SEGURANÇA DO WINDOWS
Análise sobre as APIs – Parte 2

Por: Fergo

INTRODUÇÃO

No guia anterior, vimos um pouco sobre o funcionamento das APIs do Windows e as falhas de segurança na relação de um aplicativo com o kernel do sistema. Talvez ainda não tenha ficado bem claro como todo aquele esquema funciona, então nesta segunda parte vamos colocar aqueles conhecimentos em prática.

Pretendo mostrar na prática, e passo a passo, como interceptar uma chamada de API e alterar o comportamento de um aplicativo. No caso, vamos apenas habilitar um botão em um pequeno aplicativo que eu mesmo programei para este fim (da forma mais simples possível). A princípio, pode ser um pouco complicado, já que vamos trabalhar com ferramentas de "baixo nível", como debbugers e programação em assembly. Recomendo ter um conhecimento básico de alguma das duas, mas caso não tenha, não se preocupe, pois vou tentar explicar o melhor possível como trabalhar nesses programas.

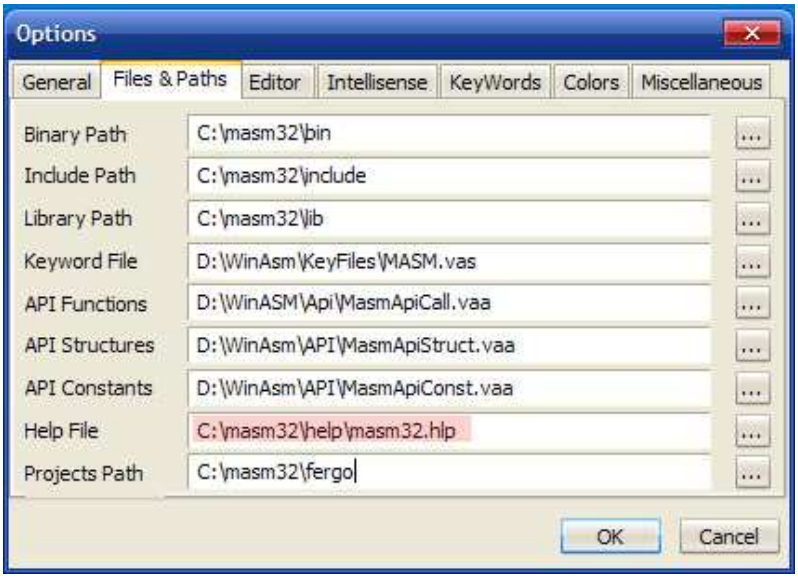
Antes de indicar quais programas vamos usar, gostaria de deixar claro que este guia e o alvo sobre o qual vamos trabalhar foram feitos apenas para fins estudantis (programados justamente para exemplificar o funcionamento das APIs), e tem o intuito de apenas alertar os usuários sobre o perigo de executar aplicativos e DLLs desconhecidos, de modo que o usuário possa identificar ou prevenir que códigos maliciosos sejam executados sem o usuário estar ciente.

FERRAMENTAS

Para iniciar nosso estudo, precisamos de algumas ferramentas básicas para entender como nosso alvo funciona e como alterar o funcionamento do mesmo.
Segue a lista dos aplicativos que vamos usar (todos são gratuitos):

- **OllyDbg** – Este é certamente o melhor debbuger/disassembler gratuito para Windows que existe. É muito completo e organizado
  www.ollydbg.de
- **Ative-me** – Nosso "alvo" sobre o qual vamos trabalhar. É apenas uma janela com um botão desativado (o qual vamos ter que ativar). Foi programado por eu mesmo utilizando o assembler MASM32
  www.fergonez.net/download.php?file=fontes_api.rar
- **MASM32** – Um dos compiladores de assembly mais famosos, criado inicialmente pela Microsoft
  www.masm32.com
- **WinAsm** – Uma IDE de programação bastante versátil, com editor de resources, coloração de sintaxe e intellisense. Faz o uso do MASM32 para gerar os binários.
  www.winasm.net
- **LordPE** – Uma ferramenta muito interessante, que mostra diversas informações técnicas sobre os arquivos .exe, assim como uma lista de todos os processos e dlls carregadas no sistema.
  www.sistemo.com/LordPE/info.htm

A instalação desses aplicativos é bastante simples. Na maioria dos casos basta descompactar o conteúdo em uma pasta qualquer. O único aplicativo que vale ressaltar algumas configurações é o *WinAsm*, de modo que aponte para os diretórios do *MASM32* (libs, includes e bin). Os diretórios podem variar dependendo do local onde você extrai, então configure o *WinAsm* (menu *"Tools->Options->Files & Paths"*) da seguinte maneira, apenas alterando os diretórios conforme necessário (o item marcado em vermelho é opcional):



UM POUCO DE ASSEMBLY E O SEU FUNCIONAMENTO

Antes de iniciar a parte prática, precisamos nos familiarizar um pouco com a linguagem que vamos usar: assembly. Muitos têm medo desse nome, mas eu diria que não é tão complicado assim (em certos casos, mais fácil que qualquer outra linguagem estruturada). Conhecendo um pouco dos comandos básicos você já é capaz de entender e analisar aplicativos simples, como o nosso alvo.

