Testes de Robustez ao DDK do Windows XP

Manuel Mendonça[1], Nuno Ferreira Neves[2]

Universidade de Lisboa, Campo Grande, Edifício C6,
1749_016 Lisboa, Portugal

[1]manuelmendonca@msn.com
[2]nuno@di.fc.ul.pt

Resumo

Os computadores modernos têm que interagir com uma grande diversidade de dispositivos externos, que nos conduziu ao estado em que os gestores de dispositivos (do inglês device drivers - DD) são uma parte substâncial do código do sistema operativo (SO). Actualmente, a maior parte das paragens verificadas podem ser atribuídos aos DD devido a falhas na sua concretização, tornando-se alvos potênciais de ataque à medida que os ataques mais vulgares vão sendo mais difícieis de realizar. Este trabalho mostra o resultado da avaliação de um conhecido SO, o Windows XP, quanto à forma como está protegido contra entradas incorrectas simulando um ataque que usa DD maliciosos. Os resultados demonstram que na generalidade o Windows XP é razoavelmente vulnerável, e que algumas entradas provocam a paragem do sistema, sobretudo devido à falta de iniciação de variáveis e passagem de apontadores para endereços incorrectos. Em contraste, nunca foi observada a devolução de dados errados por parte do SO, o que indica uma boa capacidade de contenção de erros no Windows.

1 Introdução

O computador é uma ferramenta presente em vários domínios da nossa vida, a sua aplicação não se restringe ao trabalho, mas extende-se aos estudos, ao lazer e à saúde. A sua tecnologia tem vindo a evoluir rapidamente e é hoje o centro de ligação a numerosos dispositivos electrónicos (e.g., cameras, leitores/gravadores, impressoras, dispositivos de armazenamento e telemóveis) que são vistos como uma extensão do seu hardware. Tal só é possível porque os sistemas operativos (SO) têm permitido e acompanhado essa evolução tornando-se na medida do possível independentes do hardware. Esta flexibilidade é obtida pela virtualização oferecida pelos gestores de dispositivos (device drivers - DD) que funcionam como interface entre o hardware e o SO.

O mercado da electrónica é constantemente inundado por novos dispositivos que torna produtos de topo de gama obsoletos em meses. Para acompanhar este rítmo frenético os engenheiros são obrigados frequentemente a redesenhar tanto o hardware como o software, tornando deste modo os DD no componente mais abundante e dinâmico dum SO.

Embora hoje em dia a maior parte dos DD sejam escritos em linguagem de alto nível continuam a ser difíceis de construir, de manter e verificar. Na sua elaboração estão envolvidos conhecimentos de diversas áreas, como a arquitectura da máquina, a arquitectura do SO, compiladores, requisitos de sincronização e multiprocessamento, muitas vezes não dominados pelos programadores. Esta situação leva frequentemente à introdução de erros não intencionais quer de desenho quer de programação, tornando-os numa das mais importantes causas das falhas nos sistemas. Os problemas inerentes aos DD estão presentes tanto nos SO comercias como nos de código aberto. São responsáveis, como afirma um relatório recente, por 89% das causas de paragem do Windows XP [11] e falhas no código do Linux [5]. Por essas razões os construtores de SO estão empenhados no esforço de aumentar a robustez dos seus DD. Como exemplo, a Microsoft desenvolveu um conjunto de ferramentas de apoio ao desenvolvimento de DD [10][15], apoia os programadores na sua concepção e possui um programa de certificação [14] que visa garantir a qualidade e confiança dos mesmos. Nooks é um dos exemplos no qual a comunidade open source propõe a criação dum subsistema que tenta isolar os DD do SO para aumentar a sua

robustez [12], assim como outros projectos [4, 13, 7] que propõem soluções alternativas para a contenção desse tipo de problemas.

Tradicionalmente a origem dos erros de software devem-se a erros não intencionais de desenho ou programação. À medida que os ataques mais comuns se tornam cada vez mais difíceis de realizar, os DD poderão tornar-se um dos alvos preferenciais de ataque, principalmente se os problemas a eles imputados persistirem, tendo como única protecção a capacidade do SO em se defender. Por exemplo, um pirata informático pode disponibilizar na Internet um DD contendo faltas que lhe permitem, uma vez instalado, explorar vulnerabilidades no sistema. Para que tal aconteça é necessário apenas aguardar pela vítima.

Por esses motivos, quisemos estudar o comportamento de um SO comum, o Windows XP, quanto á forma como lida com a recepção de entradas erradas de um driver faltoso. Quisemos compreender se as funções que recebem tais entradas as processam de forma segura ou se por outro lado se encontram desprotegidas, e estando desprotegidas se levam ou não frequentementemente a um bloqueio ou paragem do SO. Estes dados são importantes porque ajudam-nos a compreender a extensão do problema através da simulação de ataques com drivers faltosos e a necessidade de encontrar soluções que quando aplicadas permitem melhorar a confiança nos actuais sistemas.

Este trabalho utiliza testes de robustez para simular a preparação de um ataque que utiliza drivers maliciosos com o objectivo de explorar a forma como o Windows XP processa entradas erradas passados pelos DD [8,3,6]. Foi escolhido um grupo de funções do Device Driver Toolkit (DDK) que foram experimentalmente avaliadas em testes que simulam um conjunto de comportamentos incorrectos, desde a falta de iniciação de funções até à passagem incorrecta de parâmetros.

Os resultados mostram que na generalidade o Windows XP é relativamente vulnerável quanto à forma como processa parâmetros incorrectos e apenas algumas rotinas fazem alguma validação de parâmetros. Muitas experiências resultaram na paragem do SO e um pequeno grupo causou a corrupção de ficheiros. Em contraste não foi observado a devolução de valores incorrectos ao DD por parte do SO, o que parece demonstrar uma boa capacidade de contenção de erros por parte do Windows.

Este trabalho está organizado da seguinte forma, na secção 2 efectuamos o enquadramento dos DD do Windows quanto à sua forma de funcionamento, classificação e classes de rotinas disponíveis para o seu desenvolvimento. Na secção 3 apresentamos a metodologia desenvolvida para criar drivers faltosos que permitem exercitar o sistema e avaliar o seu comportamento. Na secção 4 apresentamos os resultados encontrados e na secção 5 expomos as principais conclusões. Finalmente na secção 6 apresentamos o trabalho futuro.

