MANIPULAÇÃO DE MEMÓRIA – TRAINERS

Por: Fernando Birck (Fergo)

SUMÁRIO

INTRODUÇÃO

Bem vindo a mais um dos meus artigos. Continuando a seqüência de tutoriais voltados a linguagem assembly, desta vez vamos falar um pouco sobre manipulação de memória dentro do ambiente Windows. Para tal, resolvi escolher um tema interessante, principalmente aos jogadores, que representa muito bem o processo de modificação de memória: os trainers.

Trainers são basicamente aplicativos que adicionam novos recursos a um determinado jogo, sem alterar nenhum arquivo em disco, mexendo apenas naquilo que foi carregado na memória. Um bom exemplo de trainers são aquelas ferramentas que adicionam veículos ao GTA. Você basicamente inicia o jogo e em seguida executa o trainer, no qual você escolhe um determinado veículo e o aplicativo faz o trabalho de gerar o automóvel no jogo. Isso é feito basicamente manipulando os dados carregados na memória pelo GTA e os adaptando e alterando para pode criar um novo veículo.

Eles são somente um exemplo. Como outros exemplos podemos citar os aplicativos anti-vírus, que alteram boa parte do espaço de memória dos aplicativos do Windows para poder ter um controle sobre o que acontece dentro deles, podendo tomar decisões e alertar o usuário antes que algo ruim aconteça ao sistema.
Programas de captura de tela também funcionam de forma semelhante. Eles se integram ao espaço de memória do Windows ou dos jogos e monitoram o teclado, podendo identificar quando determinada tecla foi pressionada e realizar a captura da imagem.

Após essa breve introdução, creio que já dá para ter certa noção da importância e da utilização desse processo, seja para reforçar a segurança do sistema ou para programar “addons” aos seus jogos.

A intenção deste guia não é ensinar a criar trainers, mas sim exemplificar uma forma de acessar a memória dos aplicativos. Tal conhecimento pode ser utilizado posteriormente para programar aplicativos semelhantes aos exemplificados dois parágrafos acima.

No nosso estudo vamos modificar um “jogo” extremamente simples e sem muito sentido, que programei apenas com a finalidade de estudar esse processo. Consiste basicamente em uma janela e alguns botões, com os quais você pode “comprar” itens enquanto possuir dinheiro para tal. O que nós vamos fazer é bolar um trainer que adicione mais dinheiro no nosso joguinho, sem modificar nada em disco ou no executável do programa.

Vale lembrar que é bom ter certo conhecimento de programação e o básico em linguagem assembly, já que vamos fazer o debug do nosso “alvo”, identificar o local onde o valor do dinheiro fica armazenado e programar um trainer para modificá-lo.

FERRAMENTAS NECESSÁRIAS

Abaixo segue uma breve lista das ferramentas que vamos utilizar:

- **OllyDbg**
  - Um dos melhores debuggers disponíveis. Vamos utilizá-lo para identificar o local de memória onde o jogo guarda a informação do dinheiro.
  - http://www.ollydbg.de/download.htm
- **WinASM**
  - IDE de programação em assembly. Gratuita e muito prática de se trabalhar. Necessita o MASM32 para compilar.
  - http://www.winasm.net/
- **MASM32**
  - Compilador de assembly mais conhecido. Também gratuito.
  - http://www.masm32.com/masmdl.htm
- **Alvo e códigos-fonte**
  - Nosso pequeno jogo e o trainer já compilado. Inclui o código-fonte de ambos os aplicativos
  - http://www.fergonez.net/files/tut_memoria.rar

OBS.: A etapa de configuração do WinASM e do MASM32 é descrita na segunda parte do meu artigo sobre a segurança do Windows, disponível no link abaixo:
http://www.fergonez.net/files/seg_windows_pt2.pdf
Recomendo ler, pois os conhecimentos descritos no guia de segurança serão utilizados neste artigo.