Não vou explicar cada instrução existente no assembly, mas vou deixar um link com uma lista de cada instrução e a explicação do seu funcionamento:

http://www.numaboa.com.br/informatica/oiciliS/assembler/referencias/opcodes/

Este site contém uma das melhores referências nacionais sobre a linguagem assembly (ou asm, como é abreviada), muito completa e bem explicada.
Além da lista de opcodes (como são chamadas as instruções), vou colocar mais algumas referências, que valem a pena serem lidos:

Arquitetura Intel:
http://www.numaboa.com.br/informatica/oiciliS/assembler/referencias/arquitetura.php

Funcionamento da *Stack* (pilha):
http://www.numaboa.com.br/informatica/oiciliS/assembler/textos/stack1.php

Outras referências:
http://www.numaboa.com.br/informatica/oiciliS/assembler/

Eu recomendo muito a leitura dessas referências, pois explicam alguns conceitos sobre a arquitetura dos computadores que são fundamentais para o entendimento deste guia.

Agora que temos todas as ferramentas em mãos, creio que podemos dar início ao nosso estudo. Vamos começar analisando o funcionamento do alvo através do *OllyDbg*.

ANÁLISE DO ALVO

Vamos dar uma olhada no alvo. Execute o arquivo *"ative-me.exe"*. Nada muito impressionante, apenas um pequeno texto e um botão desabilitado. O nosso objetivo vai ser habilitar esse botão sem alterar sequer um byte do alvo.

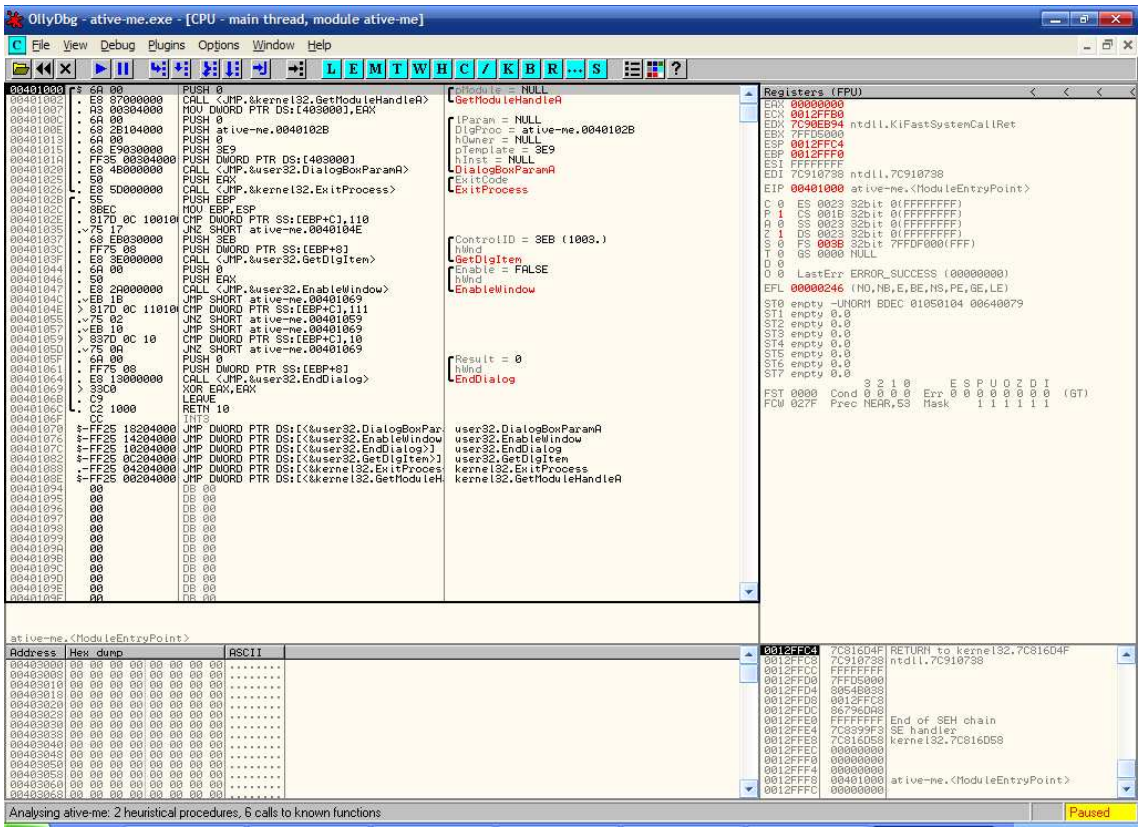
Para que algum controle (botão, caixa de texto, etc.) seja desabilitado, é usada uma função da API do Windows (*User32*) chamada *EnableWindow*.

```
BOOL EnableWindow(

    HWND hWnd,
    BOOL bEnable
);
```

Ela recebe 2 argumentos: *hWnd* e *bEnable*. O primeiro é o que chamamos de "*handle*", um valor de referência a determinado item, sobre o qual queremos ter o controle para alterar suas propriedades.
O segundo é o *bEnable*. Quando seu valor é 0 (*FALSE*), o controle indicado pelo Handle vai ter seu estado alterado para "*Desativado*". Caso contrário (*TRUE*), ele fica "*Ativado*".

Inicie o *OllyDbg*. Vá em "*File->Open*" e selecione o nosso alvo. Assim que ele carregar, você verá uma tela semelhante a essa:



Não se assuste, vou explicar o que cada pedaço dessas janelas representa.

