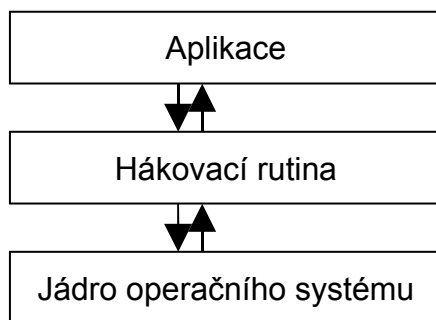


## Některé techniky užívané rootkity

Z úvodní kapitoly dokumentace již víme, co to rootkit je a jaký má cíl. Techniky skrývání jsou rozmanité. Dokonalého výsledku se většinou dosahuje kombinací několika z nich. V této části se na několik takových postupů podíváme. Budeme se zejména zabývat technikami užívanými rootkity-ovladači. Rootkity běžícími v uživatelském režimu se příliš zabývat nebudeme, protože jejich přítomnost lze z detekovat principiálně.

### Hákování

Hákování patří k nejstarším technikám používaným rootkity. Hákování se lehce implementuje a je velice efektivní. Odhalit hákování je však docela jednoduché. Tato nevýhoda je však kompenzována širokou škálou míst, kde lze hákovat.



**Obrázek 1:** Obecný princip hákování

Jak je vidět z obrázku, hákování narušuje komunikaci mezi dvěma komponentami, například mezi aplikací a jádrem operačního systému. Aplikace o něco požádá jádro systému. Jádro systému vykoná požadovaný úkon a vrátí aplikaci výsledky. V kódu zprostředkávajícím komunikaci je však umístěn *hák* – kód, který zajistí přesměrování toku na *hákovací rutinu*. Hákovací rutina převezme řízení hned po odeslání požadavku a těsně před vrácením výsledků, tudíž má nad komunikací plnou kontrolu. Hákovací rutina ani nemusí poslat požadavek jádru operačního systému, může jej zpracovat sama a výsledky požadavku si vymyslet. Pokud se tak stane, hákovací rutina plně nahradí funkci, která byla volána aplikací. Často však kód rutiny volá i originální funkci a mění jí vrácené výsledky. Takový je obecný princip hákování. Podívejme se, kde všude jej můžeme uplatnit.

### Hákování tabulky importů PE souboru (IAT hooking)

Příloze A jste se dozvěděli, že většina PE souborů obsahuje tzv. tabulku importů, v níž jsou uloženy názvy externích funkcí, která kód v PE souboru používá, a modulů, které tyto funkce exportují. Takto vypadá tabulka importů PE souboru na disku. Když je PE soubor spuštěn, operační systém podle tabulky importů zjistí, které další moduly bude nutné načíst do adresového prostoru nově vznikajícího procesu. Když je tato akce vykonána, je tabulka

importů trochu upravena – k jednotlivým názvům funkcí se zaznamenají i jejich přesné adresy. Pokud kód v PE souboru volá některou z funkcí obsažených v tabulce importů, je automaticky předáno řízení na adresu, která byla při startu do tabulky zapsána. Teoreticky by bylo možné si při každém volání externího podprogramu příslušnou adresu zjistit přímo z modulu, který podprogram exportuje, ale to by výrazně zpomalovalo chod procesu. Proto se nic nekontroluje a předá se řízení na adresu uvedenou v na příslušném místě v tabulce importů. Pokud chceme na určitou rutinu umístit hák, změním příslušnou adresu v tabulce importů na adresu naší hákovací rutiny.

Hákování tabulky importů je sice celkem jednoduché realizovat, avšak má jednu velkou nevýhodu. Pokud chceme zahákovat určitou funkci, musíme změnit adresy v tabulkách všech modulů, které ji používají. A ani potom si nemůžeme být stoprocentně jisti, že naše hákovací rutina bude zavolána při každém volání funkce. Proces si samozřejmě správnou adresu může zjistit tak, že se podívá do tabulky exportů modulu, který funkci exportuje. Jako příklad programu, který touto technikou hákování snadno neošálíme, uveďme populární souborový manažer Total Commander.

## „Vkládané“ háky (inline hooks)

Zatímco hákování tabulky importů trpí některými neduhy, „vkládané“ háky téměř žádné nehody nemají. Nemusí být nijak závislé na PE souborech a snadno se používají i v režimu jádra (což o hákování tabulky importů tvrdit nelze).

