116

Долой импорт!

Я ДАВНО УБЕДИЛСЯ, ЧТО ОТ ТАБЛИЦЫ ИМПОРТА — ОДНО ТОЛЬКО ЗЛО. ИСПОЛЬЗОВАНИЕ ЕЕ ПРОГРАММОЙ МНОГОКРАТНО УПРОЩАЕТ ЖИЗНЬ ИССЛЕДОВАТЕЛЮ. ЕМУ ВСЕГО-ТО И НАДО, ЧТО ЗАГНАТЬ ЭКЗЕШНИК В ДИЗАССЕМБЛЕР И ВНИМАТЕЛЬНО ИЗУЧИТЬ ПОЛУЧЕННЫЕ ЛИСТИНГИ. АЛГОРИТМ РАБОТЫ МОЖНО БУДЕТ ОЧЕНЬ ЛЕГКО ВОССТАНОВИТЬ ПО ЯРКИМ МЕТКАМ ВЫЗОВОВ API И ПРИЯТНОМУ ГЛАЗУ АССЕМБЛЕРНОМУ КОДУ. НЕТ, ТАК ДЕЛО НЕ ПОЙДЕТ. МНЕ ЕЩЕ НЕ ХВАТАЛО, ЧТОБЫ КАЖДЫЙ, КТО СМОЖЕТ ДИЗАССЕМБЛЕР ЗАГРУЗИТЬ СМОГ ПОНЯТЬ, КАК МОИ ПРОГРАММЫ РАБОТАЮТ. В ЭТОМ МАТЕРИАЛЕ Я ПОКАЖУ, КАК МОЖНО НЕМНОГО ПОДПОРТИТЬ ЖИЗНЬ РЕВЕРСЕРУ, ИССЛЕДОВАТЕЛЮ, ВЗЛОМЩИКУ — В ОБЩЕМ, ЛЮБОМУ, КТО ЗАХОЧЕТ ПОНЯТЬ, ЧТО ДЕЛАЕТ ТВОЯ ПРОГРАММА | Николай «GorluM» Андреев (gorlum@real.xakep.ru)

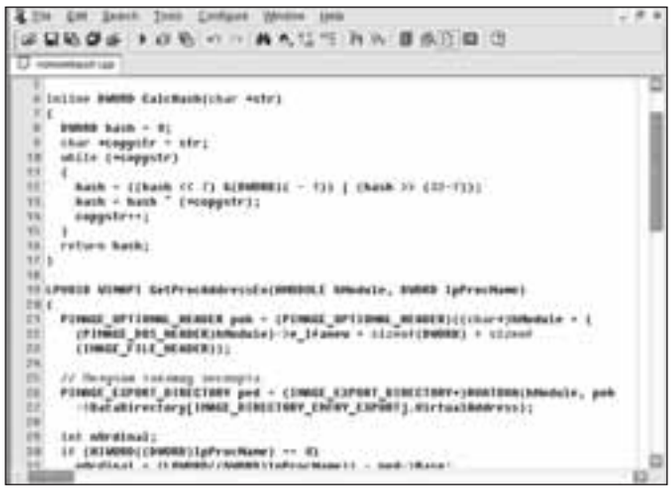
Пишем приложение, не использующее таблицу импорта, на Си

[ищем функции сами] Идея моя достаточно проста: заставить программу забыть такую страшную вещь как таблицу импорта. Причем не просто забыть, а забыть навсегда, чтобы ничто не напоминало о ней. Ни PE-заголовок, ни имена API-функций разбросанные по всему файлу. Делается это легко. Надо просто в программе вызывать все функции в обход импорта с помощью, скажем, той же GetProcAddress. Тогда компилятор не будет запихивать их в таблицу. Но сам адрес функции поиска API содержится в импорте. И даже, если все функции вначале искать с помощью GetProcAddress, таблица все равно останется. Хоть и с одной записью, но останется. Но не надо забывать о том, что одним из параметров этой функции является имя API, что, несомненно, сразу все выдаст исследователю. Поэтому имеет смысл сделать свою собственную функцию для поиска адресов функций, которые обычно прописываются в таблице импорта. Тьфу, сделать — хорошо сказал. Не надо ее делать, я ее уже давно сделал, и если ты читал предыдущие номера журнала и залезал на диск, ты должен был ее видеть. Смысл собственной функции в том, что:

