SEGURANÇA DO WINDOWS
Análise sobre as APIs

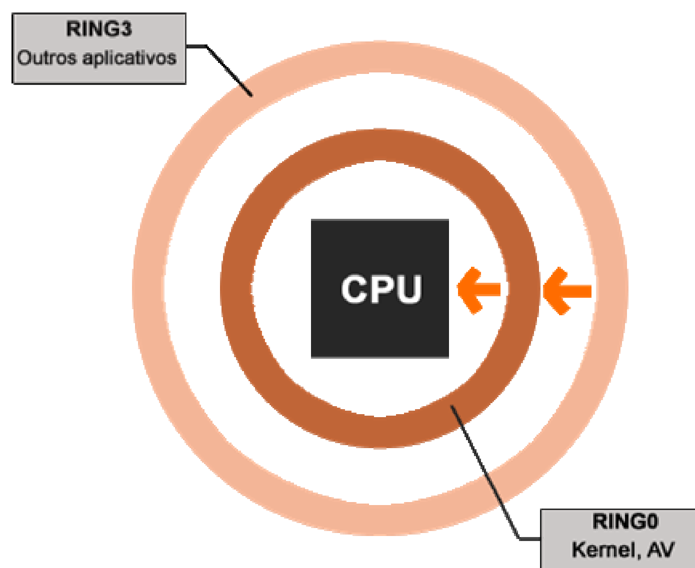
Por: Fergo

INTRODUÇÃO

Você já se perguntou alguma vez sobre o porquê do Windows suscetível à falhas e a execução de códigos mal intencionados?
Neste artigo eu vou tentar explicar melhor como funciona a relação de um aplicativo com o esse sistema operacional e porque esse método de interação permite que códigos maléficos sejam executados.

Os aplicativos não possuem acesso direto ao processador e a memória, de modo que o usuário fique restrito a alterar somente aquilo que o Windows permitir. A comunicação com esses componentes é feita através das APIs (Application Programming Interface). As APIs correspondem a uma série de funções do sistema (mais especificamente do Kernel), que tem como objetivo fazer um intermédio de comunicação dos aplicativos com o computador. Através das APIs que os executáveis se comunicam com o sistema, lêem dados do discos, exibem caixas de texto, alteram locais de memória, etc.

O nível de acesso ao hardware, é dividido nos chamados “Rings” (Ring0 e Ring3). Aplicações em Ring0 são aquelas com os maiores privilégios de acesso, podendo se comunicar diretamente com os componentes do computador. Nesta classe ficam o Kernel do Windows, assim como alguns anti-vírus e debuggers. Em Ring3 ficam todas as outras aplicações comuns, que rodam com muito menos privilégios e dependem das APIs (Ring0) para funcionar. Veja a imagem abaixo exemplificando os níveis de acesso:



Teoricamente, as APIs deveriam proteger o sistema de ataques, já que qualquer acesso ao processador e a memória deve ser feita através de funções da mesma. Infelizmente ela não é perfeita, pois quando um aplicativo incorpora uma API no código, eles estabelecem uma relação de confiança, subentendendo que não haverá interceptações nos valores enviados e recebidos pela API/programa.

FUNCIONAMENTO DA API

Como mencionado anteriormente, toda aplicação que opera em Ring3 faz o uso das APIs. Por exemplo, quando um programa precisa exibir uma mensagem de texto (como aquelas de erro), ele faz uma chamada para a função MessageBoxA pertencente a API do Windows (User32.dll), indicando o texto, o título da janela, etc. (chamados argumentos). A API, por sua vez, recebe esse argumentos (confiando que não houve alteração deles por código externo), os interpreta e envia as instruções relativas a exibição da mensagem de texto ao processador. Vejamos um exemplo de como é a função Sleep, do Windows (retirado do site da Microsoft):

```
VOID Sleep(
DWORD dwMilliseconds // sleep time in milliseconds
);
```

A função Sleep recebe como argumento o tempo que o programa deve "adormecer", em milissegundos. Veja como a chamada da função ocorre, em nível de código de execução (Assembly)

```
PUSH 1000
CALL Kernel32.Sleep
```

Nesse caso, o "PUSH 1000" (argumento da função Sleep), está sendo colocado no que chamamos de "pilha", que é apenas uma estrutura de memória (não vem ao caso explicar como uma pilha funciona). Em seguida, é feita a chamada para a função Sleep. Esta, quando iniciar a execução, vai primeiramente buscar o valor colocado na pilha (1000) e vai enviar para o processador os códigos necessários para fazer o aplicativo dormir por 1 segundo.

Como eu mencionei, a API do Kernel confia que o valor retirado da pilha foi o mesmo valor que o programa colocou. Infelizmente existem formas de interceptar essa chamada da função, desviar o código e executar qualquer outra seqüência de instruções, sendo ela danosa ou não.

Existem diversas formas de desviar o código, executar alguns procedimentos e retornar ao código original. Uma delas é alterando as instruções diretamente no executável (possível de se fazer conhecendo a linguagem Assembly), através de um assembler/debugger. Em alguns casos, o executável vem compactado e encriptado, impossibilitando a alteração direta dele. Nesse caso, o Windows tem uma grande falha, que é uma chave de registro capaz de forçar o carregamento de uma DLL para qualquer aplicativo iniciado. Vou falar um pouco sobre essa segunda falha, podendo assim entender melhor como os códigos mal intencionados são executados sem sequer tocar no executável.

CARREGAMENTO DE APLICATIVOS

Toda vez que você executa um aplicativo, o Windows pega todas as suas instruções e as coloca na memória junto com as funções do Kernel que vão ser utilizadas (dentro do executável existe uma lista de todas as APIs e funções externas necessárias).