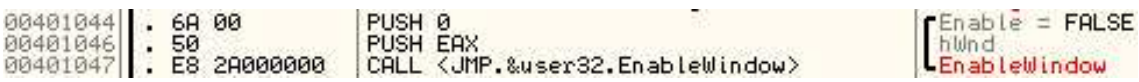
No canto superior esquerdo, encontramos o disassembly do executável. Este seria o código de máquina do nosso aplicativo. É esse código que é "enviado" para o processador para que este realize as operações.
Repare que existem 4 colunas no disassembly (caso não existe, clique com o botão direito sobre ele, vá em "*Appearence->Show Bar*"). A mais da esquerda é o *Offset*, o endereço de referência de cada instrução. Entenda como se fossem as "linhas de código" do alvo (numa generalização). A direita dela tem o *Hex Dump* de cada instrução. São esses valores em Hexadecimal que o processador usa para realizar os procedimentos. Cada operação que você realiza possui um determinado código de instrução (opcode). Em seguida vêm a coluna do *Disassembly*, que nada mais é do que os códigos em hexadecimais traduzidos para uma linguagem que o ser humano possa interpretar. Por último vem a coluna de comentários, onde o Olly analisa as instruções e informa dados úteis, como por exemplo as chamadas de APIs e os argumentos que ela recebe.

No canto superior direito está a janela de registradores (EAX, EBX, ECX, EDX...). Registradores são memórias localizadas dentro do processador, sobre as quais, boa parte das instruções realiza suas operações.

A janela inferior direita contém a *Stack* (Pilha). Não vou explicar profundamente o que ela é e como ela funciona, já que indiquei um link explicando sobre o funcionamento, mas somente para constar, ela também é uma estrutura de memória, mas se comporta de uma maneira diferente (chamada de *LIFO: Last In First Out*). Imagine uma pilha de livros, onde você necessita tirar todos os livros do topo para poder remover algum livro no meio da pilha. Os dados da pilha são colocados ou retirados por apenas 2 comandos: *PUSH* e *POP*, sendo que o primeiro é usado para colocar um dado na pilha, e o segundo para remove-lo.

Por último, o canto inferior esquerdo, correspondente ao *Memory Map*. Como o próprio nome já diz, este local informa sobre todos os dados armazenados no espaço de memória alocado para a execução do nosso aplicativo.

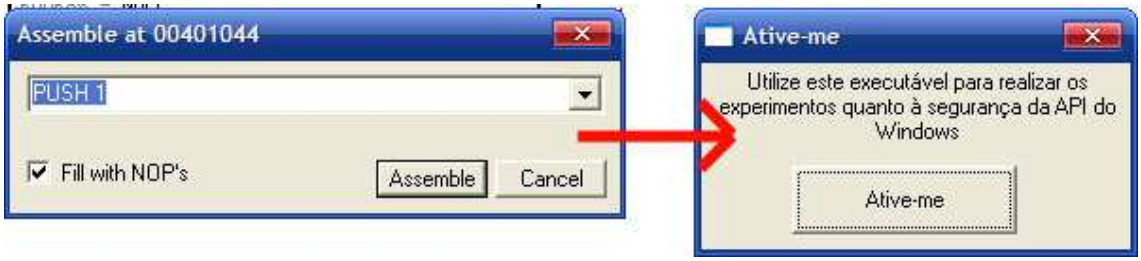
Agora que estamos um pouco mais familiarizados, podemos realmente começar o nosso estudo. Com o alvo aberto no Olly, vamos olhar as APIs que o programa utiliza. Aperte CTRL+N para abrir a janela "*Names*". Esta janela mostra uma lista de todas as APIs utilizadas pelo alvo. Um dos itens nessa lista nos chama atenção: *user32.EnableWindow*. Clique duas vezes sobre este item e o Olly nos leva ao local onde essa API é chamada pelo alvo (veja imagem abaixo).



Repare que o Olly identificou a chamada da API, assim como os dois argumentos que ela recebe (*hWnd* e *Enable*), que são colocados na pilha através do comando *PUSH*.

Repare que ele coloca o valor 0, que representa o estado do nosso controle (0 = desativado) e em seguida coloca o EAX, que logicamente contém o *Handle* do controle. Depois ele simplesmente faz a chamada para a função que vai desabilitar o botão.

Você deve estar pensando: bom, e se eu alterasse o *PUSH* 0 para *PUSH* 1? Assim eu alteraria o argumento da API, fazendo com que eu ative o botão ao invés de desativá-lo. Correto, e isso funciona. Se quiser testar, dê um duplo clique sobre o PUSH 0 e altere para *PUSH* 1 (clicando em *Assemble*). Feche a janela do "*Assemble at...*" e clique no botão "*Play*" lá em cima. Voilá! Nosso botão está ativado.



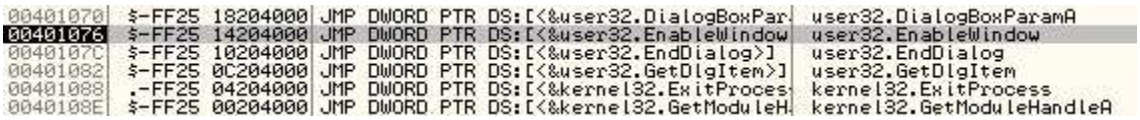
Mas o intuito deste tutorial não é esse, pois queremos fazer isso sem mexer no código do executável em si, mas sim alterar diretamente na memória.

Volte ao Olly e reinicie o alvo (através do *CTRL+F2* ou pelo "*File->Open mesmo*"). Vamos adicionar um breakpoint no local onde ele coloca o primeiro valor na pilha (*PUSH* 0).
Clique sobre a linha que contém o *PUSH* 0 (aqui o endereço dela é *0401044*) e aperte *F2*. O endereço do *PUSH* deve ter ficado vermelho. O que o breakpoint faz? Ele congela a execução do programa assim que atingir o determinado endereço, para que possamos avaliar o estado dos registradores/memória/stack.