2 Drivers Windows

Os SO da família NT, nos quais o Windows XP está incluído, foram desenhados para disponibilizar um verdadeiro ambiente de execução a 32-bit, preemptivo com memória virtual. Podem correr em múltiplas arquitecturas e plataformas de hardware e suportam sistemas de multiprocessador paralelo. Isto é conseguido colocando o núcleo do SO sobre uma camada de software que disponibiliza a abstracção do hardware (HAL – Hardware Abstraction Layer) e que ao mesmo tempo com ele se encontra comprometido. As aplicações que correm sobre o SO não chamam directamente os serviços do sistema, em vez disso, passam por uma ou mais Dynamic Link Library (DLL) de subsistemas. Estas bibliotecas exportam a interface documentada aos quais os programas estão ligados e cujos serviços podem chamar. Quando uma aplicação chama uma função de um subsistema, podem acontecer uma de três coisas: 1) A função é totalmente implementada em modo utilizador pela DLL do subsistema; 2) A função necessita de uma ou mais chamadas ao serviço executivo do SO; 3) A função necessita que seja feito algum processamento pelo processo do subsistema. Neste caso é feito um pedido ao subsistema e a DLL aguarda pela resposta antes de retornar ao chamador.

As aplicações e os DD em modo utilizador podem usar as rotinas disponibilizadas pela Win32 Application Program Interface (API) documentadas pela Platform Standard Development Kit (SDK), que por sua vez chama as rotinas exportadas pelos drivers e pelas rotinas do núcleo do SO.

A mais vulgar classificação atribuída a um DD está ligada ao facto de ser executado em modo utilizador ou em modo núcleo (kernel-mode). O Windows suporta diversos DD em modo utilizador, entre eles os gestores de dispositivos virtuais (Virtual Device Drivers - VDD), usados para emular as aplicações de 16-bit MS-DOS e o subsistema de impressoras do Windows. Os DD em modo núcleo são ainda divididos em várias categorias: os

drivers do sistema de ficheiros (File System Drivers), que aceitam os pedidos de I/O a ficheiros, os Plug and Play que trabalham com o hardware e os não Plug and Play que estendem a funcionalidade do sistema disponibilizando acesso em modo utilizador a serviços e outros gestores de dispositivos em modo núcleo. Ainda existe uma classificação mais profunda que consiste no modelo que o driver implementa seja o Modelo de Drivers do Windows (Windows Model Driver - WMD) ou o modelo em camadas (Layered Drivers).

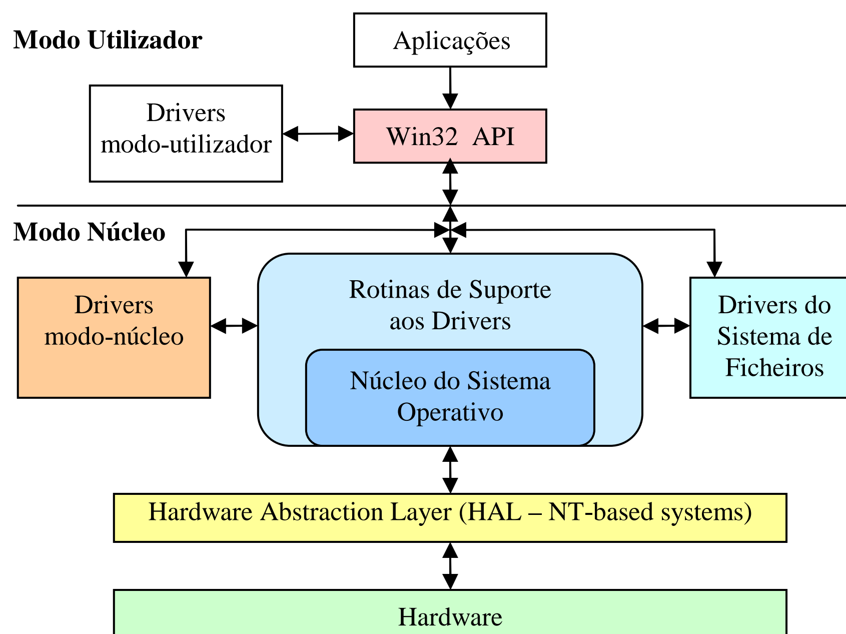

Figura 1 - Componentes do Windows.

Tal como as aplicações, os DD em modo utilizador correm no modo de processador não privilegiado e não têm acesso aos dados do sistema, excepto pela chamada das funções disponibilizadas pela Win32 API [1,2]. Contrariamente, os DD em modo núcleo correm como parte executiva do Windows e podem efectuar operações protegidas, aceder às estruturas do sistema e usar as várias rotinas exportadas pelos outros componentes do sistema (ver Figura 1). Este aumento de privilégio é acompanhado de maior dificuldade na depuração do código e no aumento do risco de corromper o sistema.

As rotinas utilizadas na construção dos DD estão documentadas no DDK e podem ser divididas em dois grandes grupos, Standard Drivers Routrines (SDR) e Driver Support Routines (DSR).

O Windows espera que todos os DD em modo núcleo implementem algumas das rotinas SDR, como a `DriverEntry` (o ponto de entrada do driver) e outras funções que compõem as funcionalidades do driver.

3 Metodologia para os Testes de Robustez

Para descobrir como é que o SO está protegido contra ataques de drivers maliciosos quisemos conhecer até que ponto as rotinas exportadas pelo DDK suportam a recepção de parâmetros com valores incorrectos ou quando as mesmas não são executadas tal como a documentação as descreve.

Na execução dos testes de robustez queremos compreender como é que uma determinada interface suporta a recepção de parâmetros errados nas funções que disponibiliza. Cada teste consiste na chamada à função a testar passando-lhe parâmetros com valores correctos e incorrectos e observar o seu comportamento e do sistema envolvente. Devido ao elevado número de funções e de parâmetros, e ao intervalo de valores possíveis para cada parâmetro, rapidamente se percebe que é necessário muito tempo para executar o número de testes que

resulta desta explosão combinatória. Este tipo de problema é particularmente relevante no Windows DDK uma vez que exporta mais de 600 funções.

