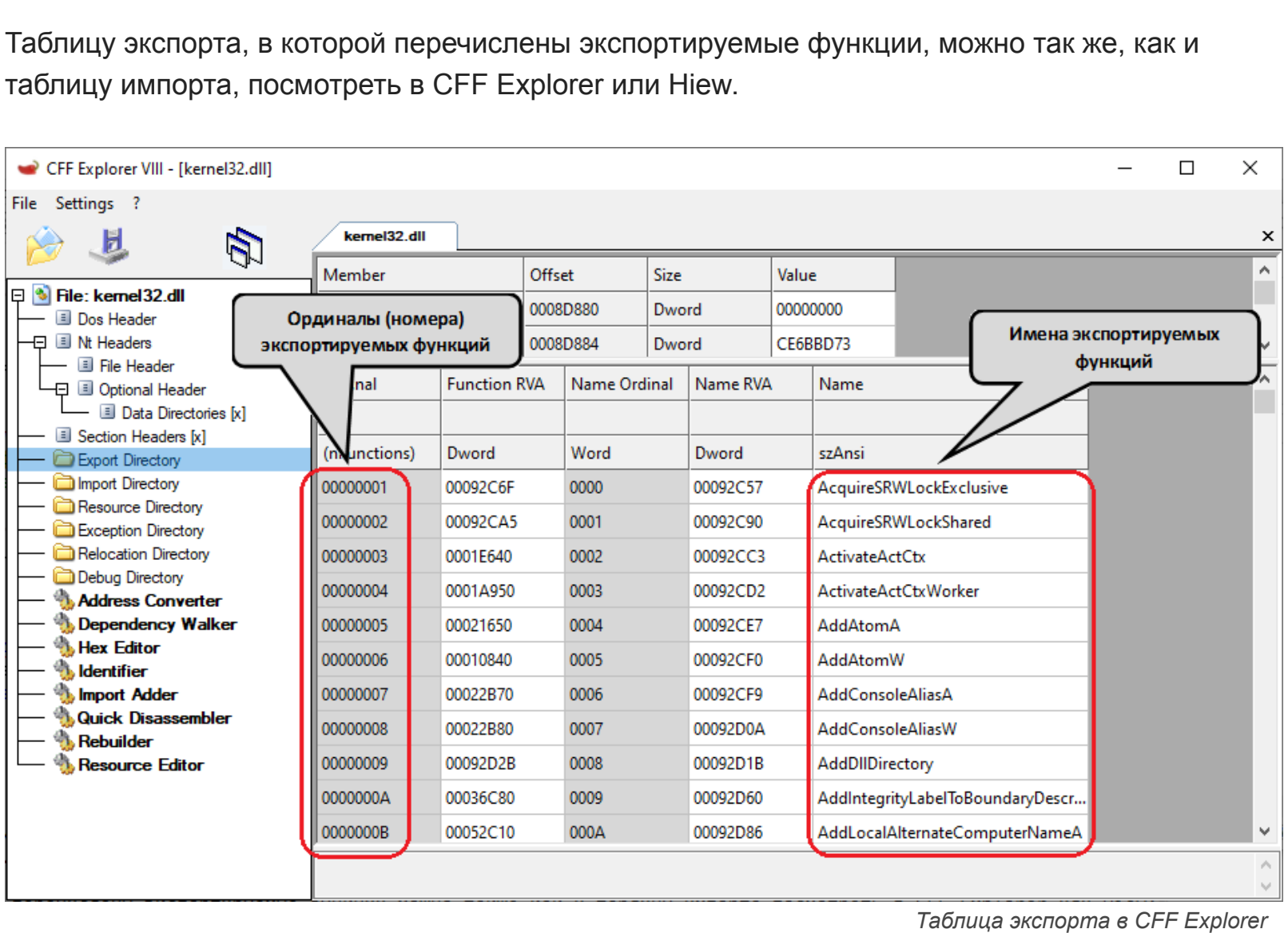


Таблица экспорта в CFF Explorer

Здесь нужно обратить внимание на говорящие сами за себя названия экспортируемых функций, либо на названия функций в виде беспорядочного набора символов, либо на отсутствие названий функций при наличии только ординалов (номеров) функций (такое тоже может быть, ведь функции можно экспортировать не только по имени, но и по ординалу).

На Python можно сделать следующий скрипт для просмотра таблицы экспорта:

```
import pefile
## не забудь указать реальный путь к исследуемому файлу
pe = pefile.PE(<путь к файлу>)
if hasattr(pe, 'DIRECTORY_ENTRY_EXPORT'):
  for export_entry in pe.DIRECTORY_ENTRY_EXPORT.symbols:
    print('\t' + export_entry.name.decode('utf-8'))
    print('\t\tОрдинал:', str(hex(export_entry.ordinal)))
    print('\t\tRVA функции:', str(hex(export_entry.address)))
else:
  print('Файл не содержит секцию экспорта.')
```

Анализ таблицы секций

Фактически все содержимое PE-файла разбито на секции. Каждая секция хранит в себе либо код, который будет исполнен при запуске файла, либо данные, необходимые для выполнения, либо ресурсы, используемые файлом. У каждой секции есть имя, однако операционная система определяет назначение секции не по имени, а по атрибуту `Characteristics`, а имена используются только для наглядности.