Princip „vkládaného háku je docela jednoduchý: funkce (nebo oblast), kterou chceme zahákovat, je přepsána tak, aby předala řízení naší hákovací rutině. Toho většinou dosahujeme tak, že přepíšeme prvních pět bytů dané funkce instrukcí nepodmíněného skoku (JMP). Původní kód si samozřejmě schováme, protože jej bude nutné provést, chceme-li během vykonávání hákovací rutiny zavolat i funkci originální. Pokud dojde k volání zahákované funkce, vykoná se instrukce JMP a řízení je předáno naší hákovací rutině. Když budeme chtít zavolat originální funkci, vykonáme přepsané instrukce a předáme řízení na [Adresa\_funkce + 5].

Instrukce procesoru však nemají stejnou délku. Některé zabírají 1 byte, jiné třeba 7 bytů. A tak se může stát, že při přepsání prvních pěti bytů kódu nějaké funkce přepíšeme pouze část nějaké instrukce a tím změním její význam. Pokud by se na začátku funkce nacházel například tento kód,

```
8BFF      mov     edi, edi
8BBC      mov     ebp, esp
FF750C    push  [ebp+0Ch]
FF7508    push  [ebp+08h]
```

nemůžeme jednoduše přepsat 5 počátečních bytů (0x8E, 0xFF, 0x8E, 0xBC, 0xFF), protože bychom změnili význam instrukce PUSH. Dobrým řešením je přepsat celých 7 bytů – 5 bytů zabere instrukce JMP a zbytek vyplní instrukce NOP, které jsou charakteristické tím, že neprovádí žádnou operaci. Po „zahákování“ bude kód funkce vypadat následovně:

```
E9xxxxxx jmp     xxxxxxh
90      nop
90      nop
```

FF7508 push [ebp+08]

Pokud budeme chtít volat z hákovací rutiny originální funkci, provedeme instrukce, které jsme přepsali, a předáme řízení na [Adresa\_funkce + 7]. Pokud budeme hákovat funkce Windows API, výše popsany problém se ani nemusí vyskytnout. V Microsoftu si jsou výhod „vkládaných“ háků také vědomi a používají je<sup>1</sup>. Aby byla instalace háku co nejjednodušší, začátky drtivé většiny funkcí Windows API byly přepsány tak, aby nebylo třeba před umístěním instrukce JMP analyzovat kód. Pokud však jste ještě nestáhli aktualizaci Service Pack 2 pro Windows XP, budete o toto milé zjednodušení ochuzeni.

Výjimečně se může nastat situace, že hákovaná funkce je kratší než 5 bytů. Tento problém se však vyskytuje velmi vzácně a není příliš těžké tuto krátkou funkci přepsat tak, aby bylo hákování úspěšné<sup>2</sup>.

„Vkládaný“ hák může být celkem snadno odhalen, je-li instrukce umístěna hned na prvních pěti bytech hákované rutiny. Pokud si však programátor dá záležet a hák umístí hluboko do těla rutiny, šance na odhalení velmi výrazně klesnou.

## Hákování SSDT

V příloze A jsme si vysvětlili, co to je tabulka interních služeb systému a jak velký má význam pro programy běžící v uživatelském režimu. Není tedy žádným překvapením, že rootkity s touto strukturou pracují. Lehké pozměnění obsahu této tabulky může schovat proces, soubor nebo klíč registru. Zkrátka všechno, na co si vzpomenete.

Pro útočnicka (autora rootkitu) je důležité pole KiServiceTable, které obsahuje adresy rutin dostupných z uživatelského režimu. Všechny adresy těchto rutin ukazují do hlavního modulu jádra – ntoskrnl.exe<sup>3</sup>. Pokud chce rootkit některou rutinu zahákovat, změní příslušnou adresu v poli KiServiceTable. Nová adresa většinou ukazuje do ovladače rootkitu. Když poté dojde k volání této rutiny, systém se podívá do pole KiServiceTable a zavolá rutinu rootkitu. Ta může (ale nemusí) volat i originální rutinu. Rutina rootkitu se stává hákovací rutinou – upraví výstupní informace tak, aby rootkit nebyl prozrazen. Tato technika hákování není nepodobná hákování tabulky importů PE souboru.

Podívejme se nyní blíže na strukturu SSDT a na kroky, které je nutné provést, aby mohlo dojít k hákování. Víme již, že adresa SSDT je exportována hlavním modulem pod symbolem KeServiceDescriptorTable. První krok, který útočnick udělá, spočívá ve zjištění adresy SSDT. Poté se útočnick podívá na data v SSDT ukrytá. Struktura dat je následující:

```
Var KeServiceDescriptorTable : Record
    KiServiceTable : Pointer;
    Neznamy : Cardinal;
    PocetRutin : Cardinal;
    ParamTable : Pointer;
End;
```

<sup>1</sup> Vkládané háky jsou používány při aktualizacích. Tok programu je přesměrován na kód aktualizace. Díky tomu není nutné po každé aktualizaci restartovat počítač.