Contudo nem todas as funções exportadas são utilizadas com a mesma frequência, algumas são mais usadas (estaticamente) por diferentes drivers do que outras e por isso certas funções têm mais impacto no sistema. Também é verdade que na generalidade as funções são muitas vezes chamadas utilizando um subconjunto de todos os valores definidos pelo seu tipo. Para além disso, para alguns grupos de funções, os tipos de parâmetros variam pouco o que permite automatizar algumas operações para a geração dos testes. Com base nestas observações construímos um conjunto de ferramentas que concretizam a metodologia representada na Figura 2 e que nos permite gerar o conjunto de testes com os quais procedemos à avaliação do sistema simulando o ataque de drivers faltosos.

Nesta abordagem construímos o DevInspect, uma ferramenta que realiza a análise automática ao sistema para elaborar a lista dos DD disponíveis e efectuar a contabilidade do número de vezes que cada função do DDK é encontrada na lista de importações. Usando esta medida podemos seleccionar um grupo de funções para teste, a lista candidata, ordenada por ordem de uso ou por categoria.

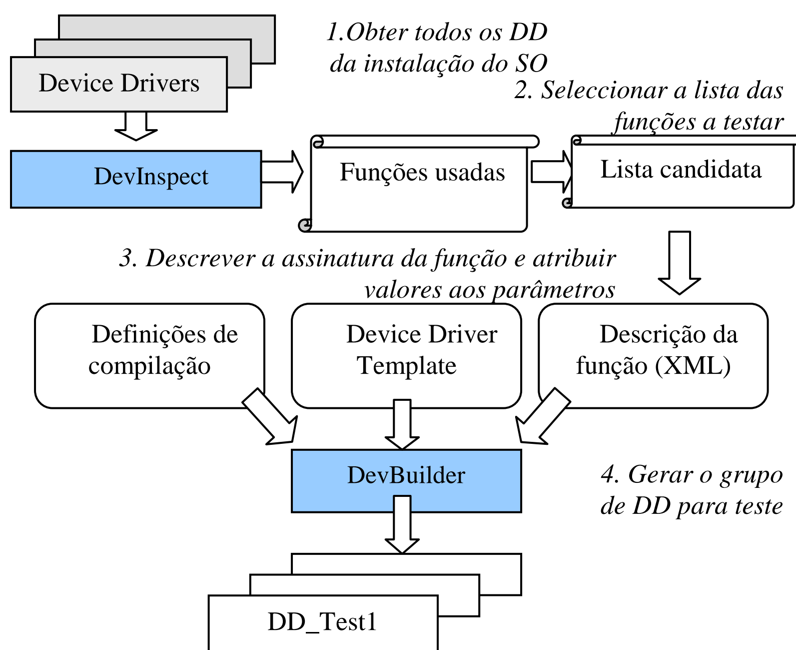

Figura 2 – Metodologia para a geração dos DD de teste.

É utilizado um ficheiro XML escrito manualmente, para descrever o protótipo de cada função e definir a carga de faltas (i.e. os valores errados que devem ser testados). De seguida outra ferramenta, o DevBuilder, toma como entrada a informação deste ficheiro XML, um ficheiro que contém o código de um DD template e alguns ficheiros de suporte à compilação e procede à geração do conjunto de testes que exercita o sistema alvo. O conjunto de testes consiste num driver distinto por cada falta que se pretende injectar numa função.

3.1 Selecção da Lista de Funções Candidatas a Teste

No SO Windows, tal como acontece para outras aplicações, os DD seguem o modelo Portable Executable File Format [9]. Neste formato, os ficheiros de imagem do executável, incluem a lista de funções exportadas e importadas.

Examinando os módulos com extensão "*.sys" localizados em \system32\drivers podemos conhecer os drivers disponíveis no sistema.

Através do exame da lista de funções importadas por cada DD podemos conhecer as funções do DDK que são utilizadas por diferentes drivers. Numa instalação típica do Windows XP, que consiste na instalação do SO

sobre um computador pessoal compatível com arquitectura Pentium, vulgarmente comercializado pelos mais variados fabricantes, foram encontrados 252 drivers que importaram 1764 funções distintas.

As 20 funções mais frequentemente importadas estão presentes em cerca de 97% dos drivers e cerca de 1000 funções são utilizadas por apenas 1 ou 2 drivers. Estes resultados sugerem que se uma das funções mais frequentemente importadas tratar de forma incorrecta os seus parâmetros, então quase todos os drivers estão potencialmente comprometidos se forem sujeitos a ataque. Por outro lado, existem muitas funções que são raramente utilizadas, por isso menos testadas pelos programadores, e que podem conter vulnerabilidades por explorar. A Tabela 1 representa as 20 funções mais frequentemente importadas que foram seleccionadas para construir o conjunto de testes. Em cada linha dessa tabela está presente um identificador da função, o nome da função, o número de DD que a importa e a classe do sistema a que pertence.

No DDK é comum encontrar funções que funcionam de forma complementar. Por exemplo, como os recursos do SO não são infinitos, normalmente quando existe uma função que obtém um recurso do SO existe uma outra função que o devolve. Esse facto é observado nas funções HAL::KfAcquireSpinLock /HAL::KfReleaseSpinLock pelo número de drivers que as usa. É interessante no entanto verificar que o mesmo não acontece com as funções ntoskrnl:: ExAllocatePoolWithTag/ntoskrnl::ExFreePoolWithTag que sendo complementares não são importadas pelo mesmo número drivers. Estes dados podem revelar a existência de uma fuga de recursos na medida em que poderá haver drivers que consomem recursos sem os devolver.

ID	Nome da função	#	Classe
1	ntoskrnl::ExAllocatePoolWithTag	215	Executive Support
2	ntoskrnl::RtlInitUnicodeString	203	Run-time library
3	ntoskrnl::IofCompleteRequest	199	I/O Manager
4	ntoskrnl::KeInitializeEvent	190	Kernel
5	ntoskrnl::KeWaitForSingleObject	185	Kernel
6	ntoskrnl::KeInitializeSpinLock	177	Kernel
7	ntoskrnl::IofCallDriver	176	I/O Manager
8	ntoskrnl::KeSetEvent	176	Kernel
9	HAL::KfAcquireSpinLock	171	Kernel
10	HAL::KfReleaseSpinLock	171	Kernel
11	ntoskrnl::ZwClose	169	ZwXxx
12	ntoskrnl::IoDeleteDevice	165	I/O Manager
13	ntoskrnl::IoCreateDevice	165	I/O Manager
14	ntoskrnl::ObfDereferenceObject	141	Object Manager
15	ntoskrnl::KeBugCheckEx	139	Kernel
16	ntoskrnl::ZwQueryValueKey	131	ZwXxx
17	ntoskrnl::ExFreePoolWithTag	126	Executive Support
18	ntoskrnl::ZwOpenKey	123	ZwXxx
19	ntoskrnl::PoStartNextPowerIrp	128	Power Management Support
20	ntoskrnl::IoFreeMdl	104	I/O Manager

Tabela 1 – As 20 funções mais importadas pelos DD numa instalação do Windows.