Помимо назначения секции атрибут Characteristics определяет операции, которые можно проводить с данными секции (чтение, выполнение, запись и т. д.). Более подробно о секции в целом и об этом атрибуте можно почитать в официальной документации. Наиболее часто используемые названия секций:

- `.text` или `CODE` — как правило, содержит исполняемый код (название секции `CODE` характерно для программ, написанных на Delphi), в большинстве случаев имеет значение атрибута Characteristics, равное `0x60000020` (`IMAGE_SCN_CNT_CODE & IMAGE_SCN_MEM_EXECUTE & IMAGE_SCN_MEM_READ`);
- `.data` или `DATA` — обычно здесь лежат данные для чтения или записи (название секции `DATA` также характерно для программ, написанных на Delphi), Characteristics чаще всего равен `0xc0000040` (`IMAGE_SCN_CNT_INITIALIZED_DATA & IMAGE_SCN_MEM_READ & IMAGE_SCN_MEM_WRITE`);
- `.rdata` — данные только для чтения, иногда здесь лежат таблицы импорта и экспорта, Characteristics равен `0x40000040` (`IMAGE_SCN_CNT_INITIALIZED_DATA & IMAGE_SCN_MEM_READ`) или `0x50000040` (`IMAGE_SCN_CNT_INITIALIZED_DATA & IMAGE_SCN_MEM_READ & IMAGE_SCN_MEM_SHARED`);
- `.idata` — информация об импорте (если секция отсутствует, то импорт содержится в секции `.rdata`), Characteristics чаще всего равен `0xc0000040` (такое же, как и у секции `.data`);
- `.edata` — информация об экспорте (если секция отсутствует, то экспорт содержится в секции `.rdata`), Characteristics обычно равен `0xc0000040` (такое же, как и у секции `.data`);
- `.rsrc` — ресурсы, используемые PE-файлом (иконки, диалоговые окна, меню, строки и т. д.), Characteristics равен `0x50000040` либо `0x40000040`.

Также могут встречаться секции `.tls`, `.reloc`, `.pdata`, `.bss` (или `BSS`), `.debug`, `.CRT`, `.didat`. Наличие секции `BSS` вместе с секциями `CODE` и `DATA` означает, что автор программы использовал компилятор Delphi, а если есть секции `.bss` и `.CRT`, это может означать, что программу компилировали в MinGW.

Увидеть это все легко можно с помощью, например, CFF Explorer.

