

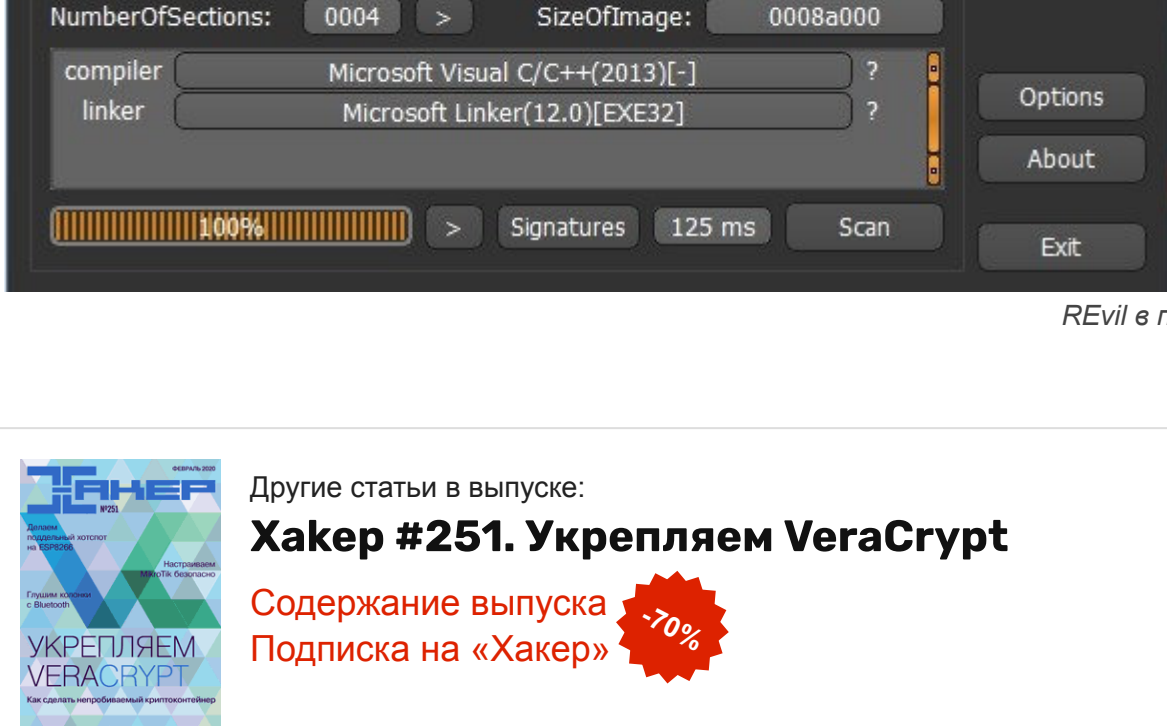
Разбираем REvil. Как известный шифровальщик прячет вызовы WinAPI

Nik Zerot, 13.02.2020 1 комментарий 22,805 Добавить в закладки

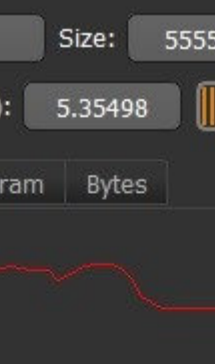


Не так давно атаке подверглась международная система денежных переводов Travelex, и виновником этого оказался шифровальщик REvil, чем и привлек мое внимание. Забегая вперед, скажу, что в этом трояне использованы простые, но эффективные методы обфускации, которые не позволяют нам так просто увидеть используемые им вызовы WinAPI. Давай посмотрим, как устроен этот анкердер изнутри.

По доброй традиции загрузим семпл в DiE и поглядим, что он нам покажет.



REvil в приложении DiE

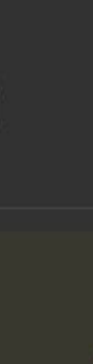


Другие статьи в выпуске:

Хакер #251. Укрепляем VeraCrypt

Содержание выпуска

Подписка на «Хакер»



DiE считает, что файл ничем не упакован. Хотя постоной-ка, давай переключимся на показания энтропии секций.