1 ее не будет в таблице импорта, а следовательно, исследователю придется прилично покопаться, чтобы понять, что делает ее вызов;
2 поиск может осуществляться не только по имени, но, например, и по хэшу, посчитанному от имени (в этом случае в программе вообще не фигурирует имени API, что серьезно затрудняет процесс исследования алгоритма работы программы… ну, конечно, не для всех серьезно, но все же).

В общем, бери ее с диска. Однако, как ты помнишь, функция GetProcAddress производит поиск адрес по таблице экспорта заданного тобой модуля. Обычно дескриптор, определяющий модуль, получается с помощью функций GetModuleHandle или LoadLibrary. Но в данном случае, вот засада, эти функции вызывать в исходном коде нельзя, так как это приведет к тому, что появится таблица импорта, о которой мы стремимся забыть. Поэтому придется отыскивать дескрипторы (привычнее, наверное, назвать их хэндлами) обходными путями.

[ищем ядро и модули] Ядро, то есть дескриптор, то есть хэндл библиотеки kernel32.dll найти очень просто (она подгружается к каждому



функции подсчета хэша и поиска адресов API-функций

процессу, поэтому не надо никак дополнительно извращаться — только найти). Этим всю жизнь занимаются вирмейкеры, и поэтому метод, где только не описан. Заключается он в том, что дескриптор модуля ядра извлекается в общей сложности из структуры PEB, блока окружения процесса, ссылку на который можно получить, если обратиться к регистру fs по смещению 30h.

[всем известный поиск ядра]

```
HMODULE GetKernel()
{
  __asm {
    mov eax, dword ptr fs:[30h]
    mov eax, dword ptr [eax+0ch]
    mov esi, dword ptr [eax+1ch]
    lodsd
    mov eax, dword ptr [eax+08h]
  }
}
```

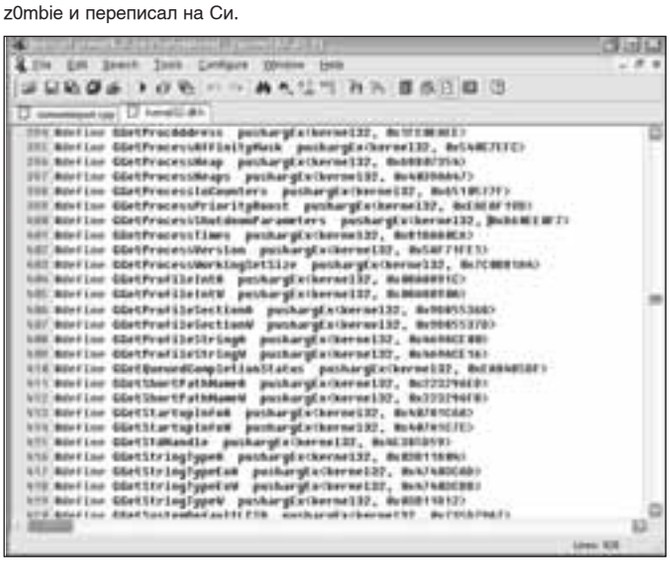
Имея ядро, можно найти в нем с помощью собственной функции GetProcAddress адрес LoadLibrary, подгрузить с ее помощью любую нужную библиотеку и искать уже в ней адрес необходимой для работы программы функции (как вариант, можно найти адрес GetModuleHandle). Геморрой, конечно, но он просто автоматизируется, а исследователю жизнь все-таки усложняет. В итоге, со всеми извращениями запуск API-функции, к примеру, MessageBox'а на Си выглядит следующим образом:

[я не такой уж и извращенец, честное слово!]

```
typedef int (WINAPI * tMessageBoxA)
  (HWND, LPCSTR, LPCSTR, UINT);
typedef HMODULE (WINAPI * tLoadLibraryA)
  (LPCSTR);

HMODULE hKernel32 = GetKernel();
tLoadLibraryA pLoadLibraryA = GetProcAddressEx
  (hKernel32, "LoadLibraryA"); // 0xC8AC8026

HMODULE hUser32 = pLoadLibraryA ("user32.dll");
tMessageBox pMessageBox = GetProcAddressEx
  (hUser32, "MessageBoxA" ); // 0xABBC680D

pMessageBox(0, "Hello world", "", 0);
```

Упс, наврал... Не со всеми извращениями. Ведь тут поиск осуществляется по имени, а нас это, как я уже говорил, не устраивает. Будем переделывать поиск, чтобы искал по хэшу.

[ищем функции по хэшу] Хэш — это некоторое число, которое считается от строки. Хэш-функция, соответственно, функция, — которая будет это число считать. Его ведь можно кучей разных способов получать. Она у нас будет очень простенькая, я ее выдрал из замечательных программ z0mbie и переписал на Си.



куча шаблонов

[хэш-функция]

```
DWORD CalcHash(char *str)
{
  DWORD hash = 0;
  char* copystr = str;
  while(*copystr) {
    hash = ((hash << 7) & (DWORD)(-1))|(hash >> (32-7));
    hash = hash^(*copystr);
    copystr++;
  }
  return hash;
}
```

Число, полученное в результате подсчета, почти уникально. Конечно, это я очень хреново выразился. Я хотел сказать, что вероятность того, что от двух разных строк ты получишь одно число — очень мала. А на пространстве имен функций какой-нибудь одной библиотеки вообще равна нулю. Потому-то и можно искать функцию в таблице экспорта, сравнивая не имена, а числа. Это позволяет хранить не палевное название API в программе, а всего лишь 4 байта хэша, которые влезают в любой регистр и могут просто жить в какой-нибудь инструкции, прямо в коде, а не в данных. Меня это обстоятельство радует безумно. Я вообще данные в программе не люблю.
Чтобы иметь возможность искать имя по хэшу, достаточно немного модифицировать функцию GetProcAddressEx. Второй параметр ее станет DWORD'ом вместо указателя на строку, а внутри вместо сравнения имен:

```
if (lstrcmp((char*)RVATOVA(hModule, *pdwNamePtr), lpProcName) == 0)
```

Будет сравнение второго параметра (hash) и подсчитанного хэша от рассматриваемого в конкретный момент имени функции:

```
if (CalcHash((char*)RVATOVA(hModule, *pdwNamePtr)) == hash)
```

Запуск MessageBox'а с такой функцией преобразиться следующим образом:

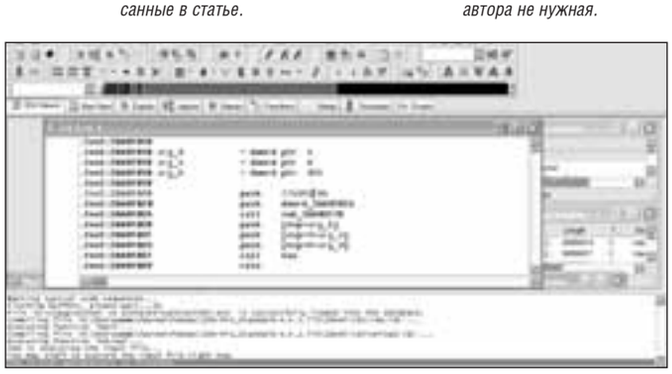
```
...
HMODULE hKernel32 = GetKernel();
tLoadLibraryA pLoadLibraryA = GetProcAddressEx
    (hKernel32, 0xC8AC8026);

HMODULE hUser32 = pLoadLibraryA ("user32.dll");
tMessageBox pMessageBox = GetProcAddressEx
    (hUser32, 0xABBC680D);
...
```

Much better, теперь имена функций нигде в программе не фигурируют. Но черт! Так невозможно программировать! Уж лучше все знают, как моя программа работает. Ведь надо считать для каждой функции хэш, подставлять по мере необходимости, заново описывать каждую API. В общем, я как обычно придумал кучу проблем, чтобы с ней разобраться.

