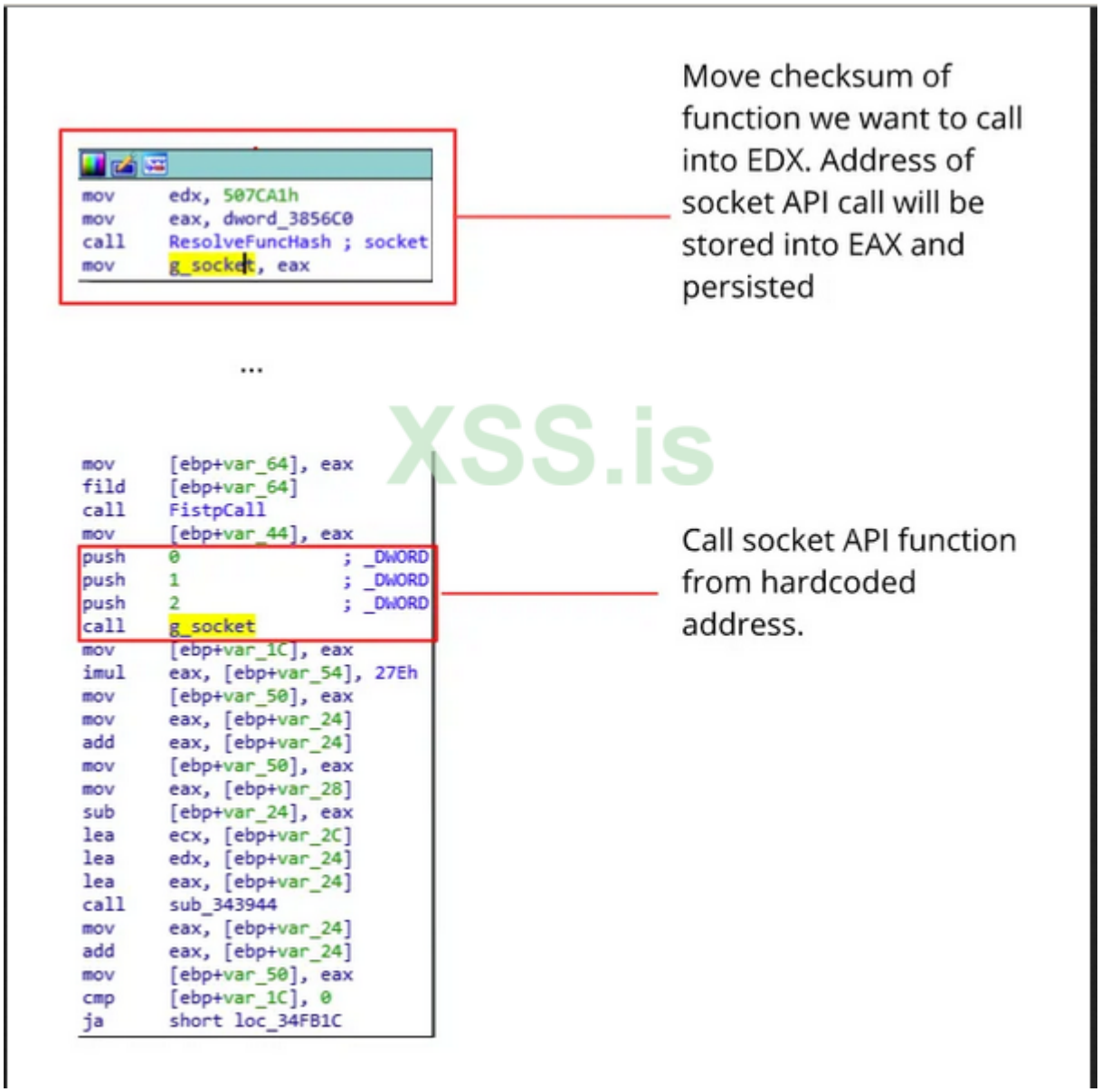
Статья Деобфускация хеширования API DanaBot

Вы, наверное, уже догадались по названию названия, API-хеширование используется для обфускации двоичного файла, чтобы скрыть имена API от инструментов статического анализа, что мешает реверс инженеру понять функциональность вредоносного ПО. Первый способ получить представление о функциональных возможностях исполняемого файла - это более или менее подробно изучить функции и найти вызовы API. Если, например, функция CreateFileW вызывается в определенной подпрограмме, это, вероятно, означает, что перекрестные ссылки или сама подпрограмма реализуют некоторые функции обработки файлов. Это будет невозможно, если используется хеширование API.