Энтропия секций REvil

Судя по названиям секций, файл упакован UPX, но их энтропия выглядит очень странно. Почему тогда DiE не распознал упаковщик? Ну, например, сигнатура UPX может быть намеренно искажена, чтобы запутать дизассемблеры. Так или иначе, перед нами упакованный файл, поэтому загружаемся в отладчик x64dbg. Давай поставим точку останова на функцию VirtualAlloc, которая мелькает у нас в окрестностях точки входа, и запустим троян.



INFO

Есть несколько функций WinAPI, бряки на которые нужно устанавливать по умолчанию при распаковке неизвестного пакера, ибо механизмы распаковки достаточно стандартны:

- VirtualAlloc — используется для выделения памяти для пейлоада;
- VirtualProtect — используется для установки атрибутов доступа к памяти;
- CreateProcessInternalW — при создании нового процесса, в эту функцию в итоге передается управление;
- ResumeThread — используется для продолжения выполнения при инъекциях.

Брыкаемся на функции и выходим из нее в наш код. В итоге видим такую картину:

```
008F9552 | FF55 B4 | call dword ptr ss:[ebp-4C] | VirtualAlloc
008F9555 | 8945 F0 | mov dword ptr ss:[ebp-10],eax | <--- мы находимся здесь
008F9558 | 8365 DC 00 | and dword ptr ss:[ebp-24],0 |
008F955C | 8B85 58FFFFFF | mov eax,dword ptr ss:[ebp-A8] |
008F9562 | 0FB640 01 | movzx eax,byte ptr ds:[eax+1] |
```

Осматриваемся дальше, видим интересный кусок кода в конце функции, в которой мы оказались:

```
00569C10 | 8985 5CFFFFFF | mov dword ptr ss:[ebp-A4],eax |
00569C16 | 8B85 5CFFFFFF | mov eax,dword ptr ss:[ebp-A4] |
00569C1C | 0385 68FFFFFF | add eax,dword ptr ss:[ebp-98] |
00569C22 | C9 | leave |
00569C23 | FF80 | jmp eax | Интересный переход!
```

Не забываем: при отработке функции VirtualAlloc адрес выделенной памяти находится в eax. Ставим точку останова на этот переход, попутно переходим на дамп (адрес в eax) и смотрим, что будет происходить в выделенной памяти. Для этого ставим на начале памяти однократную точку останова на запись, и отладчик останавливается на цикле записи данных в память. Вот так выглядит часть цикла:

```
00279DA4 | 8A11 | mov dl,byte ptr ds:[ecx] |
00279DA6 | 8B10 | mov byte ptr ds:[eax],dl |
00279DA8 | 40 | inc eax |
00279DA9 | 41 | inc ecx |
00279DAA | 4F | dec edi |
00279DAB | 75 F7 | jne 279DA4 |
```

Если мы станем вручную прокручивать цикл, в памяти начнет проявляться до боли знакомая сигнатура:

```
003C0000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....уу..
```

Отпускаем отладчик и останавливаемся на jmp eax, делаем шаг вперед — и мы в распакованном файле! Теперь можно снимать дампы и загружать его в IDA Pro. Выполнив эту нехитрую процедуру, мы увидим код стартовой функции:

```
public start
start proc near
push 0
call sub_40369D
push 0
call sub_403EEF
pop ecx
ret
start endp
```

Исследование функций и возможностей нашего вредоноса надо с чего-то начинать, поэтому заходим в первый call. Там нас ждет уже более интересный код:

```
sub_40369D proc near
call sub_406A4D // Перед вызовом функции по хешу есть только одна подпрограмма; от
push 1
call dword_41CB64 // Хм, что это?
call sub_40489C
test eax, eax
jz short loc_4036BD
...
...
```

Видим вызов подпрограммы sub_406A4D, далее вызов такого рода: call dword_41CB64. Очевидно, что если все «костыли как есть», то здесь приложение упадет при выполнении, потому что dword_41CB64 ведет на таблицу такого рода (это только часть таблицы!):

```
.data:0041CB64 dword_41CB64 dd 40D32A7Dh ; DATA XREF: sub_40369D+7;r
.data:0041CB68 dword_41CB68 dd 0C97676C4h ; DATA XREF: sub_403EE1+6;r
.data:0041CB6C dword_41CB6C dd 0D69D6931h ; DATA XREF: sub_403BC0+15;r
.data:0041CB70 dword_41CB70 dd 8AABE016h ; DATA XREF: sub_406299+C0;r
...
...
```

