Внедрение DLL и перехват API-вызовов

- О среде Windows каждый процесс получает свое адресное пространство.
- Указатели, используемые Вами для ссылки на определенные участки памяти, — это адреса в адресном пространстве Вашего процесса, и в нем нельзя создать указатель, ссылающийся на память, принадлежащую другому процессу.
- Так, если в Вашей программе есть «жучок», из-за которого происходит запись по случайному адресу, он не разрушит содержимое памяти, отведенной другим процессам.

- Раздельные адресные пространства очень выгодны и разработчикам, и пользователям. Первым важно, что Windows перехватывает обращения к памяти по случайным адресам, вторым — что операционная система более устойчива и сбой одного приложения не приведет к краху другого или самой системы.
- Но, конечно, за надежность приходится платить: написать программу, способную взаимодействовать с другими программами или манипулировать другими процессами, теперь гораздо сложнее.
- Вот ситуации, в которых требуется прорыв за границы процессов и доступ к адресному пространству другого процесса:

1. создание подкласса окна, порожденного другим процессом;
2. получение информации для отладки (например, чтобы определить, какие DLL используются другим процессом);
3. установка ловушек (hooks) в других процессах.

- Порождение подкласса окна, созданного чужим процессом, возможно.
- Проблема не столько в создании подкласса, сколько в закрытости адресного пространства процесса.
- Если бы можно было как-то поместить код своей оконной процедуры в адресное пространство процесса A, это позволило бы вызвать *SetWindowLongPtr* и передать ей адрес *MySubclassProc,* в процессе A.
- Называется такой прием внедрением (injecting) DLL в адресное пространство процесса. Существует несколько способов подобного внедрения

Внедрение DLL с использованием реестра

- HKEY_LOCAL_MACHINE\Software\Microsoft\Windows_NT\CurrentVersion\Windows\AppInit_DLLs

- Значением параметра AppInit_DLLs может быть как имя одной DLL (с указанием пути доступа), так и имена нескольких DLL, разделенных пробелами или запятыми.
- Поскольку пробел используется здесь в качестве разделителя, в именах файлов не должно быть пробелов. Система считывает путь только первой DLL в списке — пути остальных DLL игнорируются, поэтому лучше размещать свои DLL в системном каталоге Windows, чтобы не указывать пути.

- При следующей перезагрузке компьютера Windows сохранит значение этого параметра. Далее, когда User32.dll будет спроецирован на адресное пространство процесса, этот модуль получит уведомление DLL_PROCESS_ ATTACH и после его обработки вызовет *LoadLibrary* для всех DLL, указанных в параметре AppInit_ DLLs.
- В момент загрузки каждая DLL инициализируется вызовом ее функции DllMain с параметром fwdReason, равным DLL_PROCESS_ATTACH. Поскольку внедряемая DLL загружается на такой ранней стадии создания процесса, будьте особенно осторожны при вызове функций. Проблем с вызовом функций Kernel32. dll не должно быть, но в случае других DLL они вполне вероятны — User32. dll не проверяет, успешно ли загружены и инициализированы эти DLL.
- Это простейший способ внедрения DLL. Все, что от вас требуется, — добавить значение в уже существующий параметр реестра.

Недостатки способа

1. Ваша DLL проецируется на адресные пространства только тех процессов, на которые спроецирован и модуль User32.dll. Его используют все GUI-приложения, но большинство программ консольного типа — нет. Поэтому такой метод не годится для внедрения DLL, например, в компилятор или компоновщик.
2. Ваша DLL проецируется на адресные пространства всех GUI-процессов. Но Вам-то почти наверняка надо внедрить DLL только в один или несколько определенных процессов. Чем больше процессов попадет "под тень" такой DLL, тем выше вероятность аварийной ситуации. Ваш код выполняется потоками этих процессов, и, если он зациклится или некорректно обратится к памяти, Вы повлияете на поведение и устойчивость соответствующих процессов.

Поэтому лучше внедрять свою DLL в как можно меньшее число процессов.

Недостатки способа

3. Ваша DLL проецируется на адресное пространство каждого GUI-процесса в течение всей его жизни.