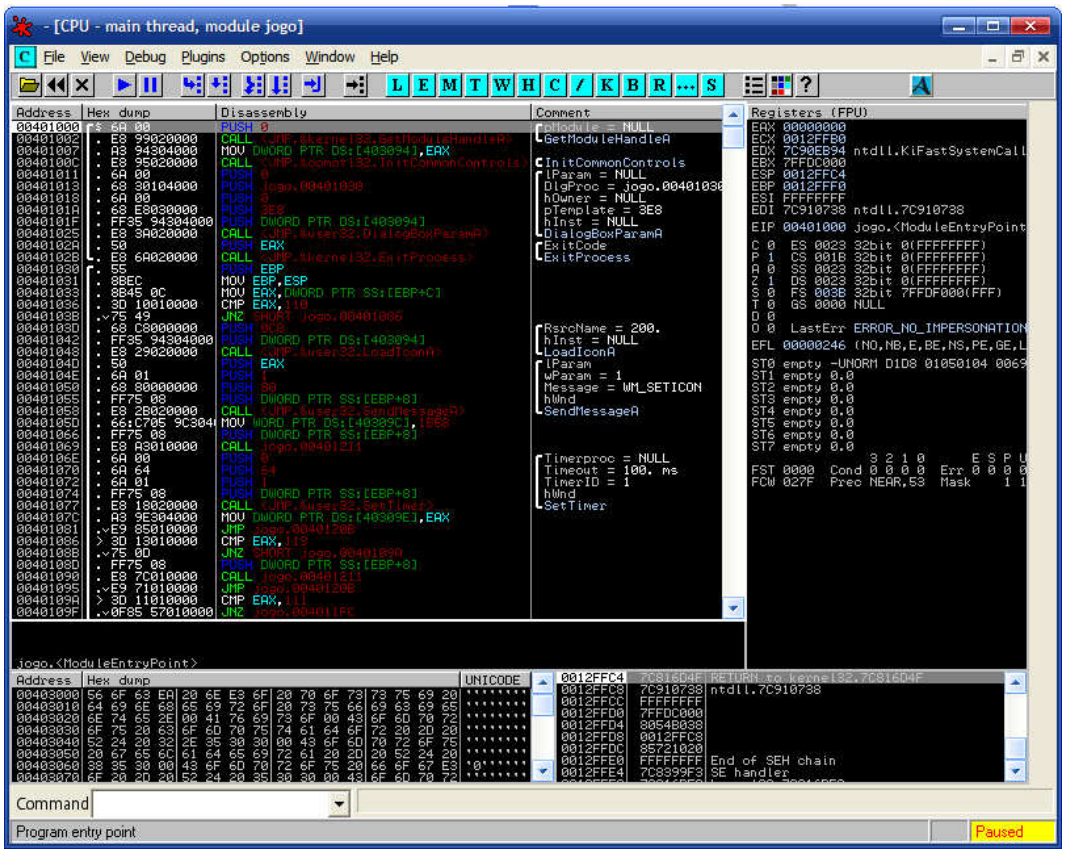
No mesmo guia consta uma breve descrição sobre o OllyDbg e o seu funcionamento. Por essa razão, não explicarei como utilizá-lo aqui.

IDENTIFICANDO O LOCAL DE MEMÓRIA

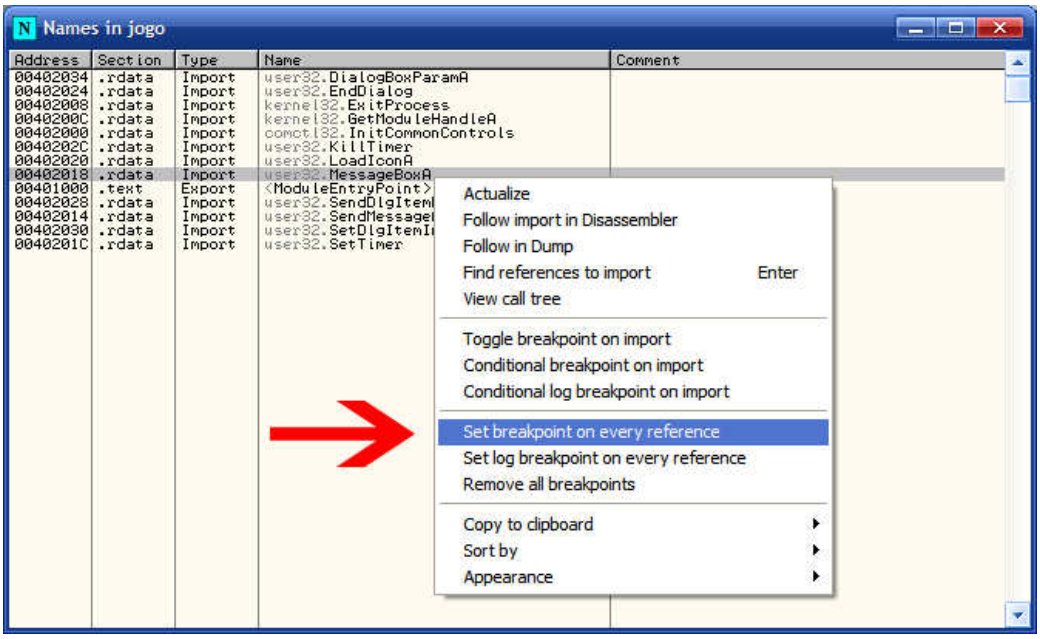
Vamos dar início ao nosso estudo. A primeira coisa que devemos fazer é analisar o básico do nosso jogo, identificando alguma de suas características que possam facilitar o trabalho dentro do OllyDbg. Apenas execute o jogo (jogo.exe, disponível dentro da pasta "Jogo de Testes"), brinque com os botões e veja um pouco do seu funcionamento. Cada vez que você pressiona algum dos botões de compra ele adiciona o item à lista de objetos adquiridos ou exibe uma mensagem caso você não tenha dinheiro suficiente. Essa mensagem vai ser de extrema importância na hora de localizar o endereço de memória onde é armazenado o dinheiro do jogo.