No Windows 95/98/ME, as DLLs do sistema ficavam carregadas em um local "público" da memória, podendo ser acessada por qualquer programa. Se por ventura algum aplicativo corrompia um pedaço da memória, isso afetaria todos os outros programas, já que eles também dependiam daqueles dados. Isso explica o porquê do Win9x ser tão instável com relação aos mais novos.

A partir do Win2000, cada utilitário possuía as suas APIs carregadas em um local restrito, podendo ser utilizada única e exclusivamente pelo executável que as carregou. Se algum trecho dessas APIs for corrompido, somente o aplicativo responsável por elas será afetado. Ou seja, cada programa possui seu próprio Kernel32.dll, seu próprio User32.dll e assim por diante.

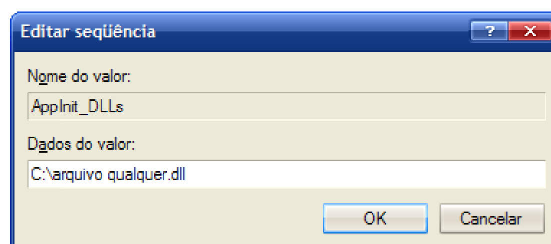
Claro que isso torna o SO muito mais estável (basta comparar o WinXP com o 9x, por exemplo). Em compensação usa-se muito mais memória, já que você vai ter várias cópias de cada DLL carregadas (uma para cada aplicativo).

Como o Windows 2000/XP protege a memória de cada aplicativo, o único modo de ler e escrever nessas regiões é executando um código que esteja contido dentro do espaço de memória alocado para o utilitário. (caso você tente acessar locais fora da região designada para o seu aplicativo, você recebe o famoso erro de "Operação Ilegal"). Mas como então que os vírus, por exemplo, conseguem alterar e monitorar dados dos aplicativos, sendo que eles não estão dentro do espaço alocado?
De várias maneiras. Uma delas é alterando o código do próprio executável, mas nem sempre você consegue editar um executável, pois ele pode estar comprimido e/ou criptografado. Aí que entra aquela chave de registro.

No registro do Windows, existe uma chave chamada *AppInit_DLLs*, que força o carregamento de qualquer DLL/EXE quando um aplicativo é iniciado.

```
HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows
```

Ela não existe por padrão, mas pode ser criada manualmente (clicando com o botão direito -> Novo -> Valor de seqüência). O valor da chave corresponde ao local da DLL a ser carregada:



Acho que agora ficou mais claro. Todo aplicativo que for iniciado, vai carregar junto com ele a DLL apontada por essa chave de registro. E o pior, vai ser carregada dentro do trecho de memória destinada ao executável, dando à DLL o controle total sobre essa região de memória.

Certo, mas como que isso é capaz de alterar o comportamento de um programa?
Bom, lá no início eu comentei que todo aplicativo e suas dependências (DLLs, etc...) são alocadas na memória quando executados, sendo que o executável tem o controle total da memória destinada a ele. Alem disso, sempre que uma DLL é carregada, o código contido na sua rotina “main” (a rotina principal), é executado.

Já que o código do executável é carregado na memória, você pode alterar as instruções do utilitário diretamente da memória.
Ironicamente, a API do Windows possui funções que retornam os endereços dos locais onde as rotinas são chamadas (dentro da memória). Imagine que você tem um programa que, ao clicar em um botão, exibe uma mensagem texto através de uma Messagebox (User32.MessageBoxA). A tal DLL poderia descobrir o endereço onde a MsgBox é chamada (através de uma função da API do Windows chamada GetProcAddress) e desviar o código da MsgBox para um outro qualquer, que pode ou não causar algum dano ao sistema. Seria basicamente este o processo:

1. Arquivo é executado
2. O aplicativo e as APIs utilizadas são carregadas na memória
3. A DLL indicada pelo registro é carregada (caso exista).
4. Todas as DLLs executam sua função principal
5. A função principal da DLL externa desvia/altera as instruções do executável na memória, pois tem acesso livre à região de memória alocada pelo Windows.
6. O executável passa a executar um código diferente do original, que foi alterado diretamente na memória pela DLL.

Sabendo dessas características dos executáveis, você agora tem pelo menos uma noção do funcionamento dos vírus. Sabendo como tudo funciona, já é um grande passo para melhorar a segurança do seu computador.
A melhor coisa é sempre desconfiar das DLLs. Elas não vem “soltas” por aí, e se vier, deve-se redobrar a atenção e ter certeza do que ela realmente faz (Google é a melhor ferramenta para descobrir informações sobre as DLLs). Também é interessante verificar se existe ou não essa chave no registro. Se existir, verifique se ela pertence a algum programa em particular. Caso não ache nenhuma informação sobre ela, remova a chave. O máximo que pode acontecer é algum aplicativo parar de funcionar (e você vai saber que foi a DLL que ocasionou isso).

Enfim, essas são as minhas recomendações quanto ao uso desse tipo de arquivo.

CONCLUSÃO

O intuito deste tutorial era tentar explicar como funciona o sistema de comunicação dos aplicativos com o Windows, assim como exemplificar algumas falhas que são aproveitadas pelos malwares para executar códigos mal intencionados sem que o Windows sequer note.
Espero ter atingido meu objetivo, mostrando mais a fundo, em nível do sistema, como os vírus e os aplicativos funcionam quando executados.

Até a próxima!

Por: Fergo
Web: www.fergonez.net

Referências:

Microsoft API Guide:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_api_start_page.asp

Weakness of Windows API, Gabriel