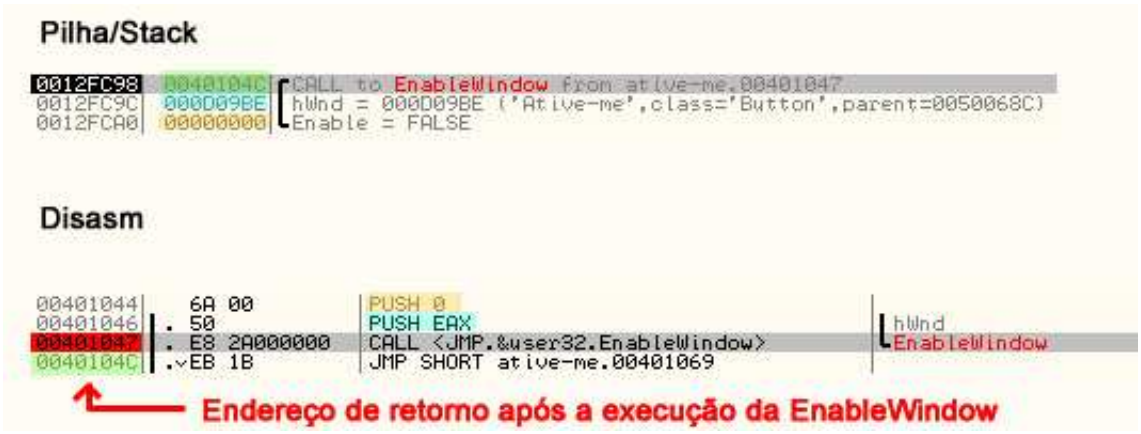
Com o breakpoint ativado, clique em "*Play*". Instantaneamente o programa congela e retorna para o Olly, indicando que atingimos o nosso breakpoint. O comando *PUSH* 0 ainda não foi executado. Vamos rodar o programa instrução por instrução agora e ver o que acontece. Aperte *F7* uma vez e preste atenção na pilha (canto inferior direito). Assim que você pressionar a tecla, o *PUSH* 0 vai ser executado e vai adicionar o valor 00000000 na pilha. Aperte *F7* novamente, para executar a instrução do *PUSH EAX*. Repare novamente que o valor de EAX foi adicionado ao topo da pilha (pode até comparar o valor na stack com o valor de EAX na janela de registradores):

A coluna da esquerda indica o endereço da pilha. A coluna do meio é o valor armazenado naquele determinado endereço. Na terceira coluna, ficam os comentários, como na janela do disasm.

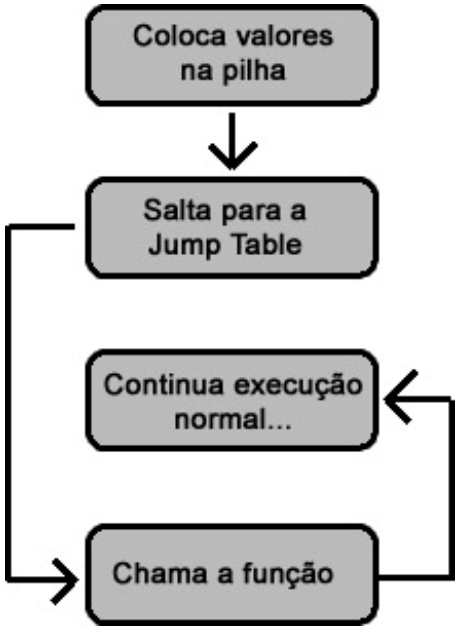
Nós estamos prestes a realizar a chamada para a API. Veja que o "cursor" (repare que o offset foi colorido de preto) indicando a posição atual do programa está sobre o *CALL <JMP.&user32.EnableWindow>.* Aperte F7 mais uma vez. Fomos levados até a chamada "*Jump Table*". Sempre que o programa chamar a função *EnableWindow*, o programa redireciona para essa *Jump Table*, que por sua vez vai em si chamar a tal função.



Repare na pilha, ela está completa agora, já possui os dois argumentos posicionados, assim como o endereço de retorno (quando a API terminar de executar seu código e for retornar ao nosso alvo):



Note também que o valor contido em *0012FC98* (pintado de verde) é exatamente o endereço que a API vai retornar quando terminar de executar a *EnableWindow*.
Podemos esquematizar a chamada da API da seguinte forma:



Certo, mas como vamos alterar aquele valor 00000000 da *stack* para 00000001, sem alterar o executável?
Nós vamos fazer o seguinte:

Assim que o alvo for iniciado, vamos carregar uma DLL junto a ele (como foi explicado na primeira parte deste tutorial). A DLL, quando for iniciada, vai buscar pela chamada da função *EnableWindow* e desviar a sua execução para uma rotina que vamos programar na DLL. Essa rotina, por sua vez, vai alterar o valor do *bEnable* diretamente na *stack* e em seguida repassar o controle para o executável novamente, de modo que ele execute a chamada para a *EnableWindow* como se nada tivesse acontecido. É o que chamamos de "*API Intercept*".

Voltando ao Olly, você deve estar na *JumpTable* para o *EnableWindow*. Aperte F7 mais uma vez e entraremos na rotina dessa API. As suas primeiras instruções são:

```
77D2C4D4  8BFF      MOV EDI,EDI
77D2C4D6  55        PUSH EBP
77D2C4D7  8BEC      MOV EBP,ESP
77D2C4D9  6A 60     PUSH 60
```

É aqui que vai ocorrer o desvio. Vamos ter que substituir essas primeiras instruções por um salto (JMP) para a nossa própria rotina (codificada na DLL). Não é possível inserir instruções no executável, pois isso comprometeria toda a integridade e alinhamento de endereços. A única forma é substituir as instruções existentes pelo desejado. Claro que também não podemos simplesmente substituir e esquecer dos dados que estava naquela posição anteriormente.