Вместо прямого вызова функции каждый вызов API имеет соответствующую контрольную сумму/хэш. Может быть получено жестко запрограммированное хеш-значение, и для каждой библиотечной функции вычисляется контрольная сумма. Если вычисленное значение совпадает с хеш-значением, с которым мы его сравниваем, мы нашли нашу цель.



В этом случае реверс-инженер должен выбрать другой путь для анализа двоичного файла или его деобфускации. В этой статье блога будет рассказано, как банковский троян DanaBot реализует хеширование API, и, возможно, самый простой способ борьбы с этим. SHA256 двоичного файла, который я здесь анализирую, добавлен в конце этого сообщения в блоге.

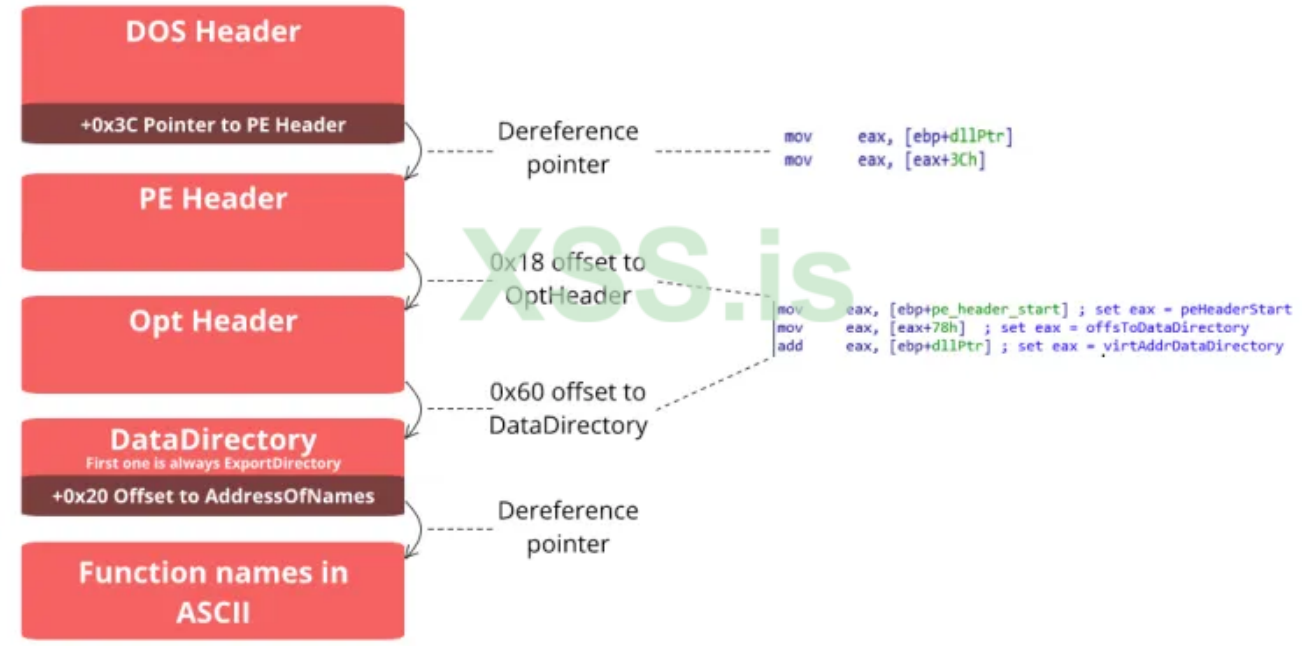
Глубокое погружение в DanaBot

Сам DanaBot является банковским трояном, существует как минимум с 2018 года и впервые был обнаружен ESET[1]. Стоит отметить, что он реализует большинство своих функций в плагинах, которые загружаются с сервера C2. Я сосредоточусь на деобфускации хеширования API на первом этапе DanaBot, DLL, которая дропается и сохраняется в системе, используемой для загрузки дополнительных плагинов.