A mensagem de texto faz parte da API do Windows, contida dentro da user32.dll e que pode ser chamada por qualquer linguagem de programação que suporte uma "linkagem" com a API. O que nós vamos fazer é abrir nosso alvo no OllyDbg e configurar um breakpoint na chamada da função MessageBoxA (que exibe a caixa de texto). Fazendo isso, o programa irá congelar quando estiver prestes a exibir a mensagem. Porque ativar um breakpoint nessa função? Pela lógica, quando você clica em algum dos botões, o jogo teoricamente verifica pelo dinheiro disponível. Caso seja suficiente, ele adiciona o item na lista, caso contrário ele exibe a mensagem. Então se nós congelarmos o programa quando a mensagem é exibida, saberemos que estamos próximos do local onde é feita a verificação de crédito e a comparação com o preço do aparelho em questão.

Inicie o OllyDbg e abra o alvo (jogo.exe). Você verá uma tela semelhante a essa (provavelmente o tema de cores será diferente, mas isso não é problema).



Vamos então buscar pela função da API que exibe a mensagem de texto (MessageBoxA). Dentro do Olly, pressione “ALT+E”. Uma janela chamada “Executable Modules” será aberta. Nela estão listadas todas as DLLs das quais o aplicativo depende, dentre elas a “user32.dll”, que comporta a função MessageBoxA. Selecione o primeiro item da lista, cujo nome deva ser “jogo”, clique com o botão direito e em seguida clique em “View Names”. Uma nova janela aparecerá, contendo todas as funções (e não mais as DLLs) utilizadas pelo aplicativo. Repare que um dos itens é a função MessageBoxA que procurávamos. Clique novamente com o botão direito sobre essa função e selecione a opção “Set breakpoint on every reference”.



Esse item fará com que o programa insira um breakpoint em cada referência feita a função MessageBoxA, congelando o programa assim que a execução atingir esse ponto.

Feito isso, feche a janela “Names” e “Executable Modules”, retornando a tela principal do Olly. Vamos iniciar a execução do programa dentro do debugger. Para isso, basta clicar no botão “Play” e o Olly começa a execução. Para pararmos no breakpoint, precisamos fazer o programa chamar a função MessageBox, então tente comprar 2 TVs HDTV. Antes de o jogo exibir a mensagem de dinheiro insuficiente, o Olly vai pausar a execução e retornar o foco para si. A linha onde o programa foi congelado ficará marcada e a região do código será semelhante a essa:

Repare que ele foi pausado no instante em que a mensagem seria exibida. É possível notar que os quatro argumentos necessários para a exibição da mensagem de texto já foram colocados na pilha: tipo da janela, título, mensagem e a janela pai (veja o guia de segurança caso não tenha entendido esse processo). Vamos retomar agora aquela

lógica mencionada no início. A lógica era que o jogo provavelmente verifica pela quantia de dinheiro e caso ela seja insuficiente, exibe a mensagem. Pois bem, nós caímos nessa região do código justamente por que não possuímos dinheiro suficiente para comprar o item. Vamos analisar as linhas logo acima da chamada da MessageBox:

```
CMP WORD PTR DS:[40309C],1194
```

Temos aqui uma instrução de comparação (CMP). Sua sintaxe é a mostrada abaixo:

```
CMP destino, fonte
```

Essa instrução subtrai virtualmente seus operandos (fonte – destino), realizando a subtração mas não armazenando resultados, alterando apenas as flags do processador. Caso essa subtração resulte em zero ou em um valor negativo (indicando que a fonte é menor que o destino), a instrução seta a CarryFlag (CF) para 0. Outras flags são alteradas no processo, mas não vão nos interessar.