Vamos utilizar os seguintes procedimentos:

- Buscar pelo endereço da chamada da *EnableWindow* (retornará o endereço daquela instrução "*MOV EDI, EDI*", da figura mostrada na página anterior).
- Fazer "backup" de alguns bytes das primeiras instruções dessa API (o suficiente para comportar o nosso comando *JMP.*
- Substituir essas instruções na memória por um *JMP*, redirecionando o código para função dentro da nossa DLL.
- Essa função por sua vez, vai alterar o valor do *bEnable* na pilha, reescrever o "*backup*" na memória (para manter a integridade original da API) e retornar o controle para que o alvo continue sua execução.

Agora começa a parte mais complexa, que é escrever a DLL. Infelizmente não poderei explicar a fundo o que cada comando vai realizar, pois tornaria o tutorial muito longo (seria mais uma aula de asm do que um artigo sobre segurança). Vou apenas comentar cada comando, da forma mais direta possível.
Caso sinta dificuldade, recomendo ler alguns daqueles links que eu indiquei no início deste guia.

Vamos ao próximo capítulo.

PROGRAMAÇÃO DA DLL

Vamos lá. Abra o *WinAsm* (com os caminhos devidamente configurados, como indiquei lá no início). Com ele aberto, vá em "*File->New Project*". Marque a opção "*Create a new empty project*" e clique em *Next*. Selecione "*Standard DLL*" e conclua o assistente. Se for perguntado para salvar, escolha algum nome qualquer.

Cole o código abaixo no projeto recém criado (ele contém o código da DLL, está todo comentado). Cuidado ao copiar, copie em partes (primeiro o código desta página e em seguida o código da página seguinte, caso contrário ele vai acabar copiando o número da página junto).

```
.586
.model flat, stdcall

include    windows.inc
include    kernel32.inc
includelib kernel32.lib

substituir  PROTO

.data
nDll           db "user32.dll",0
nProc          db "EnableWindow",0
nRedirect      db "redir.dll",0
nFunc          db "substituir",0
nSize          dd 060000h
nIncrement     dd 0Ch

.data?
nAddress              dd ?
nLoc                  dd ?
nOldProt              dd ?
nRedirectAddress      dd ?
nRedirectLoc          dd ?
nBackup1              dd ?
nBackup2              dd ?
nTopOfStack           dd ?

.code
LibMain proc h:DWORD, r:DWORD, u:DWORD
    INVOKE LoadLibrary, ADDR nDll
    mov nAddress, eax
    INVOKE GetProcAddress,nAddress,ADDR nProc
    mov nLoc, eax
    INVOKE VirtualProtect,nAddress,nSize,PAGE_EXECUTE_READWRITE,OFFSET nOldProt
    INVOKE LoadLibrary, ADDR nRedirect
    mov nRedirectAddress, eax
    INVOKE GetProcAddress,nRedirectAddress,ADDR nFunc
    mov nRedirectLoc, eax
    MOV EAX, DWORD PTR DS:[nLoc]
    PUSH ECX
    mov ECX, DWORD PTR DS:[EAX]
    MOV nBackup1, ECX
    ADD EAX, 4
    mov ECX, DWORD PTR DS:[EAX]
    MOV nBackup2, ECX
    SUB EAX, 4
    mov BYTE PTR DS:[EAX], 0E9h
    MOV ECX, nRedirectLoc
    ADD EAX, 5
    Sub ECX, EAX
    SUB EAX, 4
    mov DWORD PTR DS:[EAX], ECX
    POP ECX
    mov eax, 1
    ret
LibMain Endp
```