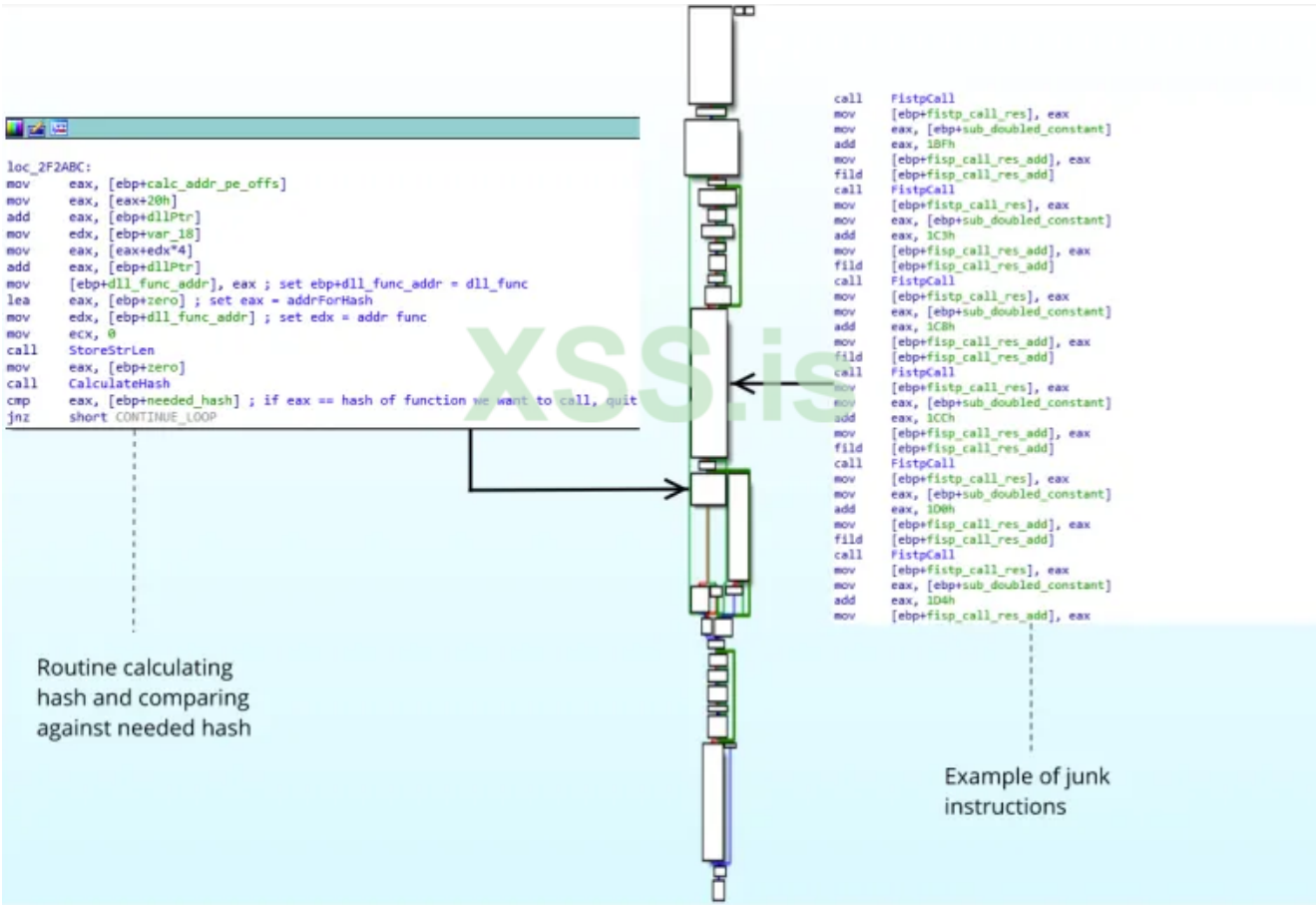
Реверсинг процедуры ResolvFuncHash

В начале функции регистр EAX хранит указатель на заголовок DOS динамически связанной библиотеки, который содержит функцию, которую двоичный файл хочет вызвать. Соответствующий хэш еще неизвестной функции API хранится в регистре EDX. Подпрограмма также содержит кучу ненужных инструкций, обфусцирующий фактический вариант использования этой функции.

Хэш вычисляется исключительно из имени функции, поэтому первым шагом является получение указателя на все имена функций целевой библиотеки. Каждая DLL содержит таблицу со всеми экспортируемыми функциями, которые загружаются в память. Этот Каталог Экспорта всегда является первой записью в ассиве Каталога Данных. Формат файла PE и его заголовки содержат достаточно информации, чтобы добраться до указанного каталога путем анализа структур заголовков:



На рисунке ниже вы можете увидеть пример упомянутых нежелательных инструкций, а также критический блок, который сравнивает вычисленный хэш с контрольной суммой функции, которую мы хотим вызвать. Подпрограмма выполняет итерацию по всем именам функций в каталоге экспорта и вычисляет хэш. Цикл прерывается, когда вычисленный хэш совпадает со значением, которое хранится в регистре EDX с начала этой процедуры.



Реверсинг алгоритма хеширования

Алгоритм хеширования довольно простой и не содержит ничего особенного. Нежелательные инструкции и непрозрачные предикаты усложняют процесс реверсинга этой процедуры.