<sup>2</sup> Příkladem takové funkce Windows API je NtDbgBreakPoint, která má délku pouze 2 bajty.

<sup>3</sup> Hlavním modulem jádra může být i ntkrnlpa.exe nebo ntkrnlp.exe. Záleží na konfiguraci systému a na počtu procesorů.

Položka *KiServiceTable* je adresou pole *KiServiceTable*. Položka *PocetRutin* udává, kolik má pole *KiServiceTable* prvků. Položka *ParamTable* odkazuje na pole, které je stejně velké jako *KiServiceTable*. V tomto je uloženo, kolik má každá rutina, jejíž adresa je v poli *KiServiceTable*, parametrů. Vidíme, že útočník z této struktury získá všechny potřebné informace.

Útočník už tedy zná adresu *KiServiceTable* a ví, pod jakým indexem v tomto poli se skrývá adresa, kterou bude muset změnit. Ani to mu však nestačí. SSDT je totiž chráněna proti zápisu. Pokus o zápis do takto chráněné oblasti vyvolá modrou obrazovku. Útočník musí ochranu proti zápisu na krátký okamžik odstranit.

Existuje několik způsobů, jak ochranu proti zápisu obejít. Nejjednodušší je změnit obsah registru CR0<sup>4</sup>. Jeden bit obsahu tohoto registru procesoru udává, zda mají být některé důležité struktury chráněny proti zápisu. Jednoduchou změnou tohoto bitu útočník ochranu odstraní. Může přitom využít třeba tohoto kódu:

```
push eax
mov eax, CR0
and eax, 0xFFFEFFFF
mov CR0, eax
pop eax
```

Jakmile je ochrana odstraněna, útočník upraví *KiServiceTable* podle svého přání. Potom opět ochranu proti zápisu povolí. Kód bude následující.

```
push eax
mov eax, CR0
or eax, NOT 0xFFFEFFFF
mov CR0, eax
pop eax
```

Často hákovanými funkcemi jsou rutiny sloužící k zjištění seznamu běžících procesů (*NtQuerySystemInformation*), rutiny pracující se soubory (*NtQueryDirectoryFile*) a rutiny pro práci s registry (*NtEnumerateKey*, *NtEnumerateValueKey*).

SSDT háky však nevyužívají pouze záškodnické programy. Například utilita *Region* od firmy *SysInternals*<sup>5</sup>, která monitoruje aktivitu registru, mění SSDT.

Hákování SSDT je opravdu mocná zbraň proti všemu, co běží v uživatelském režimu. Proti ovladačům tuto metodu použít nelze. Navíc lze změnu v tabulce docela snadno zjistit a opravit, čímž je rootkit zbaven svého krytí. Pokud tedy programujeme rootkit (pro dobré účely samozřejmě), měli bychom se této technice vyhnout.

## Hákování IDT

---

<sup>4</sup> Tato metoda odstranění ochrany proti zápisu se nazývá „CR0 trik“ a není příliš dokumentovaná. Dokumentovanou alternativou je využití struktur MDL (*Memory Descriptor List*). Tato technika zde nebude rozebírána.

<sup>5</sup> *Regmon* a další užitečné utility je možno nalézt na adrese <http://www.sysinternals.com>.

V příloze A jsme se dozvěděli, že IDT obsahuje adresy obslužných rutin pro všechna přerušení. Také jsme si řekli, že bez přerušení by mnoho základních mechanismů jako například multitasking nefungovalo. O důležitosti této tabulky není pochyby, což autoři rootkitů a dalšího škodlivého softwaru dobře vědí. Změna adresy rutiny obsluhující určité přerušení je téměř stejně těžká jako změna v tabulce interních služeb operačního systému (SSDT). IDT je chráněna proti zápisu – již jsme si však ukázali, jak tuto ochranu dočasně vypnout.

Víme, že informace o tabulce přerušení získáme instrukcí SIDT. Instrukce na adresu v operandu zapíše následující strukturu:

```
IDTInfo = Record
    Odtlumit : Word;
    LowIDTBase : Word;
    HiIDTBase : Word;
    End;
```

Pro útočníka jsou zajímavé poslední dvě položky – jejich složením získá adresu, kde se nachází vlastní tabulka. Opět se jedná o pole, které má 256 prvků. Každý prvek je tvořen 8 bytovou strukturou. Její deklaraci uvedu v jazyce C, protože v Pascalu to není možné:

```
#pragma pack(1)
typedef struct {
    unsigned short LowIDTOffset;
    unsigned short selector;
    unsigned char  unused_lo;
    unsigned char  segment_type:4;
    unsigned char  systém_segment_flag:1;
    unsigned char  DPL:2;
    unsigned char  P:1;
    unsigned short HiIDTOffset;
}
#pragma pack()
```