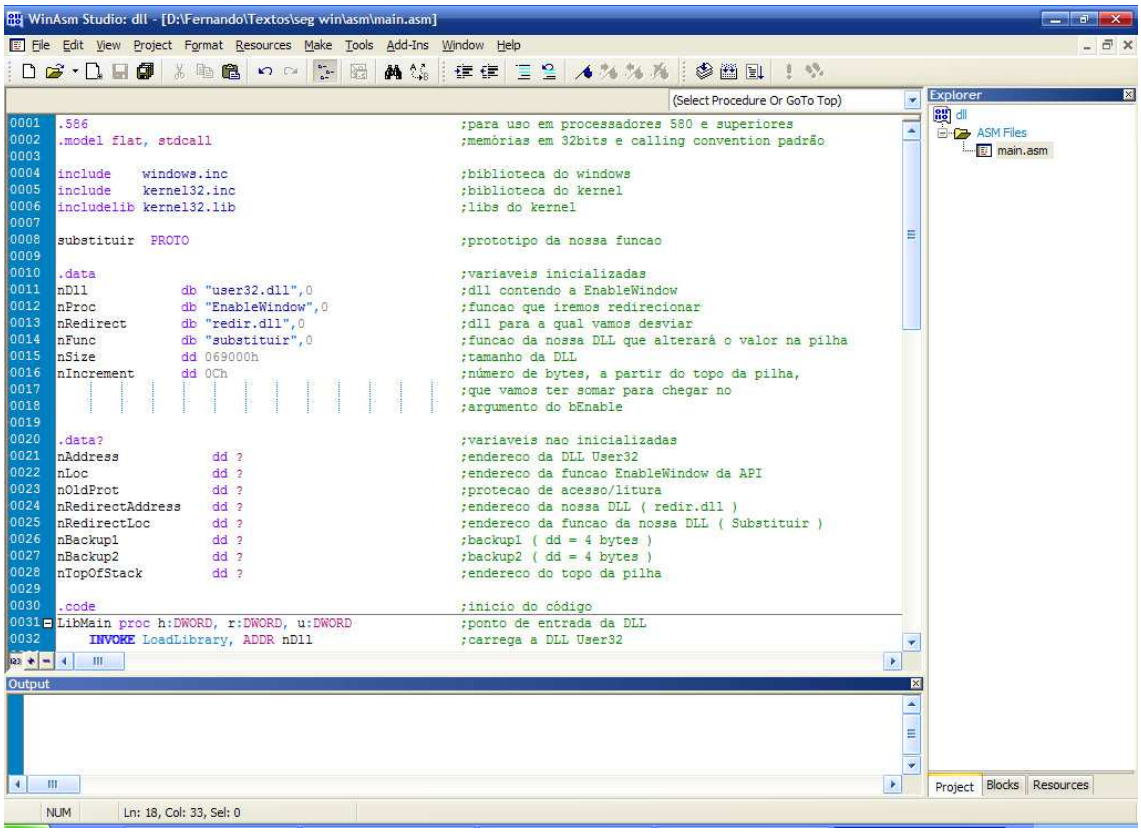
```
substituir proc
        PUSH ECX
        MOV nTopOfStack,ESP
        MOV EAX, nIncrement
        ADD EAX, nTopOfStack

        MOV DWORD PTR DS:[EAX], 01h
        MOV EAX, DWORD PTR DS:[nLoc]
        MOV ECX, nBackup1
        MOV Dword PTR DS:[EAX],ECX
        ADD EAX,4
        MOV ECX, nBackup2
        MOV Dword PTR DS:[EAX],ECX
        Sub EAX,4
        POP ECX
        JMP EAX
        ret
substituir endp

End LibMain
```

Esse código postado não está comentado, dificultando um pouco o entendimento (tive que remover os comentários devido a problemas nas quebras de linhas do PDF). Para visualizar os comentários você pode abrir o código fonte disponibilizado junto com o nosso alvo (dentro do arquivo *fontes_api.rar*). O nome do arquivo é *main.asm* e está localizado na pasta "*fontes**dll*". Lá tem cada linha comentada.

O código fica da seguinte maneira dentro do *WinAsm*:



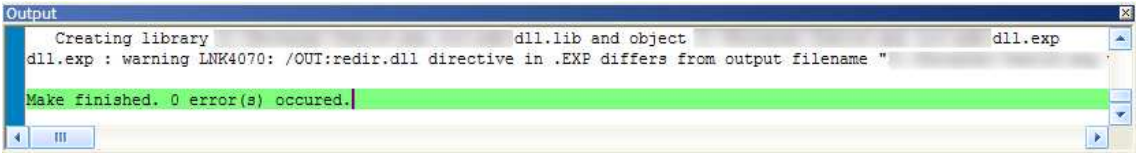
Antes de compilar a DLL, precisamos criar um arquivo de definições da mesma, contendo o nome da DLL e as funções que ela possui (no caso, somente uma).
No lado direito, em "*Explorer*" clique com o botão direito em qualquer lugar e selecione "*Add New Other*". No arquivo que ele adicionou a lista, clique com o botão direito, e vá em "*Rename*". Na janela para salvar, dê um nome qualquer e no tipo de arquivo, selecione "*Definition Files*".

Após salvar, adicione o seguinte código ao arquivo de definições que você acabou de criar:

```
LIBRARY    redir
EXPORTS    substituir
```

Salve novamente o projeto. Podemos finalmente compilar a nossa DLL.
Verifique novamente se está tudo certo, se você tem os arquivos mais ou menos desta forma (os nomes das pastas e arquivos interessam, apenas as extensões):

O arquivo *main.asm* (no meu caso) deve conter o código da nossa DLL, e o arquivo *definições.def* (no meu caso) aquelas 2 linhas que eu mencionei na página anterior.
Se estiver tudo certo, vá ao menu "*Make->Go All*". Caso os caminhos para o *MASM32* estejam configurados corretamente, sua DLL deve ter compilado com sucesso. Uma mensagem em verde, no rodapé do *WinAsm* deve indicar que não houve nenhum erro:



Se essa mensagem não apareceu, é porque algo errado aconteceu. Verifique novamente se os arquivos *.asm* e *.def* existem na pasta do seu projeto e que o conteúdo de ambos está corretamente escrito.
Verifique também nas configurações de "*Files & Paths*" do *WinAsm* se o endereço para todas aquelas pastas existem de fato.

Caso tudo tenha ocorrido normalmente, você vai ter uma .dll na pasta do seu projeto, cujo nove varia de acordo com o nome dado ao projeto. Você deve renomear essa DLL para "*redir.dll*" (sem aspas), pois foi esse o nome que indicamos no arquivo .def.