[удобный интерфейс] Первое, от чего я хочу избавиться, так это от бесконечных определений функций. Всех этих typedef'ов кошмарных и т.п. Сделать это оказалось на удивление легко. Я просто создал несколько перегру-

Обладая огромным везением и терпением, на диске ты сумеешь откопать все исходные коды, описанные в статье.

Про это ты, наверное, нигде больше не сможешь прочесть. Это уникальная разработка никому кроме автора не нужная.



вызов функции под дизассемблером

женных шаблонов функций, которые в общей сложности могли принимать любое число параметров. И для выполнения нужной функции передавал шаблону хэш, хэндл модуля и аргументы. Все. Определять при этом каждую API не нужно, знай себе юзай шаблоны для ВСЕГО. Этакие гейты для API. Можно в них встроить какую-нибудь систему логирования, чтобы видеть все вызовы своей программы. Можно какой-нибудь время от времени срабатывающий антиотладочный прием поставить. Гейт для API в своей программе — полезная штука. В качестве препятствия анализу — особенно. Шаблоны выглядят так:

```
// для функции без аргументов, к примеру GetTickCount
template <HMODULE h, DWORD hash>
inline LPVOID pushargEx()
{
  typedef LPVOID (WINAPI *newfunc)();
  newfunc func = (newfunc)GetProcAddressEx(h, hash);
  return func();
}

// для функции с одним аргументом
template <HMODULE h, DWORD hash, class A>
inline LPVOID pushargEx(A a1)
{
  typedef LPVOID (WINAPI *newfunc)(A);
  newfunc func = (newfunc)GetProcAddressEx(h, hash);
  return func(a1);
}

// для функции с двумя аргументами
template <HMODULE h, DWORD hash, class A, class B>
inline LPVOID pushargEx(A a1,  B a2)
{
  typedef LPVOID (WINAPI *newfunc)(A, B);
  newfunc func = (newfunc)GetProcAddressEx(h, hash);
  return func(a1,a2);
}

// для функции с тремя аргументами
template <HMODULE h, DWORD hash, class A, class B, class C>
inline LPVOID pushargEx(A a1,  B a2, C a3)
{
  typedef LPVOID (WINAPI *newfunc)(A, B, C);
  newfunc func = (newfunc)GetProcAddressEx(h, hash);
  return func(a1,a2,a3);
}

// и т.д.
```

Соответственно, вызов функции через них может выглядеть следующим образом:

```
// вызовем, например, Sleep с аргументом -
// 1000, то есть одна секунда
// kernel32 - хэндл ядра, вычисленный ранее

pushargEx<kernel32, 0x3D9972F5>(1000);
```

Просто, неправда ли? Однако по-прежнему смущают хэши, которые надо вычислять для каждой функции. Для того, чтобы не приходилось заниматься еще и этим геморроем, я написал отдельную маленькую утилиту, которая вычисляет хэши и создает h-файл со специальным определением всех функций в заданной dll. Она вместе с сорцами лежит на диске. Пользоваться ей очень легко, пишешь calchash user32.dll, она тебе выдает файл user32.dll.h с содержанием, вроде такого:

```
...
#define GMessageBeep        pushargEx<user32, 0xABBEE6BC>
#define GMessageBoxA        pushargEx<user32, 0xABBC680D>
#define GMessageBoxExA      pushargEx<user32, 0x1A0256AE>
#define GMessageBoxExW      pushargEx<user32, 0x1A0256B8>
...
```

Как ты уже понял, если добавить подобный хидер в программу вместе с описанными выше функциями, то для вызова API в обход таблицы импорта будет достаточно приписать к имени функции буковку G ;). Ну и, конечно, не забыть подгрузить в самом начале все библиотеки, назвав переменные с их хэндлами по левой части имени файла dll.