Pode-se argumentar, com razoabilidade, que as 20 funções mais frequentemente importadas poderão não corresponder às 20 funções mais utilizadas no sistema. Embora possa corresponder à realidade, as funções mais utilizadas dependerão do que o sistema tiver e em determinadas circunstâncias poderão estar restringidas a um conjunto limitado de drivers. A escolha efectuada, que assenta na análise estática, não pretende suplantar a importância da actividade dinâmica do sistema ou os resultados obtidos com essa escolha, mas antes ser o mais abrangente possível, agregando o maior número e tipo de drivers do sistema.

3.2 Driver Template

A principal responsabilidade do DevBuilder é construir os DD que injectam as faltas, baseados no código do driver template (ver Figura 2). Para executar essa tarefa foi construído um DD em modo núcleo que implementa o conjunto mínimo de funcionalidades necessárias para poder ser instalado no SO e desencadear a falta.

Este driver implementa as funções de entrada, saída e processamento do driver que são adaptadas pelo DevBuilder ao interpretar os dados contidos no ficheiro XML. No código do driver foram colocadas marcas que permitem ao DevBuilder conhecer o local onde deve colocar o código que traduz do ficheiro XML. Em cada driver, o DevBuilder realiza o novo processamento do serviço IOCTL para chamar a função em teste e atribuir um conjunto de valores aos parâmetros da função. Para desencadear a falta uma aplicação tem que pedir ao driver a execução do serviço apropriado utilizando o mecanismo IOCTL disponibilizado pelo SO. Nesse pedido é passado um bloco de dados para que o driver o possa preencher com informação que permita determinar o sucesso ou insucesso da chamada à função em teste e conhecer os valores devolvidos pela função ou pelos seus parâmetros.

O ficheiro XML contém os dados relevantes para construir um conjunto de drivers que possam chamar uma função passando-lhe um conjunto de parâmetros pré-definidos. Esse ficheiro é constituído pela assinatura da função, o nome, o tipo de retorno, o tipo de cada parâmetro e os valores que estes podem assumir.

Como certas funções necessitam que algumas estruturas de dados sejam iniciadas antes de se proceder à chamada da função, muitas vezes obrigando à chamada de outras funções, o ficheiro XML possibilita a inserção de código que é executado antes da chamada à função.

Desta forma dá-se a possibilidade de executar a função garantindo que foram cumpridos todos os requisitos da sua utilização. Por outro lado a omissão desse código possibilita a criação dum tipo de falta específico. De forma análoga, também existe a possibilidade de inserir código que deve ser executado após a chamada à função e que permite, por exemplo, avaliar o valor retornado pela função ou pelos seus parâmetros.

3.3 Faltas Injectadas

Para os testes que se pretenderam realizar foram definidos seis tipos de valores correctos e incorrectos (ou faltas) e que podem ser resumidos da seguinte forma:

- **Valor aceitável (VA):** o parâmetro é iniciado com um valor correcto;
- **Variável não iniciada (VNI):** o parâmetro é iniciado com um valor aleatório;
- **Apontador inválido (AI):** esta falta é normalmente utilizada em funções que esperam que o parâmetro tenha um recurso previamente atribuído pelo SO. Incluem o valor NULL ou áreas de memória incorrectas;
- **Valores proibidos (VP):** o parâmetro usa um valor que é explicitamente identificado pela documentação como incorrecto;
- **Valor fora de intervalo (VFI):** o parâmetro usa um valor fora do intervalo de valores esperado;
- **Função relacionada não executada (FNE):** esta falta é produzida quando não se chama uma função que a documentação do DDK explicitamente informa que deve ser executada antes da chamada à função que se pretende testar. Normalmente está associada à falta AI.

Os valores atribuídos aos parâmetros utilizados no teste a cada função estão distribuídos conforme representado na Tabela 2. A primeira coluna refere-se ao identificador da função (ver Tabela 1). As seis colunas que se seguem referem-se ao tipo de falta presente em determinado parâmetro. Em cada célula estão identificados os parâmetros da função (sendo P0 o primeiro parâmetro, P1 o segundo, etc.) e o número de faltas atribuídos aos parâmetros. Como exemplo, a função com ID=1, recebe três parâmetros (P0, P1 e P2). Para o parâmetro P0 foram definidos 6 valores do tipo VA, 1 valor do tipo VP e 1 valor do tipo VFI. Para o parâmetro P1 foram definidos 10 valores do tipo VA. Por último para o parâmetro P2 foram definidos 4 valores do tipo VA.

Uma vez definidos os valores a atribuir a cada parâmetro, é possível determinar o número de drivers gerados para cada função calculando a combinação dos diversos parâmetros. Tomando por exemplo a função com ID=1, o número de drivers gerado será: P0 (6+1+4) x P1 (10) x P2 (4) = 440.