EXECUÇÃO DA DLL

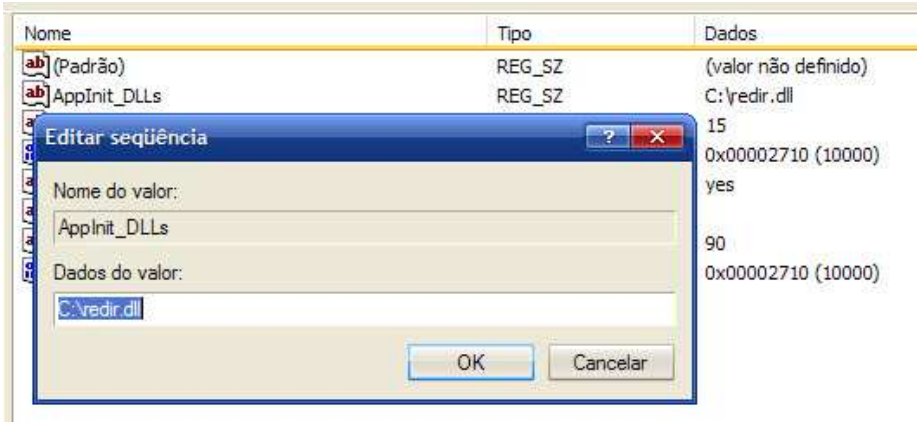
Temos a DLL, agora só nos resta testá-la. Para facilitar o processo, copie a redir.dll para a raiz de uma partição (C:\ por exemplo). Feito isso, vamos agora forçar a execução dela quando o nosso alvo for executado.

Como mencionado na primeira parte deste guia, utilizaremos uma chave do registro do Windows, que força o carregamento de uma DLL sempre que QUALQUER aplicativo for iniciado. Isso significa que, caso essa chave de registro exista e aponte para uma dll, qualquer programa iniciado (não somente o nosso alvo), vai iniciar ela junto (e os efeitos da DLL vão ser notados em todos os aplicativos).