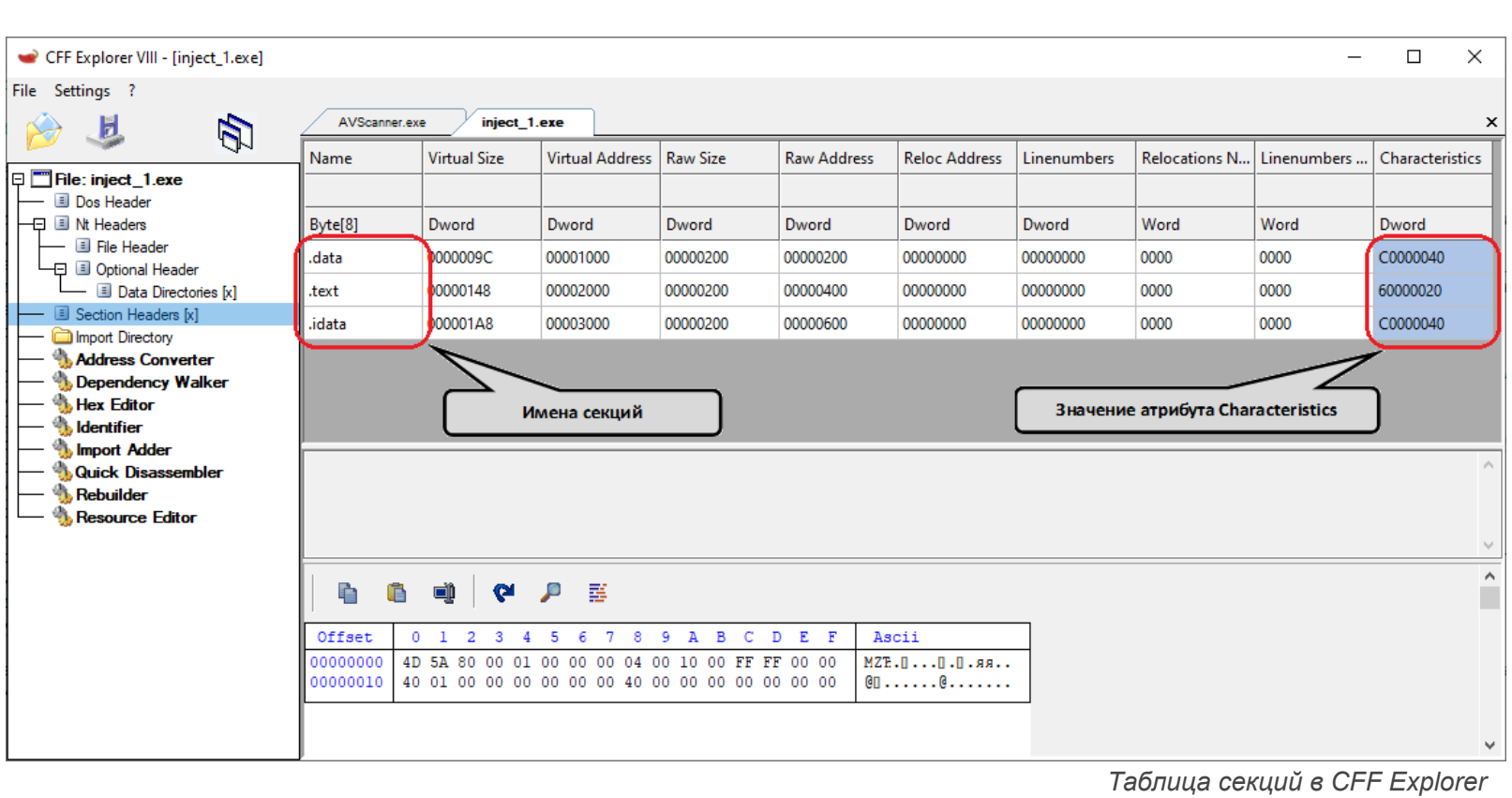


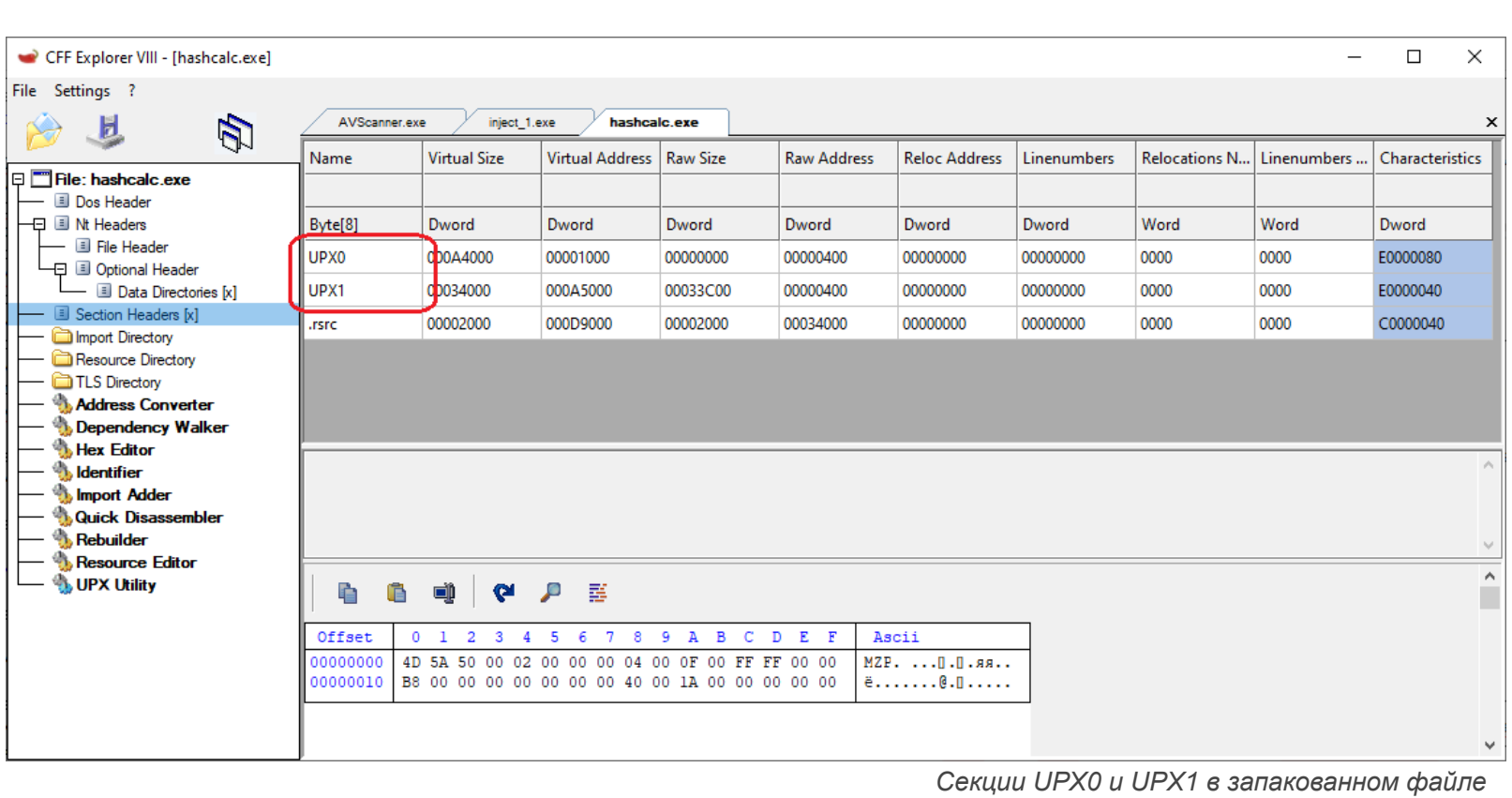
Таблица секций в CFF Explorer

На что нужно обратить внимание при анализе таблицы секций?

Во-первых, на необычные названия секций. Например, в виде беспорядочного набора символов.