ID	VA	VNI	AI	VP	VFI	FNE
1	P0=6 P1=10 P2=4	-	-	P0=1	P0=4	-
2	P0=2 P1=1	P 0 = 1 P1=1	P 0 = 1 P1=1	-	-	-
3	P1=16	P 0 = 1 P1=1	P 0 = 2	-	-	-
4	P0=1 P1=2 P2=2	P 0 = 1 P1=1	P 0 = 1	-	-	-
5	P1=2 P2=2 P3=2	P 0 = 1 P1=1 P2=1	P 0 = 1 P4=1	-	-	-
6	P0=1	P 0 = 1	P 0 = 1	-	-	-
7	-	P 0 = 1 P1=1	P 0 = 2 P1=2	-	-	-
8	P0=2 P1=2 P2=2	P 0 = 1 P1=1	P 0 = 1	-	-	-
9	-	P 0 = 1 P1=1	P 0 = 3 P1=1	-	-	(P0=8)
10	P1=11	P 0 = 1 P1=1	P 0 = 3	-	-	(P0=36)
11	-	P 0 = 1	P 0 = 2	-	-	(P0=3)
12	-	P 0 = 1	P 0 = 3	-	-	(P0=3)
13	P0=1 P1=2 P2=1 P3=1 P4=10 P5=1	P6=1	P 0 = 1	-	P4=2 P5=1	-
14	-	P 0 = 1	P 0 = 2	-	-	-
15	P0=2 P1=2 P2=1 P3=1	P 0 = 1 P1=1 P4=1	P1=1	-	-	-
16	P1=1 P2=3 P4=6	P 0 = 1 P1=1 P2=1 P3=1 P4=1 P5=1	P 0 = 2 P1=1	-	P2=1	-
17	P0=1 P1=3	P 0 = 1 P1=1	P 0 = 2	-		(P0=15)
18	P1=26	P 0 = 1 P2=1	P 0 = 2 P2=1	-	-	-
19	-	P 0 = 1	P 0 = 3	-	-	-
20	P0=1	P 0 = 1	P 0 = 2	-	-	(P0=3)

Tabela 2 - Distribuição das faltas pelas funções.

3.4 Resultados Esperados

Inicialmente a lista de resultados esperados nas experiências executadas era algo mais vasta do que a representada na Tabela 3. Essa lista foi elaborada tendo em conta várias fontes, como por exemplo trabalhos disponíveis na literatura e a opinião de diversos especialistas na administração de sistemas Windows. Contudo, á medida

que as experiências foram sendo realizadas decidiu-se reduzir essa lista uma vez que esses resultados na prática não eram observados.

Genericamente existem dois tipos de resultados, ou a falha teve como resultado uma paragem do sistema ou foi de alguma forma suportada mantendo-o em funcionamento.

ID	Descrição
F1	Não foram detectados problemas na execução do sistema.
F2	A aplicação ou todo o sistema sofreu um bloqueio.
F3	O sistema sofre uma paragem e depois inicia; o sistema de ficheiros é verificado e não são encontrados ficheiros corrompidos.
F4	O mesmo que F3 mas são encontrados ficheiros corrompidos.

Tabela 3 - Resultados esperados na execução da falta.

Uma vez que os mecanismos de suporte às faltas também podem ter problemas de implementação, o resultado do tipo F1 foi subdividido em três categorias. Para determinar qual dessas categorias se aplica a uma dada experiência, o DD verifica a correcção do valor de retorno (caso seja diferente de `void`) e dos parâmetros de saída e devolve essa informação à aplicação que desencadeou a experiência. Foram considerados três resultados quanto ao sucesso de execução da função em teste:

- **ERRO (R1):** o valor de retorno da função indica que foi detectado um erro na chamada à função, possivelmente devido a parâmetros incorrectos. Isto significa que os parâmetros inválidos foram suportados correctamente;
- **OK (R2):** o valor de retorno da função indica que a chamada à função foi executada com sucesso. Estão abrangidos por esta categoria três casos: 1) a função executou correctamente mesmo com parâmetros incorrectos; 2) não executou e retornou sucesso na mesma; 3) todos os parâmetros estavam correctos. Esta última situação ocorre quando a combinação de parâmetros válidos se combinam na chamada à função;
- **INVALIDO (R3):** o valor da função indica um valor não esperado na documentação do DDK. Isto significa que ou a documentação ou a implementação da função têm um problema.

Sempre que ocorre uma paragem, o Windows XP tem a capacidade de gerar um ficheiro *minidump* que descreve o contexto de execução do sistema no momento em que ocorreu a falha. A análise deste ficheiro é de extrema importância porque permite a quem desenvolve os sistemas determinar o motivo da paragem. Embora continuem a ser desenvolvidos esforços no sentido de determinar correctamente os motivos da paragem do sistema, ainda persistem alguns problemas que impedem a sua correcta determinação. Sempre que estas ocorrem, os ficheiros que contém os *minidumps* são analisados para verificar a capacidade de detecção da origem do problema. Neste sentido foram considerados três tipos de resultados quanto à capacidade do SO em deteminar a origem da paragem:

- **Identificação OK (M1):** o SO determinou correctamente a origem da paragem identificando o driver faltoso que a provocou;
- **Erro na identificação (M2):** o SO determinou que foi outro módulo que originou a paragem, não identificando o driver faltoso como origem;
- **Não identificado (M3):** o SO foi incapaz de determinar a origem da paragem.

Quando o SO sofre uma paragem ou o computador fica sem energia, algumas das operações de I/O que estejam em execução são permaturamente interrompidas podendo causar inconsistências nos ficheiros. Sempre que o Windows XP arranca, verifica a consistência do sistema de ficheiros e caso seja necessário executa o aplicativo `chkdsk` para proceder à sua recuperação. Quando são encontrados ficheiros corrompidos, esta aplicação move-os para uma directoria cuja nomenclatura é FOUND.XXX (onde XXX é um número sequêncial de 000 a 999). Em todas as experiências que envolvem a paragem ou o bloqueio do SO são contabilizados os ficheiros corrompidos encontrados pelo `chkdsk`.

3.5 Execução das Experiências

Uma vez que as experiências em várias situações podem provocar o bloqueio ou a paragem sistema, foram utilizadas duas máquinas para automatizar a maior parte das tarefas relacionadas (ver Figura 3). O sistema alvo tem instalado o SO e o conjunto de DD de testes.