- Тут есть некоторое сходство с предыдущей проблемой. Желательно не только внедрять DLL в минимальное число процессов, но и проецировать ее на эти процессы как можно меньшее время.
- Так что лучшее решение — внедрять DLL только на то время, в течение которого она действительно нужна конкретной программе.

Внедрение DLL с помощью ловушек

- Внедрение DLL в адресное пространство процесса возможно и с применением ловушек. Чтобы они работали так же, как и в 16-разрядной Windows, Microsoft пришлось создать механизм, позволяющий внедрять DLL в адресное пространство другого процесса

Рассмотрим его на примере

- Процесс А (вроде утилиты Spy++) устанавливает ловушку WH_GETMESSAGE и наблюдает за сообщениями, которые обрабатываются окнами в системе. Ловушка устанавливается вызовом *SetWindowsHookEx*

- HHOOK hHook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, hInstDll, 0);

Как все это действует:

- 1. Поток процесса B собирается направить сообщение какому-либо окну.
- 2. Система проверяет, не установлена ли для данного потока ловушка WH_GETMESSAGE.
- 3. Затем выясняет, спроецирована ли DLL, содержащая функцию *GetMsgProc,* на адресное пространство процесса B.
- 4. Если указанная DLL еще не спроецирована, система отображает ее на адресное пространство процесса B и увеличивает счетчик блокировок (lock count) проекции DLL в процессе B на 1.

- 5. Система проверяет, не совпадают ли значения *hinstDll* этой DLL, относящиеся к процессам A и B Если *hinstDll* в обоих процессах одинаковы, то и адрес *GetMsgProc* в этих процессах тоже одинаков Тогда система может просто вызвать *GetMsgProc* в адресном пространстве процесса A. Если же *hinstDll* различны, система определяет адрес функции *GetMsgProc* в адресном пространстве процесса B по формуле:

GetMsgProc B = histDll B + (GetMsgProc A - hinstDll A)

Вычитая *hinstDll* из *GetMsgProcA,* Вы получаете смещение (в

байтах) адреса функции *GetMsgProc.*

Добавляя это смещение к *hinstDll B,* Вы получаете адрес *GetMsgProc,* соответствующий проекции DLL в адресном пространстве процесса B

- 6. Счетчик блокировок проекции DLL в процессе B увеличивается на 1.
- 7. Вызывается *GetMsgProc* в адресном пространстве процесса B.
- 8. После возврата из *GetMsgProc* счетчик блокировок проекции DLL в адресном пространстве процесса B уменьшается на 1.

- Когда система внедряет или проецирует DLL, содержащую функцию фильтра ловушки, проецируется вся DLL, а не только эта функция. А значит, потокам, выполняемым в контексте процесса B, теперь доступны все функции такой DLL.
- Итак, чтобы создать подкласс окна, сформированного потоком другого процесса, можно сначала установить ловушку WH_GETMESSAGE для этого потока, а затем — когда будет вызвана функция *GetMsgProc* - обратиться к *SetWtndowLongPtr* и создать подкласс Разумеется, процедура подкласса должна быть в той же DLL, что и *GetMsgProc.*
- В отличие от внедрения DLL с помощью реестра этот способ позволяет в любой момент отключить DLL от адресного пространства процесса, для чего достаточно вызвать:

- BOOL UnhookWindowsHookEx(HHOOK hHook);

- Когда поток обращается к этой функции, система просматривает внутренний список процессов, в которые ей пришлось внедрить данную DLL, и уменьшает счетчик ее блокировок на 1 Как только этот счетчик обнуляется, DLL автоматически выгружается. Вспомните: система увеличивает его непосредственно перед вызовом *GetMsgProc* (см. выше п. 6). Это позволяет избежать нарушения доступа к памяти Если бы счетчик не увеличивался, то другой поток мог бы вызвать *UnhookWindowsHookEx* в тот момент, когда поток процесса В пытается выполнить код *GetMsgProc,*
- Все это означает, что нельзя создать подкласс окна и тут же убрать ловушку — она должна действовать в течение всей жизни подкласса.