Во-вторых, на несоответствие атрибута Characteristics назначению и содержимому секции. Например, может быть так, что у секции `.text`, в которой содержится исполняемый код, помимо прочего еще добавлено `IMAGE_SCN_MEM_WRITE`, то есть в секцию с кодом можно писать данные. Это явный признак самомодифицирующегося кода, и в обычных программах такое почти не встречается. То же можно сказать и про секцию с данными (`.data` или `DATA`): если в атрибуте Characteristics помимо прочего присутствует `IMAGE_SCN_MEM_EXECUTE`, то это очень серьезный повод для подозрения.

В-третьих, наличие секций вроде `UPX0`, `UPX1` или `.aspack` красноречиво свидетельствует о применении соответствующих упаковщиков.

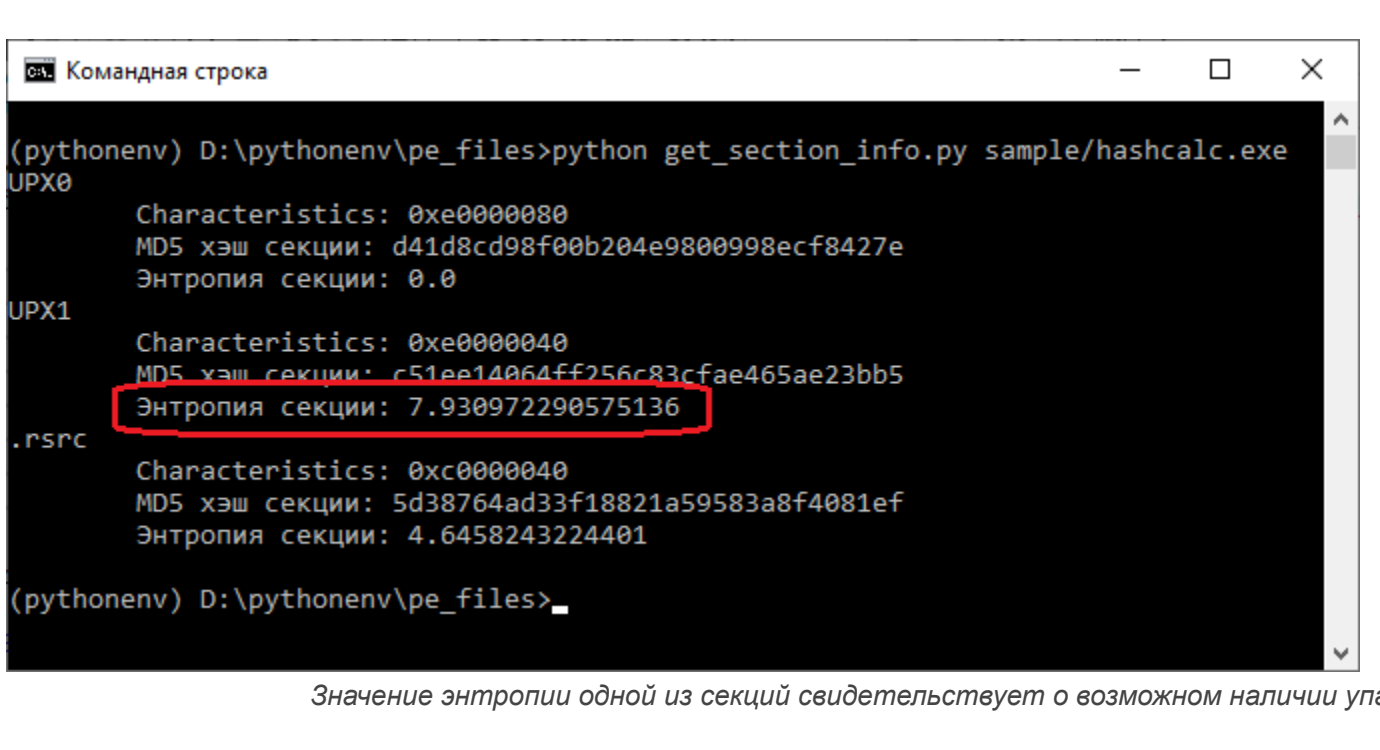


Секции UPX0 и UPX1 в запакованном файле

Также о применении упаковщиков может свидетельствовать высокое значение (близкое к 8) энтропии какой-либо секции. Посчитать энтропию секций PE-файла можно с помощью вот такого скрипта на Python:

```
import pefile
## не забудь указать реальный путь к исследуемому файлу
pe = pefile.PE(<путь к файлу>)
for section_entry in pe.sections:
  print(section_entry.Name.decode('utf-8'))
  print('\tCharacteristics:', hex(section_entry.Characteristics))
  print('\tMD5 хэш секции:', section_entry.get_hash_md5())
  print('\tЭнтропия секции:', section_entry.get_entropy())
```

Скрипт выводит имена всех секций PE-файла, значение атрибута Characteristics, md5-хеш и энтропию каждой секции.