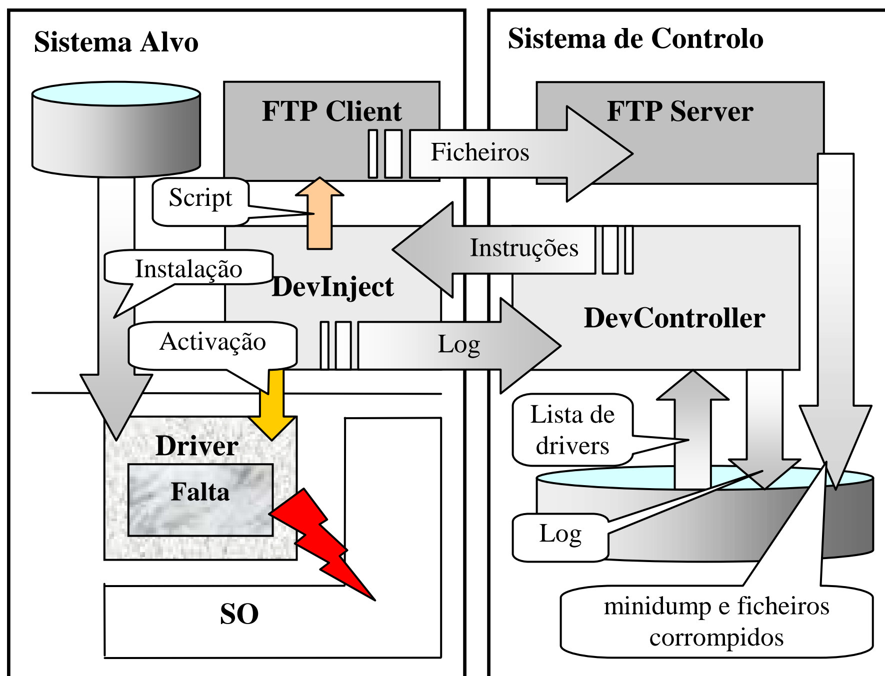

Figura 3 - Processo de injecção de faltas.

O sistema de controlo tem a seu cargo a selecção dos testes que devem ser executados, coleccionar os dados e ordenar o reboot do sistema alvo sempre que necessário.

Após o arranque do sistema alvo, o `DevInject` contacta o `DevController` para o informar de que está pronto para realizar uma experiência, em resposta o `DevController` informa-o de qual o próximo driver a utilizar. O `DevInject` instala o driver, desencadeia a falta, verifica o resultado da experiência e remove o driver do SO. Em todas as fases o `DevController` é informado do estado da experiência para que possa indicar o próximo passo ao `DevInject`. O sistema de ficheiros do sistema alvo não é utilizado para guardar os resultados da experiência, uma vez que no caso de ocorrer uma paragem estes podem ficar corrompidos. Contudo, é utilizado para manter os *minidump* e os ficheiros corrompidos que forem encontrados no início de cada arranque. No final de cada conjunto de testes, o `DevController` solicita ao `DevInject` que transfira por FTP os ficheiros *minidump* e os ficheiros corrompidos mantendo a associação entre os ficheiros, o log da experiência e os drivers utilizados.

4 Resultados Experimentais

Todas as medições foram obtidas num protótipo do sistema composto por dois PCs instalados com o Windows XP Professional Service Pack2, ligados numa rede Ethernet a 100Mbits/s.

Os resultados inscritos na Tabela 4 foram obtidos num cenário em que a actividade do sistema de ficheiros do sistema alvo provocada pelo `DevInject` era inexistente, por isso sem carga no sistema de ficheiros.

As primeiras duas colunas contêm o identificador da função em teste (ver também Tabela 1) e o número de drivers utilizados no teste à função. As colunas F1 a F4 contêm os resultados observados nas experiências. Nas

20 funções testadas não foi observado nenhum driver que tenha causado o bloqueio do sistema (coluna F2=0), no entanto, a maior parte deles deu origem a paragem (F3 e F4).

ID	Drivers	Execução da Falta				Valor de retorno			Minidump			Ficheiros Corruptos
		F1	F2	F3	F4	R1	R2	R3	M1	M2	M3	
1	440	416	0	18	6	20	396	0	24	0	0	10
2	12	9	0	2	1	0	9	0	0	3	0	1
3	51	0	0	44	7	-	-	-	51	0	0	7
4	18	6	0	10	2	-	-	-	12	0	0	3
5	36	18	0	13	5	0	18	0	18	0	0	5
6	3	2	0	0	1	-	-	-	1	0	0	1
7	9	0	0	4	5	0	0	0	9	0	0	5
8	24	12	0	11	1	0	12	0	12	0	0	2
9	8	0	0	8	0	0	0	0	4	4	0	0
10	48	3	0	42	3	-	-	-	22	22	1	3
11	3	3	0	0	0	3	0	0	0	0	0	0
12	4	0	0	3	1	-	-	-	4	0	0	1
13	96	48	0	35	13	0	48	0	48	0	0	15
14	3	2	0	1	0	-	-	-	1	0	0	0
15	12	0	0	12	0	-	-	-	12	0	0	0
16	315	315	0	0	0	315	0	0	0	0	0	0
17	16	1	0	13	2	-	-	-	15	0	0	2
18	155	104	0	47	4	104	0	0	51	0	0	5
19	4	0	0	3	1	-	-	-	4	0	0	1
20	4	1	0	3	0	-	-	-	3	0	0	0

Tabela 4 - Resultados dos testes de robustez ao Windows XP sem carga.

Apenas 2 funções se mostraram 100% imunes às faltas injectadas (funções 11 e 16).

Estes resultados revelam que a grande maioria das funções não está bem protegida contra parâmetros inválidos e que o SO mostra-se desprotegido face às entradas de DD faltosos.

As três colunas a seguir (R1 a R3) mostram a análise efectuada aos valores da coluna F1 que resultam das experiências onde não ocorreram problemas de execução. Algumas células não possuem valores, foram preenchidas com o símbolo '-', correspondendo a funções que não retornam qualquer valor. Nos casos em que existem valores retornados, em nenhum caso foi detectado um valor incorrecto, quer por parte da função quer por parte de algum parâmetro de saída (a coluna R3 apresenta sempre o valor 0). Isto significa que o Windows XP parece não propagar os erros de volta para os drivers.

A análise dos ficheiros de *minidump* produzidos em cada paragem permitiu verificar que na maior parte dos casos o SO identifica correctamente o driver faltoso (coluna M1). No entanto em três casos (funções 2, 9 e 10) foram culpabilizados outros módulos do SO (coluna M2) e houve inclusive um caso (função 10) em que o SO não conseguiu determinar a origem da paragem (coluna M3).

4.1 Motivos de Paragem

A análise ao log de execução permite estabelecer a relação entre as paragens verificadas e os drivers que as provocaram e assim desta forma conhecer o tipo de falta (ou faltas) responsáveis pelo acontecimento.