Алгоритм берет n-й и stringLength-n-1-й символ имени функции и сохраняет их, а также версии с заглавной буквы в памяти, в результате чего получается 4 символа. Каждый из этих символов подвергается операции XOR с длиной строки. Наконец, они умножаются, и значения складываются каждый раз при запуске цикла, в результате чего получается хеш-значение.

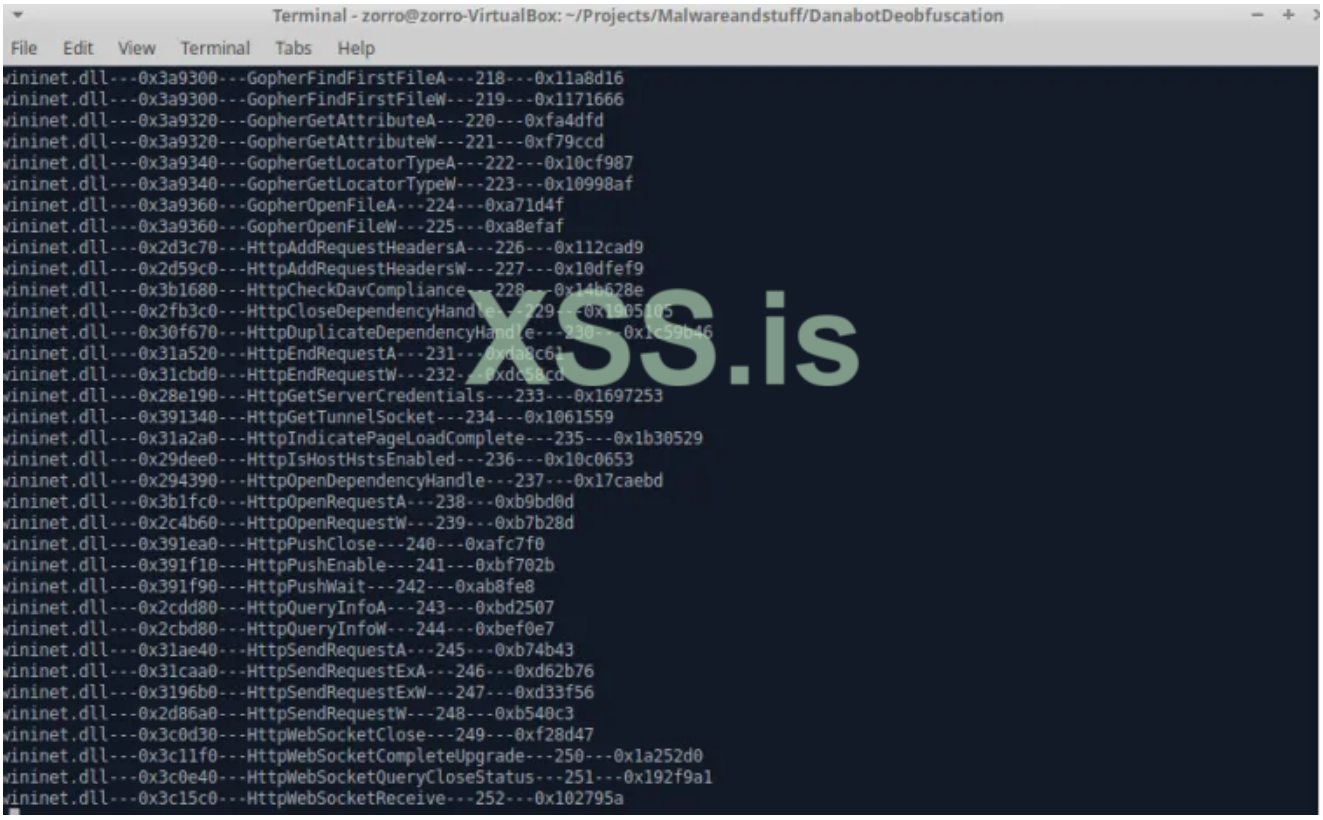
Python:

```
def get_hash(funcname):
    """Calculate the hash value for function name. Return hash value as integer"""
    strlen = len(funcname)
    # if the length is even, we encounter a different behaviour
    i = 0
    hashv = 0x0
    while i < strlen:
        if i == (strlen - 1):
            ch1 = funcname[0]
        else:
            ch1 = funcname[strlen - 2 - i]
        # init first character and capitalize it
        ch = funcname[i]
        uc_ch = ch.capitalize()
        # Capitalize the second character
        uc_ch1 = ch1.capitalize()
        # Calculate all XOR values
        xor_ch = ord(ch) ^ strlen
        xor_uc_ch = ord(uc_ch) ^ strlen
        xor_ch1 = ord(ch1) ^ strlen
        xor_uc_ch1 = ord(uc_ch1) ^ strlen
        # do the multiplication and XOR again with upper case character1
        hashv += ((xor_ch * xor_ch1) * xor_uc_ch)
        hashv = hashv ^ xor_uc_ch1
        i += 1
    return hashv
```