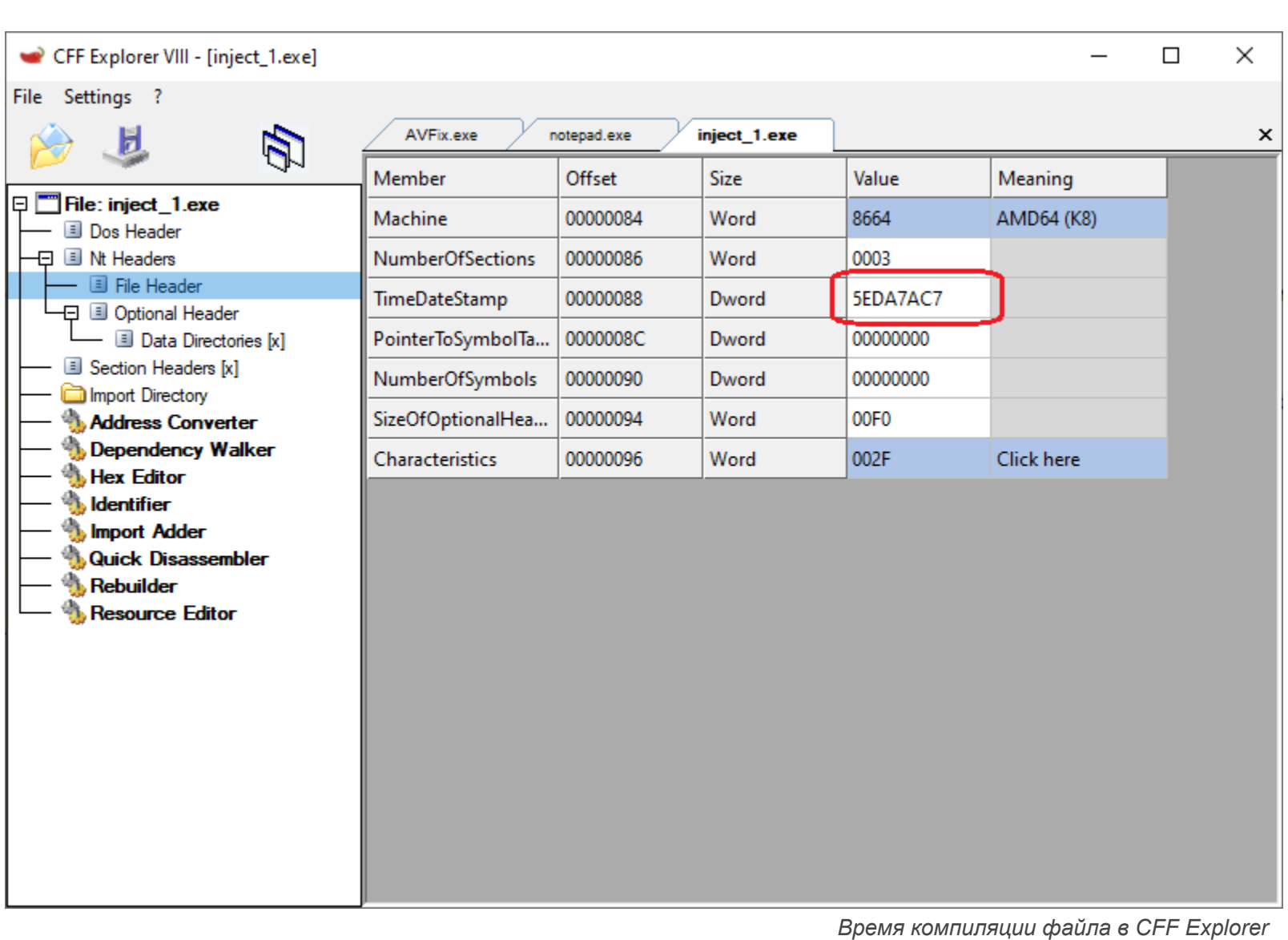


Значение энтропии одной из секций свидетельствует о возможном наличии упаковки

В Yara энтропию можно определить с помощью функции `entropy()` модуля math.

Временная метка компиляции

Информация о времени компиляции анализируемого файла может быть полезна при построении графа атаки и его анализе. Время создания (и, соответственно, компиляции PE-файла) хранится в PE-заголовке в виде четырехбайтового числа, содержащего количество секунд, прошедших с 0 часов 0 минут 1 января 1970 года.



Время компиляции файла в CFF Explorer

В удобоваримом виде значение этого времени можно посмотреть с помощью такого скрипта:

```
import pefile
## не забудь указать реальный путь к исследуемому файлу
pe = pefile.PE(<путь к файлу>)
print('Дата и время компиляции:', time.strftime('%Y-%m-%d %H:%M:%S', time.gmtime(pe.FILE
```

Для Delphi (а вредоносные файлы, написанные на Delphi, встречаются и по сей день) это значение всегда равно `0x2a425e19`, что значит 0 часов 0 минут 19 июня 1992 года. В этом случае реальную дату компиляции можно попытаться определить из отметки времени секции `.rsrc` (в файлах, создаваемых Delphi, она всегда присутствует). Временная метка находится по смещению 4 от начала секции `.rsrc` и представляет собой четырехбайтовое число в формате MS-DOS time.