Внедрение DLL с помощью удаленных потоков

- Третий способ внедрения DLL — самый гибкий. В нем используются многие особенности Windows: процессы, потоки, синхронизация потоков, управление виртуальной памятью, поддержка DLL и Unicode.
- Большинство Windows-функций позволяет процессу управлять лишь самим собой, исключая тем самым риск повреждения одного процесса другим. Однако есть и такие функции, которые дают возможность управлять чужим процессом
- Изначально многие из них были рассчитаны на применение в отладчиках и других инструментальных средствах. Но ничто не мешает использовать их и в обычном приложении.

- Внедрение DLL этим способом предполагает вызов функции *LoadLibrary* потоком целевого процесса для загрузки нужной DLL.
- Так как управление потоками чужого процесса сильно затруднено, Вы должны создать в нем свой поток.
- Windows-функция *CreateRemoteThread* делает эту задачу несложной:

```
HANDLE CreateRemoteThread( HANDLE hProcess,
 PSECURITY_ATTRIBUTES psa,
 DWORD dwStackSize,
 PTHREAD_START_ROUTINE pfnStartAddr,
 PVOID pvParam,
 DWORD fdwCreate,
PDWORD pdwThreadId);
```

- Имеет дополнительный параметр *hProcess,* идентифицирующий процесс, которому будет принадлежать новый поток. Параметр *pfnStartAddr* определяет адрес функции потока.
- Этот адрес, разумеется, относится к удаленному процессу — функция потока не может находиться в адресном пространстве Вашего процесса.

- Теперь создан поток в другом процессе. Но как заставить этот поток загрузить нашу DLL?
- Нужно, чтобы он вызвал функцию *LoadLibrary:*

HINSTANCE LoadLibrary(PCTSTR pszlibFile);

- Заглянув в заголовочный файл WinBase.h, Вы увидите, что для *LoadLibrary* там есть такие строки:

- HINSTANCE WINAPI LoadLibraryA(LPCSTR pszLibFileName);
- HINSTANCE WINAPI LoadLibraryW(LPCWSTR pszLibFileName);

- В действительности существует две функции *LoadLibrary LoadLibraryA* и *LoadLibraryW.*
- Они различаются только типом передаваемого параметра. Если имя фаЙла библиотеки хранится как ANSI-строка, вызывайте *LoadLibraryA;* если же имя файла представлено Unicode-строкой — *LoadLibraryW.*
- Самой функции *LoadLibrary* нет.
- В большинстве программ макрос *LoadLibrary* раскрывается в *LoadLibraryA.*

- По сути, требуется выполнить примерно такую строку кода
- HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0, LoadlibraryA, "C:\\MyLib.dll", 0, NULL);

- Или, если Вы предпочитаете Unicode
- HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0, LoadLibraryW, L"C :\\MyLib.dll" , 0, NULL);

- Новый поток в удаленном процессе немедленно вызывает *LoadLibraryA* (или *LoadLibraryW),* передавая ей адрес полного имени DLL.

Так что *CreateRemoteThread* надо вызвать так:

```
// получаем истинный адрес LoadLibraryA в Kernel32 dll
   PTHREAD_START_ROUTIHE pfnThreadRtn =
   (PTHREAD_START_ROUTINE)
GetProcAddress(GetModuleHandle(TEXT("Kernel32")),
   "LoadLibraryA");
HANDLE hThread = CreateRemoteThread(hProcessRemote,
   NULL, 0, pfnThreadRtn, "C.\\MyLib.dll", 0, NULL);
```

Или, если Вы предпочитаете Unicode:

```
// получаем истинный адрес LoadLibraryA в Kernel32.dll
   PTHREAD_START_ROUTINE pfnThreadRtn =
   (PTHREAD_START ROUTINE)
GetProcAddress(GetModuleHandle(TEXT("Kernel32")),
   "LoadLibraryW");
HANDLE hThread = CreateRemoteThread(hProcessRemote,
   NULL, 0, pfnThreadRtn, L"C:\\MyLib.dll", 0, NULL);
```

- Проблема связана со строкой, в которой содержится полное имя файла DLL. Строка «C:\\MyLib.dll» находится в адресном пространстве вызывающего процесса.
- Ее адрес передается только что созданному потоку, который в свою очередь передает его в *LoadLibraryA.*
- Но, когда *LoadLibraryA* будет проводить разыменование (dereferencing) этого адреса, она не найдет по нему строку с полным именем файла DLL и скорее всего вызовет нарушение доступа в потоке удаленного процесса;
- пользователь увидит сообщение о необрабатываемом исключении, и удаленный процесс будет закрыт.
- Все верно: Вы благополучно угробили чужой процесс, сохранив свой в целости и сохранности.