Ao isolarmos os tipos de faltas VNI, AI, VFI e FNE para comparar com o número de paragens que ocorreram no sistema (ver Tabela 5) é possível concluir que as mesmas se devem na sua maioria à presença de variáveis não iniciadas (VNI) ou a apontadores inválidos (AI).

No caso das funções 6, 12, 13, 14, 19 e 20 fica claro que o motivo da paragem se deve ao parâmetro P0 transportar faltas do tipo VNI e AI.

Na execução dos testes à função 18 foram registadas 50 paragens devido ao parâmetro P2 transportar a falta do tipo AI e 1 paragem devido a uma falta do tipo VNI.

No caso das funções 2, 3, 4, 7, 8, 9, 10 e 17 não é possível determinar com exactidão a atribuição da responsabilidade da paragem, se devido ao parâmetro P0 ou ao parâmetro P1, uma vez que ambos transportam simultâneamente valores susceptíveis de a causar.

ID	Total Paragens	P0				P1			P2		P4	
		VNI	AI	VFI	FNE	VNI	AI	VFI	VNI	AI	VNI	AI
1	24			12				24				
2	3		3			1	1					
3	51	17	34			3						
4	12	6	6			4						
5	18		18			8			6			18
6	1		1									
7	9	3	6			3	6					
8	12	6	6			4						
9	8	2	6		8	4	4					
10	45	11	34		33	3						
11	0											
12	4	1	3		3							
13	48		48									
14	1		1									
15	12	4				3	3				12	
16	0											
17	15	4	8		15	4						
18	51								1	50		
19	4	1	3									
20	3	1	2									

Tabela 5 - Principais contribuições para a paragem do sistema.

Também não é possível determinar com segurança qual o parâmetro que origina a paragem no caso da função 5 uma vez que os parâmetros P0, P1, P2 e P4 transportam valores que a podem causar, mas mais uma vez estamos restringidos aos tipos AI e VNI.

Muito provavelmente a origem das paragens verificadas na função 15 devem-se ao facto do parâmetro P4 tomar valores do tipo VNI, mas não é de excluir que os parâmetros P0 e P1 também tenham influência.

A função 1 é a excepção a esta análise pois nenhum dos parâmetros transporta os tipos de falhas que se tem vindo a considerar. Na realidade as paragens são motivadas pelo facto dos parâmetros P0 e P1 transportarem valores do tipo VFI.

As funções 11 e 16 não sofreram paragens.

4.2 Sensibilidade do Sistema de Ficheiros

Posteriormente realizou-se o mesmo conjunto de experiências fazendo variar a carga de actividade do sistema de ficheiros do sistema alvo, neste caso o `DevInject` renomeava e fazia cópia de alguns ficheiros. Os valores obtidos e registados na Tabela 6 permitiram constatar a reprodutibilidade dos resultados com a excepção do número de paragens com e sem ficheiros corrompidos (F3 e F4) e consequentemente do número de ficheiros corrompidos.

A fórmula F4/(F3+F4)x100 permite medir a sensibilidade relativa do sistema de ficheiros onde um valor de 100% significa que uma paragem provocada pelo driver provoca sempre a corrupção dos ficheiros e um valor de 0% significa que o sistema de ficheiros é imune a essa paragem.

Essa fórmula foi aplicada aos valores da Tabela 4 e da Tabela 6 e permite-nos verificar a sensibilidade relativa do sistema de ficheiros face à actividade nele existente.

ID	Drivers	Execução da Falta				Ficheiros corruptos
		F1	F2	F3	F4	
1	440	416	0	13	11	14
2	12	9	0	1	2	3
3	51	0	0	29	22	22
4	18	6	0	7	5	6
5	36	18	0	25	11	11
6	3	2	0	0	1	1
7	9	0	0	7	2	3
8	24	12	0	10	2	6
9	8	0	0	6	2	3
10	48	3	0	35	10	15
11	3	3	0	0	0	0
12	4	0	0	3	1	1
13	96	48	0	23	25	28
14	3	2	0	0	1	1
15	12	0	0	11	1	2
16	315	315	0	0	0	0
17	16	1	0	9	6	6
18	155	104	0	34	17	20
19	4	0	0	3	1	5
20	4	1	0	3	0	4

Tabela 6 - Resultados dos testes de robustez ao Windows XP com carga.

O gráfico resultante, ilustrado pela Figura 4, permite-nos verificar que na generalidade o sistema de ficheiros é bastante sensível às paragens provocadas pelos drivers, e que aumentam com a sua actividade, pois havendo um maior número de ficheiros em operações de I/O maior é o risco de algum deles ficar corrompido. No gráfico o valor (-1) distingue os casos onde nunca ocorreram paragens (funções 11 e 16) dos casos onde ocorreram mas não foram encontrados ficheiros corrompidos (funções 9, 14, 15, 20).

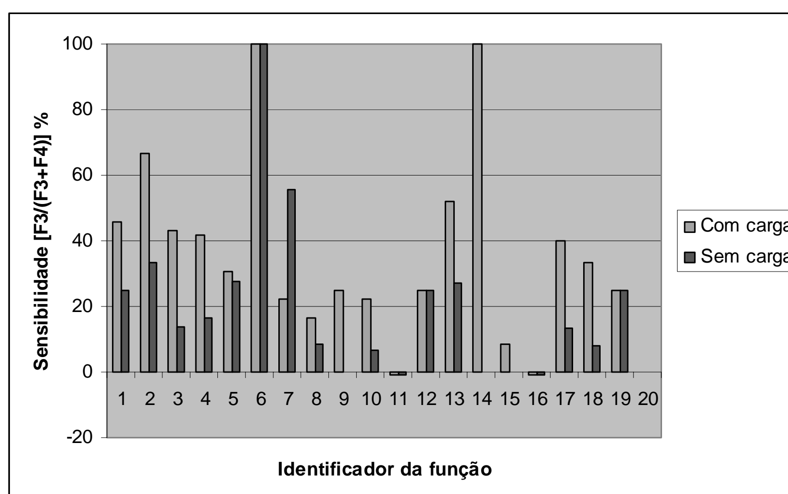

Figura 4 - Sensibilidade relativa do sistema de ficheiros [F3/(F3+F4)*100].