Посмотреть это значение в пристойном виде можно следующим скриптом:

```
import pefile
## не забудь указать реальный путь к исследуемому файлу
pe = pefile.PE(<путь к файлу>)
time_stamp_dos = pe.DIRECTORY_ENTRY_RESOURCE.struct.TimeDateStamp
## преобразуем время из MS DOS формата в «нормальный» вид
day = time_stamp_dos >> 16 & 0x1f
month = time_stamp_dos >> 21 & 0x7
year = (time_stamp_dos >> 25 & 0xff) + 1980
second = (time_stamp_dos & 0x1f) * 2
minute = time_stamp_dos >> 5 & 0x3f
hour = time_stamp_dos >> 11 & 0x1f
print('Дата и время компиляции: {}-{}-{} {:02d}:{:02d}:{:02d}'.format(day, month, year,
```

Если PE-файл компилировался Visual Studio и при компиляции в файл была включена отладочная информация (а это можно определить по наличию таблицы Debug Directory в PE-файле), то дата компиляции (помимо заголовка PE-файла) также содержится и в этой таблице:

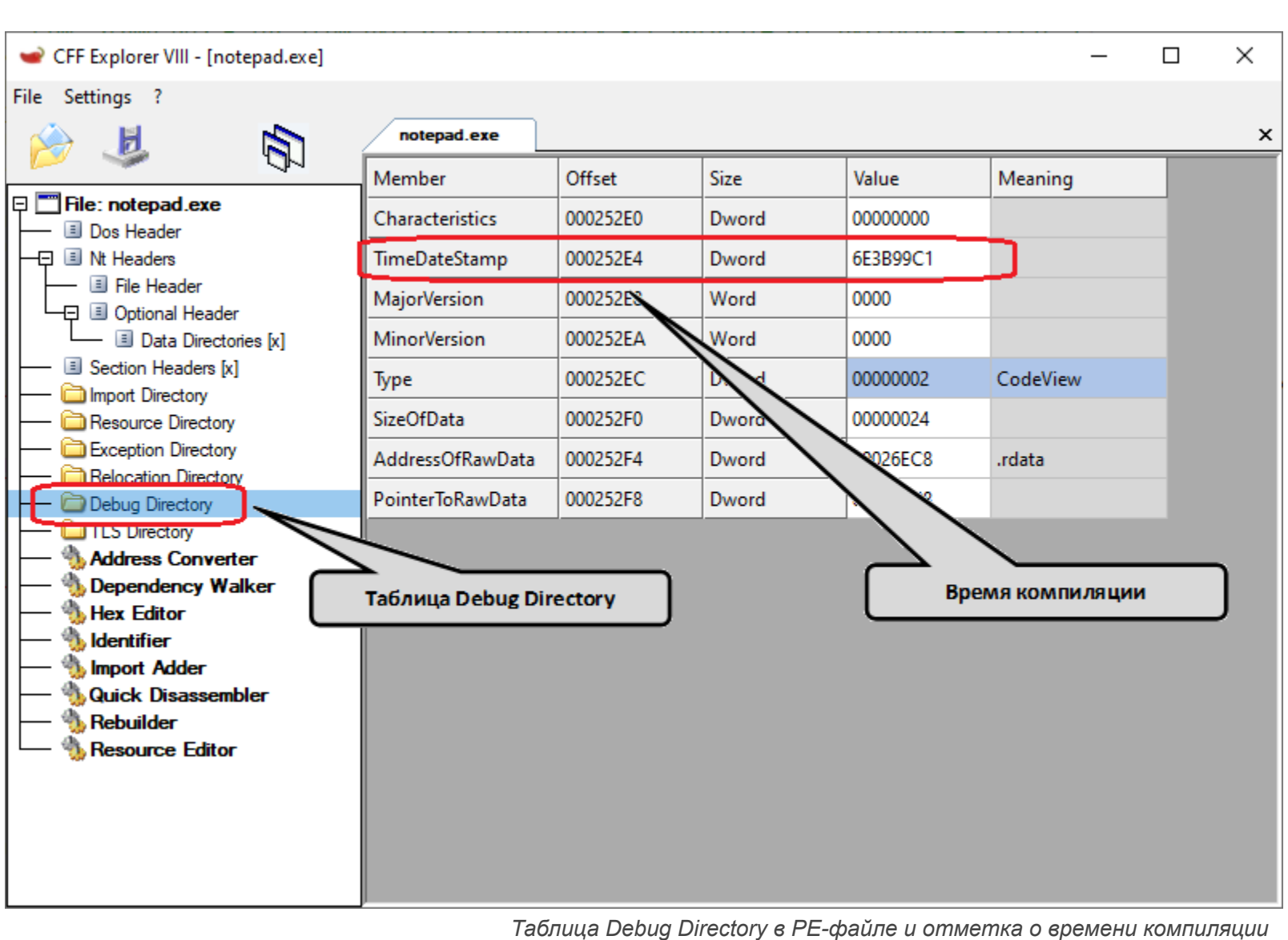


Таблица Debug Directory в PE-файле и отметка о времени компиляции

Посмотреть временную метку компиляции из Debug Directory в «нормальном» виде можно следующим скриптом:

```
import pefile
## не забудь указать реальный путь к исследуемому файлу
pe = pefile.PE(<путь к файлу>)
time_stamp = pe.DIRECTORY_ENTRY_DEBUG[0].struct.TimeDateStamp
print('Дата и время компиляции:', time.strftime('%d-%m-%Y %H:%M:%S', time.gmtime(time_st
```

При анализе временной метки компиляции под подозрением должны быть:

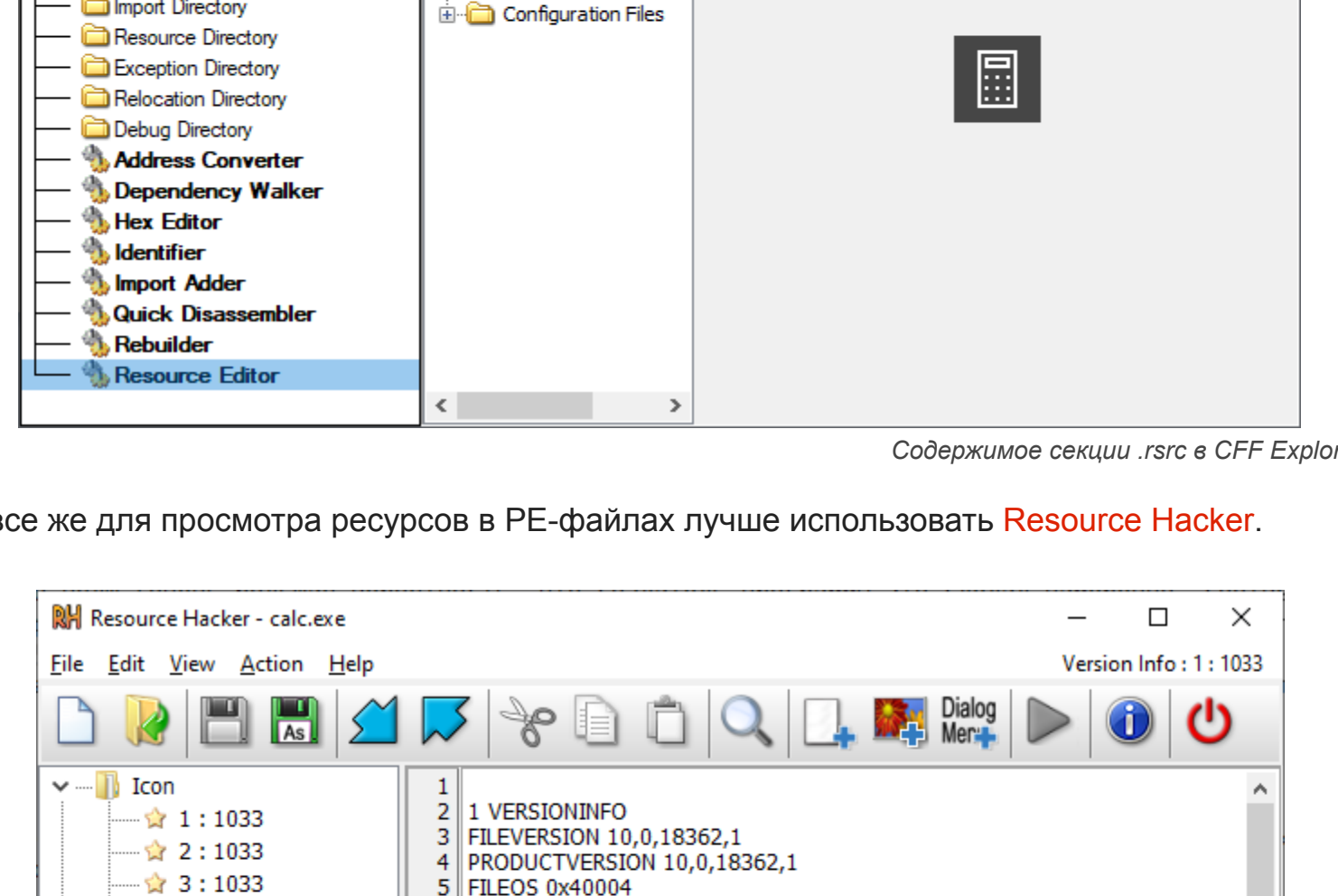
- недавняя дата компиляции программы (или нереальные ее значения, например еще не наступившая дата);
- несовпадение даты компиляции и версии компилятора (согласись, странно видеть дату компиляции, например, 20 июня 2005 года для экзешника, откомпилированного Visual Studio 19 версии);
- значение даты компиляции равно нулю (велика вероятность, что создатель программы это сделал намеренно, соответственно возникает вопрос — зачем);
- для файлов, которые по всем признакам откомпилированы в Delphi, дата не соответствует значению 0x2a425e19, а дата, полученная из секции `.rsrc`, равна нулю или меньше, чем дата в PE-заголовке;
- дата компиляции из заголовка анализируемого файла не совпадает с датой, указанной в Debug Directory (весьма вероятно, что эти значения были зачем-то скорректированы).

INFO

У некоторых компонентов Windows поле `TimeDateStamp` в PE-заголовке может иметь интересные значения — либо из будущего (к примеру, у меня для «Блокнота» это 8 сентября 2028 года), либо из прошлого (встречаются компоненты, датированные 1980 годом).