Сценарий python для вычисления хэша для заданного имени функции также загружен на мою страницу github и доступен всем желающим. Я также загрузил текстовый файл с хешами для экспортируемых функций часто используемых DLL.

Деобфускация с помощью комментирования

Итак, теперь, когда мы взломали алгоритм, мы хотим обновить наш дизассемблирный код, чтобы знать, какое хеш-значение представляет какую функцию. Как я уже упоминал, мы хотим сосредоточиться на простоте. Самый простой способ - вычислить хеш-значения для экспортируемых функций часто используемых DLL и записать их в файл.



С помощью этого файла мы можем написать сценарий IdaPython, чтобы прокомментировать имя библиотечной функции рядом с вызовом хеширования Api. К счастью, функция хеширования Api всегда вызывается по одному и тому же шаблону:

- Переместите желаемое значение хеш-функции в регистр EDX
- Переместить DWORD в регистр EAX

Сначала мы получаем все ссылки на функцию хеширования Api. Каждый XRef будет содержать адрес, по которому вызывается функция хеширования Api, что означает, что как минимум в 5 предыдущих инструкциях мы найдем упомянутый шаблон. Таким образом, мы будем извлекать предыдущую инструкцию до тех пор, пока не извлечем желаемое значение хеш-функции, которое будет помещено в EDX. Наконец, мы можем использовать это немедленно, чтобы извлечь соответствующую функцию api из значений хэша, которые мы сгенерировали ранее, и прокомментировать имя функции рядом с адресом Xref.

Python:

```
def add_comment(addr, hashv, api_table):
    """Write a comment at addr with the matching api function.Return True if a corresponding
api hash was found."""
    # remove the "h" at the end of the string
    hashv = hex(int(hashv[:-1], 16))
    keys = api_table.keys()
    if hashv in keys:
        apifunc = api_table[hashv]
        print "Found ApiFunction = %s. Adding comment." % (apifunc,)
        idc.MakeComm(addr, apifunc)
        comment_added = True
    else:
        print "Api function for hash = %s not found" % (hashv,)
        comment_added = False
    return comment_added


def main():
    """Main"""
    f = open(
        "C:\\Users\\luffy\\Desktop\\Danabot\\05-07-2020\\Utils\\danabot_hash_table.txt", "r")
    lines = f.readlines()
    f.close()
    api_table = get_api_table(lines)
    i = 0
    ii = 0
    for xref in idautils.XrefsTo(0x2f2858):
        i += 1
        currentaddr = xref.frm
        addr_minus = currentaddr - 0x10
        while currentaddr >= addr_minus:
            currentaddr = PrevHead(currentaddr)
            is_mov = GetMnem(currentaddr) == "mov"
            if is_mov:
                dst_is_edx = GetOpnd(currentaddr, 0) == "edx"
                # needs to be edx register to match pattern
                if dst_is_edx:
                    src = GetOpnd(currentaddr, 1)
                    # immediate always ends with 'h' in IDA
                    if src.endswith("h"):
                        add_comment(xref.frm, src, api_table)
                        ii += 1
    print "Total xrefs found %d" % (i,)
    print "Total api hash functions deobfuscated %d" % (ii,)


if __name__ == '__main__':
    main()
```

Заключение

Как реверс-инженеры, мы, вероятно, продолжим сталкиваться с Api Hashing по-разному. Надеюсь, я смог показать вам какой-нибудь быстрый и грязный метод или дать вам хотя бы некоторые основы того, как победить эту технику обфускации. Я также надеюсь, что в следующий раз, когда blue team товарищу придется проанализировать DanaBot, эта статья может оказаться для него полезной и сэкономит ему время на реверс инжиниринг этого банковского трояна.

- Dropper = e444e98ee06dc0e26cae8aa57a0cddab7b050db22d3002bd2b0da47d4fd5d78c
- DLL = cde01a2eeb558545c57d5c71c75e9a3b70d71ea6bbeda790a0b871fcb1b76f49

Источник: https://malwareandstuff.com/deobfuscating-danabots-api-hashing/
Автор перевода: yashechka
Переведено специально для https://xss.is