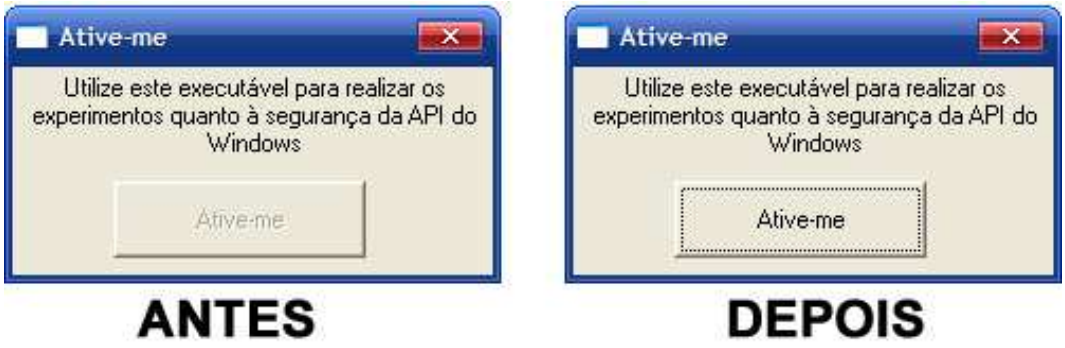
Vá até o menu "*Iniciar->Executar*" e digite "*regedit*" (sem aspas), seguido do *Enter*. O Editor de registro foi aberto. Navegue até o caminho:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
```

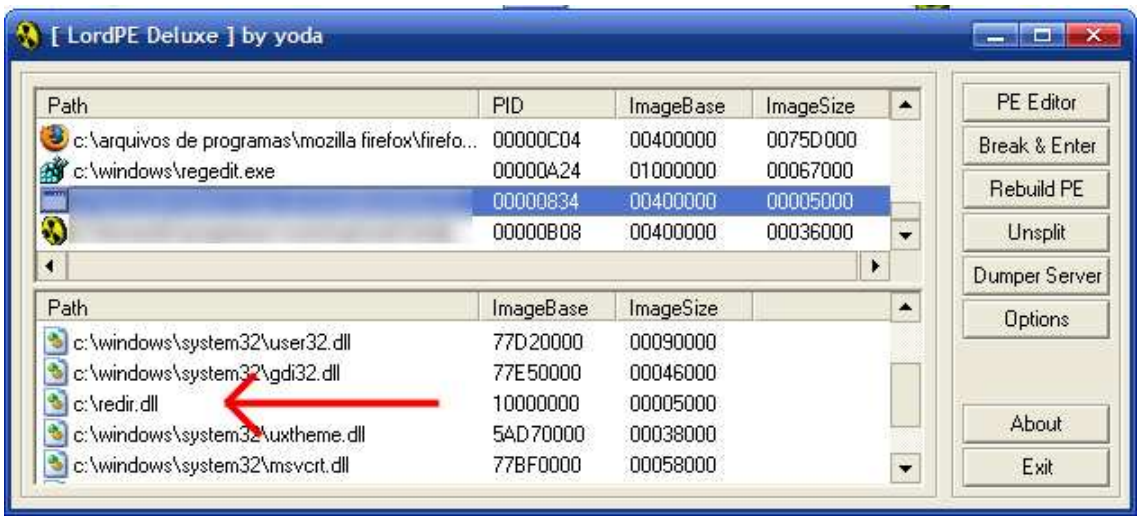
Dentro dessa chave, vão existir alguns registros. Clique com o botão direito sobre qualquer espaço em branco do lado direito da janela, vá em "*Novo->Valor de seqüência*". O nome desse valor deve ser obrigatoriamente "*AppInit_DLLs*". Depois de renomear, clique com o botão direito sobre esse registro e vá em "*Modificar*". No campo que aparecer, digite o caminho para a DLL que você compilou (*C:\redir.dll*, por exemplo).



Agora é a hora da verdade. Pode fechar o seu regedit e iniciar o nosso alvo. Se tudo foi feito corretamente, o botão do "*Ative-me*" não está mais desativado:



Se quiser ter certeza de que a DLL foi carregada, você pode utilizar o *LordPE*. Selecione o alvo na lista de aplicativos carregados, e no espaço logo abaixo você vai encontrar as DLLs carregadas pelo aplicativo selecionado. A *redir.dll* vai ser uma delas:



Para voltar ao estado normal, basta fechar o alvo e remover aquela entrada de registro.
Recomendo não deixar ela lá, pois alguns programas podem parar de responder devido ao carregamento desta DLL. Então use apenas para testar esse tutorial.

Vou aproveitar o final deste capítulo para esclarecer algumas dúvidas que podem surgir durante a análise do código.

1. Porque o nSize é 060000h?

   Não há uma razão para ter escolhido esse valor, poderia ser qualquer outro, desde que não seja menor que o tamanho da DLL gerada (como ela tem apenas 3.072 bytes, eu poderia diminuir bastante esse número)

2. Porque o valor do incremento é 0Ch?

   Pois esse é o número de bytes (12, em decimal), a partir do topo da pilha, que se encontra o valor do bEnable. Se você for traçando as instruções (através do F7) no Olly com a DLL carregada, verá que no momento que ocorrer o desvio (após o JMP), o topo da stack é no endereço 0012FC94. Somando 0Ch a esse valor, temos 0012FCA0, que é bem o endereço do bEnable (veja na imagem da stack na página 8).

3. O que é aquele monte de subtração/soma com o valor 4?

   Aquelas operações são para deslocar o ponteiro da memória (usamos o EAX, naquele caso) para outras regiões. Por exemplo: quando EAX apontava para a posição do nosso backup, eu peguei 4 bytes a partir daquela posição (*DWORD PTR DS:[EAX]* sendo que *DWORD* indica operação com 4 bytes) e coloquei no registrador ECX. Para pegar os próximos 4 bytes e colocar no backup 2, eu preciso mover o ponteiro de memória (EAX) 4 bytes a frente, já que esses quatro eu já fiz o backup.
   Para entender isso, a melhor coisa é ter o Olly aberto no alvo e ir traçando instrução por instrução e fazendo um “desenho” da memória em algum papel, com o endereço e valor de cada byte, assim como dos registradores.

CONCLUSÃO

É isso aí. Aqui se encerra este guia. Antes de finalizá-lo, vou colocar algumas dicas de como se proteger desse tipo de falha, já que é a maior razão para esse tutorial ter sido escrito.

- Não execute arquivos e instaladores que você não sabe a procedência. É uma dica um tanto quanto óbvia, mas muita gente às vezes executa algum arquivo perdido para somente "ver o que é" e sem querer acaba infectando e introduzindo algum arquivo perigoso ao sistema.
- Tenha medo de DLLs. Devido ao fato de elas não serem executáveis via duplo clique, mas sim em conjunto e controlado por outro aplicativo, é nelas que se concentram a maior parte dos códigos que danificam o sistema. Muitas vezes os softwares antivírus não detectam e nem fazem análise das DLLs, mas sim dos programas que usam elas. Infelizmente, como acabamos de ver, é possível fazer com que qualquer programa, mesmo sendo o mais inofensivo possível, execute uma DLL que pode conter algoritmos perigosos (que não foi o caso).
- Verifique periodicamente por aquela chave do registro e veja se por ventura o valor de seqüência "*AppInit_DLLs*" está apontando para um arquivo. Por padrão, nem o Windows, nem qualquer outro software necessita usar aquele valor, já que as DLLs são normalmente carregadas pelo próprio programa. Se ele existe, fique esperto, procure no *Google* pelo nome da DLL que ele está apontando e veja se encontra referências. Se não encontrar nada, recomendo excluir o "*AppInit_DLLs*", mas guardar o seu valor. Caso alguns programas parem de funcionar, você só precisa recriar a chave.
- Utilize também programas de monitoramento das DLLs carregadas na memória. Existem programas poderosíssimos, como o LordPE, que conseguem desmembrar um executável e mostrar informações preciosas sobre o comportamento e as dependências dos executáveis
- Em caso de dúvida sobre *.exe* ou .dll, use o Olly. Se não possuir experiência em assembly, use para pelo menos verificar pelos nomes das APIs utilizadas pelo arquivo (pelo atalho *CTRL+N*).

Acho que as principais dicas que eu posso apontar são essas. Não deixe de fazer valer aquelas outras tradicionais:

- Sempre manter um antivírus e firewall ativado
- Não entrar em sites dos quais não se sabe a procedência e/ou conteúdo,
- Ficar atento para os links que são divulgados nas grandes redes (como Orkut), pois em alguns casos eles mascaram links para arquivos *.bat/cmd/exe/scr*, comumente utilizados para infiltração nos computadores do usuário.

Espero que tenham gostado e aproveitado bastante. Pode ter ficado um pouco confuso, mas confesso que foi o melhor que eu pude fazer para tentar deixar o mais simples possível.

Abraços e até a próxima!

Guia escrito por: Fergo - 2007
Website: www.fergonez.net

Agradecimentos especiais:

- Gabri3l (ARTeam), pelo ótimo artigo sobre as APIs do Windows, de onde partiu a idéia de fazer algo semelhante para o nosso idioma.
- Ao site oicìliS, pela ótima referência da linguagem Assembly, assim como algumas explicações sobre a arquitetura e funcionamento de computadores
- Ao Oleh Yuschuk (OllyDbg), pela criação de um dos melhores debbugers (e ainda por cima gratuito) já feitos para a plataforma Windows.
- Aos criadores de todos os outros aplicativos utilizados por este guia. Certamente foram de muita valia.