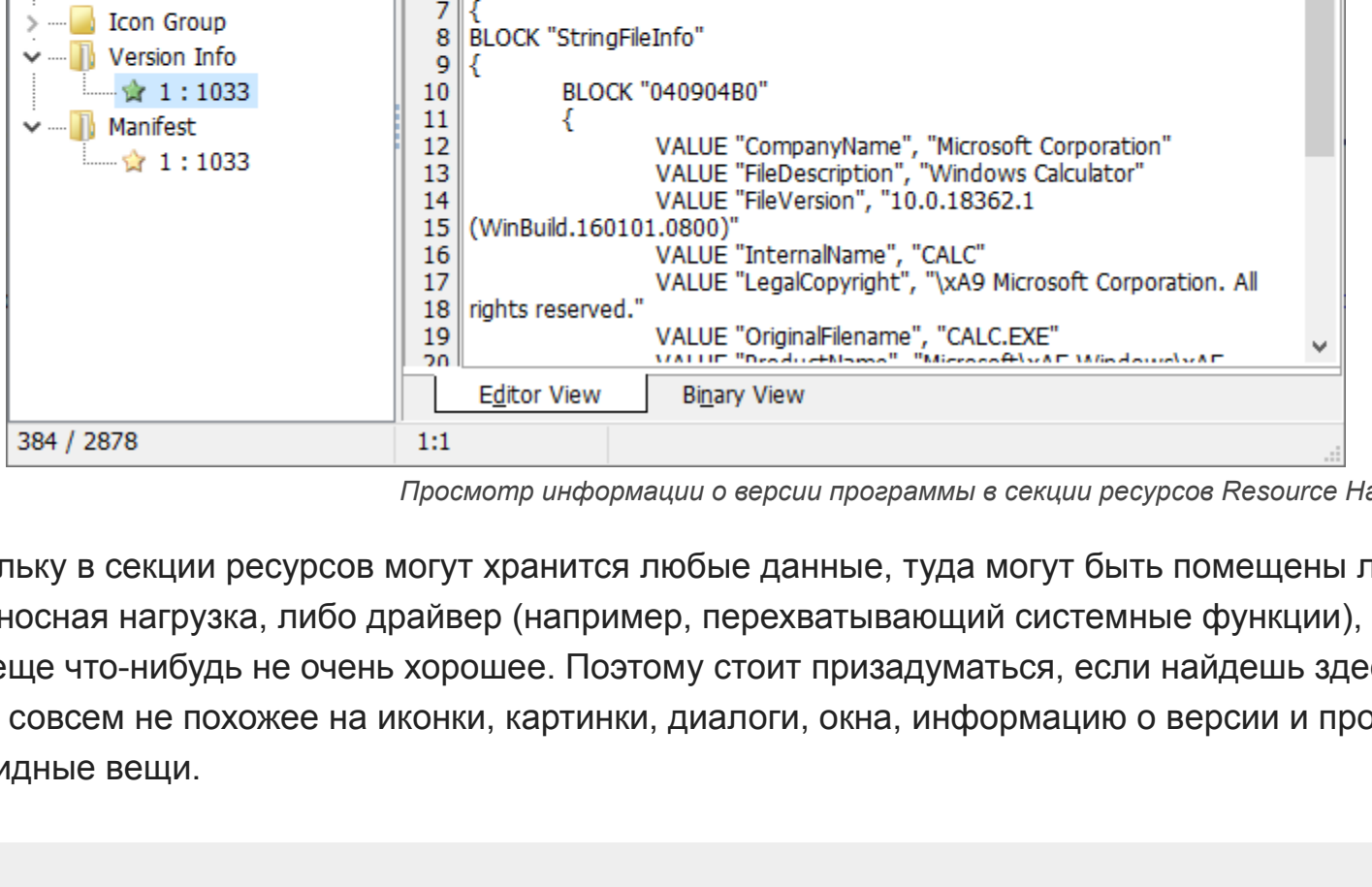
Анализ ресурсов исполняемого файла

Ресурсы, необходимые для работы exe-файла (такие как иконки, диалоговые окна, меню, изображения, информация о версии, конфигурационные данные), хранятся в секции `.rsrc`. Просмотреть можно при помощи все того же CFF Explorer.



Содержимое секции .rsrc в CFF Explorer

Но все же для просмотра ресурсов в PE-файлах лучше использовать Resource Hacker.



Просмотр информации о версии программы в секции ресурсов Resource Hacker

Поскольку в секции ресурсов могут хранится любые данные, туда могут быть помещены либо вредоносная нагрузка, либо драйвер (например, перехватывающий системные функции), либо еще что-нибудь не очень хорошее. Поэтому стоит призадуматься, если найдешь здесь что-то совсем не похожее на иконки, картинки, диалоги, окна, информацию о версии и прочие безобидные вещи.

Rich-сигнатура

В статье я не упомянул такую примечательную структуру PE-файла, как Rich-сигнатура. На нее тоже стоит обратить внимание при анализе подозрительных файлов. Почитать о ней на английском можно здесь:

- Rich Headers: Leveraging this Mysterious Artifact of the PE Format (PDF)
- Microsoft's Rich Signature (NTCore)

Также я написал скрипт на Python для просмотра Rich-сигнатуры.

Заключение

С помощью нескольких относительно простых инструментов мы можем быстро проанализировать подозрительный исполняемый файл, частично (а иной раз и достаточно полно) понять, что он собой представляет, и вынести вердикт о возможной вредоносности. Однако при анализе сложных файлов, подвергнутых упаковке, шифрованию и обфускации, этого может оказаться недостаточно. Тогда без более детального статического и динамического анализа не обойтись. Об этом мы и поговорим в следующий раз.

WWW

Все исходники на Python, приведенные в статье;
YaraEditor и YaraGui — программы, которые упрощают работу с Yara;
Утилиты для подсчета контрольных сумм: HashMyFiles, Hash Calculator, Hash Calc.