Logo abaixo dessa instrução temos um jump condicional:

```
JNB SHORT jogo.004011B7
```

JNB significa “Jump if not Below”, que traduzindo para o português seria “Salte se não abaixo”. A condição de salto dela depende da CarryFlag. Se ela for zero, o salto é realizado, caso contrário ele executa a próxima instrução, dando continuidade a execução.

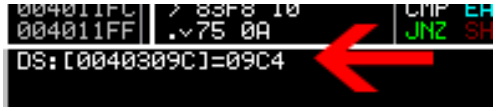
Podemos resumir essas instruções de uma maneira mais simples e didática (não da forma mais correta), através de um pseudocódigo:

```
Se fonte <= destino
      Pule para endereço
Caso contrário
      Exiba mensagem de dinheiro insuficiente
Fim Se
```

Adaptando o pseudocódigo aos valores reais mostrados pelo Olly teremos (lembre-se de que estamos sempre trabalhando com valor hexadecimal):

```
Se 0x1194 <= valor_do_endereço(0x40309C)
      Pule para 0x4011B7
Caso contrário
      Chame a MessageBox(Dono, Mensagem, Título, Estilo)
Fim Se
```

Repare que ele está comparando um número com o valor contido em um endereço de memória. O número é o 0x1194, que converteido para a base decimal fica 4500. O valor contido no endereço 0x40309C é possível obter pelo Olly selecionando a linha onde está localizada a instrução CMP. Logo abaixo da área do disassembly é mostrado o valor daquele endereço de memória:



Pela imagem podemos ver que o valor contido no endereço 0x40309C é 0x09C4, que convertendo para a base decimal nos dá 2500.
São valores extremamente interessantes, pois se você reparar, 4500 é o preço do item “HDTV” e 2500 é o dinheiro que você possui atualmente no jogo. Notando isso, nós matamos a charada, pois identificamos qual o endereço de memória utilizado pelo alvo para armazenar a quantia de dinheiro que o jogador possui atualmente, que nesse caso é 0x0040309C. Guarde bem este número, pois vamos utilizá-lo na hora de programar o trainer.

PROGRAMANDO O TRAINER

Vamos dar início ao processo de programação. Antes de entrar com códigos, vou dar uma breve explicação de como é feito esse processo, utilizando apenas os recursos e a documentação oferecida pela própria Microsoft e pela API do Windows.

Nós temos um alvo, e este possui seu espaço de memória delimitado, paginado e protegido pelo sistema, para evitar que outros aplicativos invadam ou modifiquem a sua região por descuido. No entanto, limitar completamente a memória paginada do aplicativo tornaria o sistema muito sufocado, dificultando bastante a interface entre duas ou mais aplicações distintas. Por essa razão que o Windows possui funções específicas que permitem ter o controle do espaço de memória alheio, caso contrário não teríamos aplicativos “anti-malware” ou até mesmo de monitoramento de recursos.

O processo que vamos utilizar faz o uso apenas da API do Windows, utilizando as funções programadas e embutidas no próprio sistema. A primeira atitude é obter o valor do handle (controle) da janela alvo. Tendo o controle da janela, é possível buscar o handle do processo “pai” do alvo, que posteriormente vai ser utilizado para obter o domínio sobre o processo e fazer a alteração desejada na memória.

Todo o código está disponível dentro do arquivo RAR indicado no início do tutorial. Ele já vem pronto para ser aberto e compilado no WinAsm. Abra o arquivo ‘trainer.wap’ contido dentro da pasta Trainer. Como disse no início do tutorial, é recomendável ter um conhecimento básico de assembly e da sintaxe adotada pelo MASM32. Vou ressaltar alguns trechos importantes do código aqui (ele está comentado no arquivo):

```
.data
ddEndereco        dd    0040309Ch


.data?
hInstance         dd    ?
dwDinheiro        dw    ?
dwProcessId       dd    ?
hProcess          dd    ?
```

Temos aqui apenas algumas declarações de variáveis (as mais importantes). O endereço de memória que vamos modificar fica armazenado na variável “ddEndereço”. O valor dessa variável foi aquele endereço que obtivemos no OllyDbg.