Кроме того, в нашем образе таблица импорта пустая: разумеется, функции WinAPI получают динамически, имена функций не хранятся в открытом виде, и, похоже, используются их хеши. На самом деле «Хакер» уже описывал подобную технику обфускации вызовов WinAPI, следовательно, нашим постоянным читателям будет проще разобраться в устройстве REvil. Итак, ныряем в функцию sub_406A4D, видим там один безусловный переход и следуем дальше в sub_405BCD. Практически в начале функции видим очень интересный код:

```
loc_405BD6:
// Кладем на стек элемент из таблицы хешей, на которую указывает ESI
push dword_41C9F8[esi]
// Работаем над этими данными, обработанное значение вернется в EAX
call sub_405DCF
// Возвращаем обратно
mov dword_41C9F8[esi], eax
// Идем по списку дальше (шагом по четыре байта)
add esi, 4
pop ecx
cmp esi, 230h
jnb short loc_405BD6
```

Разумеется, мы не можем не заглянуть в функцию sub_405DCF. Там мы видим целую портянку кода, поэтому придется переключиться в декомпилированный псевдокод, чтобы не портянугу в этом болоте с головой. Конечно, если ты гуру ассемблера и тебе не составляет труда читать много кода на этом языке, можешь оставить все как есть, а лично мне привычнее псевдокод IDA Pro.

Функция большая, поэтому полностью приводить ее в статье не имеет смысла, но мы можем сконцентрироваться на ее основных частях. Работу функции можно разделить на два этапа. Первый — трансформация имеющихся хеш-сумм, указанных в программе. Второй — получение из таблицы экспорта системных библиотек имен функций, хеширование и сверка с шаблонами, полученными из таблицы, которую мы уже видели.

Парсинг таблицы экспорта системной библиотеки на псевдокоде выглядит таким образом:

```
v17 = (IMAGE_EXPORT_DIRECTORY *) (v13 + *(_DWORD *) (*(_DWORD *) (v13 + 0x3C) + v13 + 0x78)
v21 = (int) v17->AddressOfNameOrdinals + v13;
v18 = (int) v17->AddressOfNames + v13;
v22 = (int) v17->AddressOfNames + v13;
v20 = (int) v17->AddressOfFunctions + v13;
v23 = v17->NumberOfNames;
if (1 < v23)
{
return 0;
}
while ( (sub_405BAE (v14 + *(_DWORD *) (v18 + 4 * v16)) & 0xFFFFFFFF) != v15 ) {
v18 = v22;
if ( ++v16 >= v23 )
return 0;
}
```

Почему именно этот кусок псевдокода привлек мое внимание? Разумеется, бросаются в глаза такие смещения, как 0x3C или 0x78. Кроме того, переменная v13, работающая с этими числами, приводится к типу DWORD*, говоря нам, что мы смотрим на некое смещение. Разумеется, все указывает на заголовок PE-файла:

```
0x00 WORD emagic Magic DOS signature MZ (0x4D 0x5A)
0x02 WORD e_chlp Bytes on last page of file
0x04 WORD e_cp Pages in file
0x06 WORD e_crlc Relocations
0x08 WORD e_cpahdr Size of header in paragraphs
0x0A WORD e_minalloc Minimum extra paragraphs needed
0x0C WORD e_maxalloc Maximum extra paragraphs needed
0x0E WORD e_ss Initial (relative) SS value
0x10 WORD e_sp Initial SP value
0x12 WORD e_csum Checksum
0x14 WORD e_ip Initial IP value
0x16 WORD e_cs Initial (relative) CS value
0x18 WORD e_lfarlc File address of relocation table
0x1A WORD e_ovno Overlay number
0x1C WORD e_res[4] Reserved words (4 WORDs)
0x24 WORD e_oemid OEM identifier (for e_oeminfo)
0x26 WORD e_oeminfo OEM information; e_oemid specific
0x28 WORD e_res2[10] Reserved words (10 WORDs)
0x3C DWORD e_lfanew Offset to start of PE header
```

В коде мы видим смещение 0x3C, которое соответствует полю e_lfanew. Двигаясь далее по e_lfanew по смещению 0x78 (смотрим псевдокод), мы видим вот такое поле:

```
0x78 DWORD Export Table RVA of Export Directory
```

Значит, идет разбор таблицы экспорта, что говорит о динамическом получении WinAPI.