5 Conclusões

O conjunto de experiências realizadas permitem avaliar a capacidade de autoprotecção do Windows XP quando recebe dados de drivers faltosos. A análise destes resultados mostra que a maior parte das funções não consegue efectuar uma verificação segura dos seus parâmetros – das 20 funções analisadas, apenas 2 demonstraram ser

100% imunes aos parâmetros passados. Apesar de em nenhuma das experiências se ter verificado um bloqueio do sistema, foi observado um número razoável de paragens. A análise do log das experiências permitiu constatar que as paragens ocorrem principalmente devido a variáveis não inicializadas ou a apontadores inválidos. Com o aumento da actividade do sistema de ficheiros aumenta também o número de ficheiros corrompidos decorrentes das paragens. Em contraste, as funções nunca devolveram valores ou parâmetros de saída incorrectos o que indica uma boa contenção de erros do Windows. Para além disso, na maior parte dos casos, o SO foi capaz de identificar correctamente a origem das paragens.

Estes resultados permitem sustentar que embora estejam a ser desenvolvidos esforços no sentido de assegurar que os drivers não exibem comportamentos faltosos, a Microsoft deveria diligenciar esforços no sentido de tornar mais robustas as funções do núcleo do SO testando os parâmetros que lhes são passados. Eventualmente, esta atitude poderá reduzir o desempenho do sistema, embora com a tecnologia actual, o impacto para utilizador final poderá ser mínimo.

No entretanto, as organizações que utilizam SO da Microsoft devem adoptar ou reforçar as medidas de controlo de instalação de software autorizado, deixando a tarefa de instalação de drivers a cargo em exclusivo aos administradores de sistemas, e garantir o cumprimento das recomendações quanto à instalação de drivers não certificados e não assinados.

Uma vez que comunidade Open Source não dispõe de um processo centralizado de análise aos DD as organizações que utilizam este tipo de SO devem ficar ainda mais sensíveis para os problemas originados na utilização de drivers pouco experimentados.

Neste momento existem importantes caminhos de evolução na investigação em SO [16] com o objectivo de os tornar SO mais robustos: 1) compartimentar o código dos drivers para controlar a interacção com o sistema operativo; 2) a criação de máquinas paravirtuais que colocam os drivers em uma ou mais máquinas virtuais distintas da principal que contém o núcleo do SO; 3) a aproximação multi-servidor que coloca a execução de drivers e componentes do SO em processos que correm em modo utilizador, permitindo a comunicação entre eles através de mecanismos de comunicação entre processos disponibilizados pelo código do micro-núcleo que é executado em modo protegido; 4) a utilização de linguagens que realizam contractos formais e asseguram o que cada módulo pode efectuar.

Os dois primeiros caminhos de investigação pretendem aumentar a confiança nos SO actuais, enquanto que os dois últimos, propõem a substituição por outros mais confiáveis. Em qualquer dos casos o aumento da robustez e confiança no sistema pode traduzir-se num aumento da sua segurança.

6 Trabalho Futuro

Os resultados obtidos neste trabalho encorajam, para já, a aplicação das ferramentas desenvolvidas noutros SO da família NT, pelo que estão a ser efectuados os preparativos para a realização de experiências com a mesma metodologia para o Windows Server 2003 e assim que disponível, o Windows Vista.

Está a ser preparado um conjunto de experiências para obter e avaliar os resultados das funções mais utilizadas tendo em conta a dinâmica do sistema. Esses resultados permitirão concluir a importância da frequência de utilização das funções para a selecção e preparação dos casos de teste.

Por último, pretende-se desenvolver soluções que aumentem a robustez dos sistemas operativos da família NT face aos DD faltosos.

Agradecimentos

Este trabalho foi parcialmente suportado pela FCT através do projecto POSC/EIA/61643/2004 (AJECT) e do Laboratório de Sistemas Informáticos de Grande-Escala (LaSIGE).

Referências

1. Russinovich, M., Solomon D., "Windows Internals", Microsoft Press (2005).
2. Oney W., "Programming the Microsoft Windows Driver Model", Microsoft Press (2003).
3. Albinet A., Arlat J., and Fabre, J.C. "Characterization of the impact of faulty drivers on the robustness of the Linux kernel." In Proc. of the Int. Conference on Dependable Systems and Networks (June 2004), pp. 867–876.
4. Chou, A., Yang J., Chelf, B., Hallem, S., and Engler, D. "On u-kernel construction", In Proc. of the Symposium on Operating Systems Principles (December. 1995), pp. 237–250.
5. Chou, A., Yang J., Chelf, B., Hallem, S., and Engler, D. "An empirical study of operating system errors. " In Proc. of the Symposium on Operating Systems Principles (October 2001), pp. 73–88.
6. Durães, J., and Madeira, H. "Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation. " In Proc. of the Pacific Rim Int. Dependability Symposium (December 2002), pp. 201–209.
7. Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., and Shivers, O. "The Flux OSKit: a substrate for OS language and resource management. " In Proc. of the Symposium on Operating Systems Principles (October 1997), pp. 38–51.
8. Koopman, P., and Devale, J. "Comparing the robustness of POSIX operating systems. " In Proc. of the Int. Symp. on Fault-Tolerant Computing (June 1999), pp. 30–37.
9. Microsoft Corporation. "Microsoft Portable Executable and Common Object File Format Specification", (February 2005).
10. Microsoft Corporation. "Introducing Static Driver Verifier", (May 2006).
11. Simpson, D. "Windows XP Embedded with Service Pack 1 Reliability." Tech. rep., Microsoft Corporation, (January 2003).
12. Swift, M., Bershad, B., and Levy, H. "Improving the reliability of commodity operating systems." In Proc. of the Symposium on Operating Systems Principles (October 2003), pp. 207–222.
13. Young, M., Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., and Tevanian, A. Mach: "A new kernel foundation for UNIX development" In Proc. of the Summer USENIX Conference (June 1986), pp. 93–113.
14. http://www.microsoft.com/whdc/default.mspx (July 2006)
15. http://www.microsoft.com/whdc/DevTools/tools/DDKTools.mspx (July 2006)
16. Tanenbaum, A., Herder, J., Bos, H., "Can We Make Operating Systems Reliable and Secure? ", IEEE Computer (May 2006).