Em seguida vem as variáveis não inicializadas que serão utilizadas basicamente para os handles e IDs obtidos através de API. Uma exceção é a “dwDinheiro”, que armazenará a quantia de créditos injetada no alvo.

Em seguida vem o trecho mais importante do trainer, que é a função de callback utilizada para processar as mensagens do Windows e verificar os eventos que ocorreram. Abaixo uma imagem de como ela se assemelha (removi os comentários para poupar espaço):

```
DlgProc proc hWin:DWORD,uMsg:DWORD,wParam:DWORD,lParam:DWORD
    mov eax,uMsg
    .if eax == WM_INITDIALOG
        invoke  LoadIcon,hInstance,200
        invoke  SendMessage, hWin, WM_SETICON, 1, eax
    .elseif eax == WM_COMMAND
        mov eax,wParam
        .if eax == IDB_APLICAR
            invoke GetDlgItemInt,hWin, IDC_EDIT, NULL, FALSE
            .if eax != NULL
                .if eax > 65535
                    mov dwDinheiro, 65535
                .else
                    mov dwDinheiro, ax
                .endif
            .else
                mov dwDinheiro, 0
            .endif

            invoke FindWindow,NULL,offset szJanela

            .if eax != NULL
                invoke GetWindowThreadProcessId,eax, offset dwProcessId
                invoke OpenProcess,PROCESS_ALL_ACCESS, FALSE, dwProcessId
                mov hProcess, eax
                invoke WriteProcessMemory,hProcess, ddEndereco, addr dwDinheiro, 2, offset lpWrittenBytes
                invoke CloseHandle, hProcess
            .else
                invoke MessageBox, hWin, addr szErroMsg, addr szErroTit, MB_ICONEXCLAMATION + MB_OK
            .endif
        .elseif eax == IDB_EXIT
            invoke  SendMessage, hWin, WM_CLOSE, 0, 0
        .endif
    .elseif eax == WM_CLOSE
        invoke  EndDialog, hWin, 0
    .endif
    xor eax,eax
    ret
DlgProc endp
```

Para quem já programa sobre a API do Windows esse código não é novidade, portanto vou ressaltar a parte mais importante, que realmente executa o patch de memória, começando pela função “FindWindow”.

```
invoke FindWindow,NULL,offset szJanela
```

E a sua estrutura, retirada do MSDN:

```
HWND FindWindow(
    LPCTSTR lpClassName,
    LPCTSTR lpWindowName
);
```

Bem simples, recebe apenas dois argumentos. O primeiro é o “lpClassName”, que não é utilizado neste guia e pode ser nulo. O segundo é um ponteiro para o nome da janela (foi declarada na seção “.data” do código).

Em seguida temos a função “GetWindowThreadProcessId”:

```
invoke GetWindowThreadProcessId,eax, offset dwProcessId
```

Sua sintaxe é descrita pelo MSDN também:

```
DWORD GetWindowThreadProcessId(
    HWND hWnd,
    LPDWORD lpdwProcessId
);
```

Ela pede o handle da janela e um ponteiro para armazenar o valor do ProcessID. O handle foi obtido na função anterior (“FindWindow”) e ficou armazenado em “EAX” (registrador de retorno padrão do Windows). Estamos passando o EAX diretamente como parâmetro desta função. O “ldwProcessId” é o ponteiro para uma DWORD onde será armazenado o valor do ProcessId, após o processamento da função.