Чтобы IDA Pro «понимала» структуру таблицы экспорта, ее необходимо объяснить в Local Types, нажав Shift + F1. После этого на переменной v17 нужно скопировать из контекстного меню Convert to struct*. Структура таблицы экспорта PE-файла выглядит таким образом:

```
struct IMAGE_EXPORT_DIRECTORY {
long Characteristics;
long TimeDateStamp;
short MajorVersion;
short MinorVersion;
long Name;
long Base;
long NumberOfFunctions;
long AddressOfNames;
long AddressOfFunctions;
long AddressOfNames;
long AddressOfNameOrdinals;
}
```

Как раз здесь мы видим используемые поля: *AddressOfFunctions, *AddressOfNames и *AddressOfNameOrdinals. По псевдокоду понятно, что хеши из уже имеющихся в коде получаются таким образом:

```
int __cdecl sub_405DCF(int (*a1)(void)) { // Передача аргумента
... // Много строк, которые можно пропустить
v1 = (unsigned int) a1 ^ (((unsigned int) a1 ^ 0x76C7) << 16) ^ 0xAFB9;
... //
v15 = v1 & 0xFFFFFFFF;
... //
}
```

Да, в теле семпла используются не «готовые» хеши, их еще предстоит привести в правильный вид. Если отбросить все лишнее, мы получим следующий алгоритм:

```
hash_api_true = (hash ^ ((hash ^ 0x76C7) << 16) ^ 0xAFB9) & 0xFFFFFFFF
```

где hash — переданный в качестве аргумента функции хеш из таблицы. Хорошо, что IDA подсвечивает одинаковые переменные, иначе анализ семпла занял бы намного больше времени. В псевдокоде этот хеш хранится в переменной под именем a1, которая является аргументом функции.

Если пропустить через этот алгоритм указанные в коде хеши (помнишь таблицу?), получаем «правильные» хеши, которые будут сравниваться с полученными из таблицы экспорта системной библиотеки, точнее, из имен экспортируемых функций. Псевдокод получения хеша из символического имени функции будет выглядеть на Python так:

```
def hash_from_name(name):
result = 0x2b
for x in name:
result = ord(x) + 0x10f * result
return result & 0xFFFFFFFF
```

Вызов функции:

```
hash_from_name(name) # Name — переменная, содержащая символическое имя функции
```

Итак, нам осталось лишь пропустить всю таблицу представленных в семпле псевдохешей через алгоритм hash_api_true и составить эту таблицу «правильных» хешей. Далее нужно пропустить WinAPI, состоящий из обычных символических имен, через алгоритм hash_from_name, получив хешированные имена. Заключительная часть — надо сопоставить эти два списка, таким образом декодируя имена и хеш-представления. Разумеется, удобнее всего это сделать с помощью питонового скрипта для IDA, а не вручную.

Можно ли сделать быстрее?

Конкретно в данном случае REvil строит таблицу деобфусцированных функций сразу целиком. Поэтому, загрузив семпл в отладчик, можно исполнить подпрограмму получения и деобфускации WinAPI-функций, после чего отладчик автоматом подставит уже расшифрованные имена функций API в код. Затем можно снять дампы и работать уже с ним, функции будут присутствовать на своих местах. Но этот способ подходит далеко не всегда. Например, если деобфускация выполняется не сразу со всем списком используемых функций, а с каждой функцией по отдельности, то мере ее вызова, этот прием уже не сработает. Кстати, именно так и было сделано в статье про обфускацию вызовов WinAPI.

Заключение

В этой статье мы разобрали, каким образом восстановить вызовы WinAPI, которые обфусцированы методом обращения по их хешам. Как видите, такой метод обфускации достаточно легко преодолевается при помощи отладчика или дизассемблера, при том что реверсера не останавливают математические манипуляции с хешем. Подобные вещи прекрасно видны в псевдокоде и замедляют исследование семпла разве что на пару минут.

На самом деле подобные приемы направлены на противодействие автоматическим системам анализа, а не человеку. Поэтому, если вы хотите защитить свой код от автоматического анализа, вам нужно использовать более сложные методы обфускации, которые не позволяют так просто увидеть используемые им вызовы WinAPI.