[пример программы без таблицы импорта]

```
#pragma comment(linker,"/ENTRY:WinMain")

#include <windows.h>
#include "kernel32.dll.h"
#include "user32.dll.h"

// хидер с функциями GetProcAddressEx, шаблонами и т.п.
#include "noimport.h"

HMODULE kernel32;
HMODULE user32;

int WINAPI WinMain(HINSTANCE, HINSTANCE, PTSTR, int)
{
  kernel32 = GetKernel();
  user32 = GLoadLibraryA ("user32.dll");

  GMessageBox(0,"Hello world",0,0);
  return 0;
}
```

[тестируем] Ну, что в итоге мы получили, кроме того, что в PE-заголовке больше нельзя найти ссылки на таблицу импорта. Обычный определяемый любым дизассемблером вызов функции, вроде этого:

```
push  1000
call  [Sleep]
```

Изменился на:

```
push  3D9972F5h
push  [esp+var_4]
call  sub_2AA026E4
push  1000
call  eax
```

В коде теперь мало что понятно, правда? И если покруче покопаться, то с дизассемблером будет тоже самое. Но можно постепенно понять, что sub_2AA026E4 — это аналог функции GetProcAddress. Однако ищет функцию он не по имени, а по посчитанной хэш-функции от имени. Кошмар! Чтобы разобраться в том, что делает программа, написанная подобным образом, придется не один час просидеть с отладчиком и дизассемблером в руках, пытаясь сопоставить, какому из подобных вызовов какой обычный API-вызов соответствует.

Не одному только реверсеру, кстати, кошмар. Всяческие эвристики и подобная фигня на этом деле тоже вымрут — проверено. Взять, к примеру, NOD32. Я для теста написал маленькую программу, копирующую себя в системную директорию, в реестр и слушающую порт. NOD32 на параноидальном уровне безопасности тот час же окрестил ее как «вероятно вирус». Потом я заменил все вызовы на мои «хитрые». Угадай, что на это сказал антивирус? Ничего плохого! Скушал, буркнул «спасибо, вирусов не найдено, приходите еще», и на боковую.

И это ведь только начало такой замечательной штуки как precompiled-метаморфизм (вероятно, термин я придумал дурацкий, зато достиг в этой штуке немалого). Компилятор можно научить таким офигенным штукам, какие и не снились обычным полиморфным или метаморфным движкам. Полная перестройка программы для них — пустяк. Главное — правильно оформить исходный код, то есть научить всему компилятор. Можно, к примеру, заставить его оформлять разные вызовы. Скажем, в зависимости от номера строки исходного кода вставлять либо call, либо push $+5\push addr\ret, либо еще что-нибудь. Хэш-функции использовать все время разные. Можно попробовать научить компилятор (о, недокументированные возможности, как вы прекрасны и ужасны одновременно) перегружать операторы стандартных типов. Ты представь только — подсунуть свой оператор присваивания вместо дефолтного, и впихивать в него все время кучу лишнего кода и каких-нибудь нехороших трюков еще на этапе компиляции. Можно написать отличную систему, которая позволяла бы обычные нормальные программы компилировать так, словно компилятор в задницу оса укусила, так что никакие протекторы и упаковщики не понадобятся.

Тебя, наверно, мучает вопрос: а зачем все это действительно нужно? Усложнение анализа и т.п. — это понятно, а зачем метаморфизм какой-то, да еще и на этапе компиляции. Почему бы ни использовать готовые навесные протекторы, зачем устраивать весь этот геморрой?

Буду краток. Ради Дао