Tendo o ProcessId é possível já obter o acesso ao programa alvo. Isso é feito pela função “OpenProcess”:

```
invoke OpenProcess,PROCESS_ALL_ACCESS, FALSE, dwProcessId
```

Sua sintaxe:

```
HANDLE OpenProcess(
  DWORD fdwAccess,
  BOOL fInherit,
  DWORD IDProcess
);
```

Essa função pede três argumentos: tipo de acesso (total, restrito, para debug, etc), herança (se vai ser possível controlar os processos filhos) e o ID do processo alvo. Ela retorna um handle para o processo aberto (caso seja bem sucedida), que é movido para o “hProcess”. Por fim, esse hProcess é utilizado na operação mais importante do programa, que é a escrita na memória do alvo, executado pela função “WriteProcessMemory”:

```
invoke WriteProcessMemory,hProcess, ddEndereco, addr dwDinheiro, 2,
offset lpWrittenBytes
```

Sintaxe:

```
BOOL WriteProcessMemory(
  HANDLE hProcess,
  LPVOID lpBaseAddress,
  LPVOID lpBuffer,
  DWORD nSize,
  LPDWORD lpNumberOfBytesWritten
);
```

Ela pede cinco argumentos. Os nomes são auto-explicativos, mas vou explicar cada um deles mesmo assim. “hProcess” é o handle do processo alvo, obtido na operação anterior. “lpBaseAddress” é o endereço de memória que será modificado pela função. Ele foi declarado lá no início do código, na seção “.data”, como sendo 0040309Ch (ou 0x0040309C nos padrões do C/C++). O próximo argumento é o “lpBuffer”, um ponteiro para os valores que serão inseridos. No nosso caso ele corresponde ao dinheiro

injetado. “nSize” é a quantidade de bytes a serem escritos a partir do buffer indicado no “lpBuffer”. Como estamos inserindo uma WORD, devemos escrever dois bytes na memória. Por último tem o “lpNumberOfBytesWritten” que armazena a quantidade de bytes escritos com sucesso. Não vamos utilizar ele no nosso trainer, mas é sempre bom verificar se o número de bytes escritos corresponde ao número que bytes que desejávamos escrever. Um valor diferente indica uma falha parcial ou total na operação (é possível verificar pelo estado do EAX após a execução, pois caso ele seja nulo, a execução foi mal-sucedida).

A última função da API utilizada durante o processo é a “CloseHandle”:

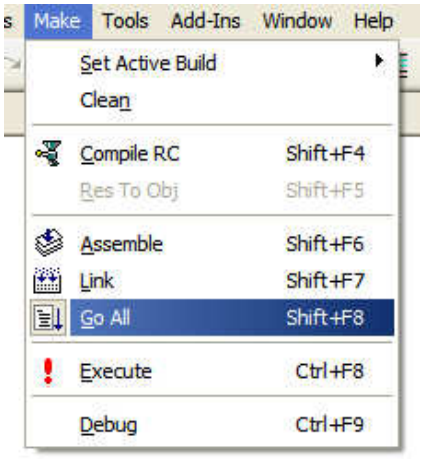
```
invoke CloseHandle, hProcess
```

Sintaxe:

```
BOOL CloseHandle(
  HANDLE hObject
);
```

Como o próprio nome já diz, ela apenas fecha o processo aberto, liberando a conexão previamente estabelecida entre o alvo e o trainer. Seu único argumento é o “hObject”, onde devemos passar o handle do processo (ou qualquer outro objeto) a ser fechado. No nosso caso é o “hProcess”.

Essas funções descritas realizam todo o processo de modificação de memória, utilizando apenas os meios legais e oficiais oferecidos pela Microsoft. O projeto do WinAsm já vem com um arquivo “.rc”, contendo os itens da janela do trainer, então basta apenas compilar o projeto para gerar o executável.



Ele vai compilar, linkar e executar o trainer após a conclusão do processo. Lembre-se de que para o trainer funcionar, é preciso ter o jogo rodando junto, então abra o alvo (jogo.exe na pasta “Jogo de Teste”) antes de aplicar o patch. Com ambos os aplicativos abertos, basta determinar a quantia de dinheiro e clicar em “Aplicar”. Automaticamente a quantia disponível no jogo será alterada para o valor escolhido.

ANTES

DEPOIS

CONCLUSÃO

Vamos ficando por aqui. Espero que este guia tenha sido útil, explicando um pouco sobre o uso das APIs do Windows para obter dados de outros aplicativos. Como disse no início do tutorial, o trainer foi apenas um exemplo devido à praticidade de se fazer um tutorial baseado neles. O mesmo processo pode ser utilizado para criação de outros softwares, principalmente àqueles relativos a segurança do sistema, proteção residente ou qualquer aplicação em que é preciso monitorar as ações antes que elas aconteçam ou sejam processadas pelo “alvo”, como no caso dos aplicativos de captura de tela.

Abraços e até a próxima!
Fergo – Agosto 2007

Website: http://www.fergonez.net