- Эта проблема решается размещением строки с полным именем файла DLL в адресном пространстве удаленного процесса.
- Впоследствии, вызывая *CreateRemoteThread,* мы передадим ее адрес (в удаленном процессе).
- На этот случай в Windows предусмотрена функция *VirtualAllocEx,* которая позволяет процессу выделять память в чужом адресном пространстве:

```
PVOID VirtualAllocEx( HANDLE hProcess, PVOIO
    pvAddress, SIZE_T dwSize, DWORD flAllocationType,
    DWORD flProtect);
```

- А освободить эту память можно с помощью функции *VirtualFreeEx.*

```
BOOL VirtualFreeEx( HANDLE hProcess, PVOID pvAddress,
    SIZE_T dwSize, DWORD dwFreeType);
```

- Выделив память, мы должны каким-то образом скопировать строку из локального адресного пространства в удаленное.
- Для этого в Windows есть две функции

```
BOOL ReadProcessMemory( HANDLE hProcess, PVOID
    pvAddressRemote, PVOID pvBufferLocal, DWORD dwSize, PDWORD
    pdwNumBytesRead);

BOOL WriteProcessMemory( HANDLE hProcess, PVOID
    pvAddressRemote, PVOTD pvBufferLocal, DWOHD dwSize,
    PDWORD pdwNumBytesWritten);
```

- Параметр *hProcess* идентифицирует удаленный процесс, *pvAddressRemote* и *pvBufferLocal* определяют адреса в адресных пространствах удаленного и локального процесса, а *dwSize* — число передаваемых байтов.
- По адресу, на который указывает параметр *pdwNumBytesRead* или *pdwNumBytesWritten,* возвращается число фактически считанных или записанных байтов.

Итак:

1. Выделите блок памяти в адресном пространстве удаленного процесса через *VirtualAllocEx.*
2. Вызвав *WriteProcessMemory,* скопируйте строку с полным именем файла DLL в блок памяти, выделенный в п. 1
3. Используя *GetProcAddress,* получите истинный адрес функции *LoadLibraryA* или *LoadLibraryW* внутри Kernel32.dll.
4. Вызвав *CreateRemoteThread,* создайте поток в удаленном процессе, который вызовет соответствующую функцию *LoadLibrary,* передав ей адрес блока памяти, выделенного в п. 1.

- Теперь в удаленном процессе имеется блок памяти, выделенный в п. 1, и DLL, все еще «сидящая» в его адресном пространстве.
- Для очистки после завершения удаленного потока потребуется несколько дополнительных операций.

1. Вызовом *VirtualFreeEx* освободите блок памяти, выделенный в п. 1.
2. С помощью *GetProcAddress* определите истинный адрес функции *FreeLibrary* внутри Kernel32.dll.
3. Используя *CreateRemoteThtead,* создайте в удаленном процессе поток, который вызовет *FreeLibrary c* передачей HINSTANCE внедренной DLL.

Перехват API-вызовов подменой кода

1. Найдите адрес функции, вызов которой вы хотите перехватывать (например, *ExitProcess в Kernel32.dll).*
2. Сохраните несколько первых байтов этой функции в другом участке памяти. На их место вставьте машинную команду JUMP для перехода по адресу подставной функции. Естественно, сигнатура вашей функции должна быть такой же, как и исходной, т. е. все параметры, возвращаемое значение и правила вызова должны совпадать. Теперь, когда поток вызовет перехватываемую функцию, команда JUMP перенаправит его к вашей функции. На этом этапе вы можете выполнить любой нужный код.
3. Снимите ловушку, восстановив ранее сохраненные (в п. 2) байты.

Если теперь вызвать перехватываемую функцию (таковой больше не являющуюся), она будет работать так, как работала до установки ловушки.