V položkách *LowIDTOffset*, *HiIDTOffset* a *selector* je uložena adresa obslužné rutiny, kam je předáno řízení, když je vygenerováno dané přerušení. V položce *segment\_type* se nachází hodnota 0xE, která indikuje, že se jedná o přerušení. *DPL* udává, v jakém Ring módu může být přerušení voláno – používají se pouze hodnoty 0 (pro režim jádra) a 3 (pro uživatelský režim), ostatní Ring módy se ve Windows nepoužívají. Hodnota 3 samozřejmě neznamená, že by toto přerušení nebylo možné volat v režimu jádra. Bit *P* udává, zda je obslužná rutina implementována. Pokud má tento bit hodnotu 0, přerušení je volně k dispozici libovolnému programu.

Z předchozího odstavce vyplývá, že útočníkovi stačí změnit jen položky *HiIDTOffset* a *LowIDTOffset*, obsah položky *selector* je pro všechna přerušení stejný. Zajímavá může být i změna položky *DPL*. Nyní se v krátkosti podívejme, jak lze přerušení využít.

Při každém stisku či uvolnění klávesy je generováno přerušení, jehož číslo může být různé, ale většinou má hodnotu 0x31. Pokud zahákujeme příslušnou položku v IDT, z našeho ovladače se rázem stane keylogger.

Přerušení 0x2E bylo používáno, když proces běžící v uživatelském režimu volal službu jádra. Hákování tohoto přerušení tedy mělo podobný účinek jako hákování SSDT, jen

bylo pracnější a hák nebyl tak viditelný. Windows XP místo tohoto přerušení používají instrukci SYSENTER. Jelikož při volání SYSENTERu dojde de facto k vygenerování přerušení, lze tuto instrukci hákovat. Hákování je ještě snazší než hákování obyčejného přerušení. Tato metoda je využívána například rootkitem Rustock B.

Manipulace s IDT s sebou přináší mnoho nevýhod. Obslužná rutina přerušení je vykonávána s vysokou prioritou, a tudíž nemůže provádět některé operace (práce se soubory, synchronizace). Obslužná rutina by měla být co nejkratší, aby nezatěžovala procesor<sup>6</sup>. Doporučuje se, aby rutina uložila data do nestránkované paměti a skončila, rozhodně by neměla data nějakým způsobem zpracovávat.

Druhá nevýhoda hákování IDT se projeví na víceprocesorových strojích. Každý procesor má totiž svoji vlastní tabulku přerušení. Instrukce SIDT vrací IDT procesoru, na kterém byla provedena. Pokud si tedy útočník chce být jist, že odchytí každé volání určitého přerušení, musí změnit tabulky přerušení všech procesorů.

## **Nahrazení procedury okna (subclassing)**

Subclassing rovněž patří mezi povinné znalosti každého systémového programátora. Užitím této techniky lze dosáhnout některých specifických cílů, které mohou být jinak splněny poněkud obtížněji. Subclassing je navíc plně dokumentován. Aby čtenář pochopil podstatu tohoto postupu, musí znát některé základní informace o základních jednotkách grafického uživatelského rozhraní Windows – o oknech.

Okna tvoří téměř všechny ovládací prvky – textová pole, pracovní plochu textových editorů, tlačítka, rolovací menu, formuláře atd. Okna komunikují s okolím prostřednictvím zpráv. Když například kliknete na tlačítko, příslušné okno tlačítko reprezentující dostane zprávu, že na něj bylo kliknuto a vykoná se příslušná činnost. Okna obsluhují mnoho druhů zpráv, které je informují o různých akcích uživatele – o pohybech myši nebo o stisknutých klávesách.

Každé okno má svoji vlastní rutinu, která je za obsluhu zpráv zodpovědná. Nazývá se *procedura okna*, ačkoliv toto označení není přesné, protože obsluha některých zpráv vyžaduje vrácení hodnoty.

Podstata techniky subclassing spočívá v nahrazení této rutiny za svojí vlastní. Nově definovaná procedura okna má naprostou kontrolu nad zprávami, které jsou oknu posílány a může je blokovat či dokonce měnit. Ve většině případů je však volána originální procedura okna, aby se zajistila obsluha zpráv, které útočníka nezajímají.

## **Odchycení kombinace kláves Ctrl+Alt+Del**

Každý uživatel tuto klávesou zkratku zná – po jejím stisknutí se spustí Správce úloh, nebo je vyvolán dialog umožňující provést některé akce (vypnutí počítače, odhlášení...). Pokud jste však zkusili napsat program, který by tuto zkratku zachytil, pravděpodobně jste zjistili, že to není tak jednoduché – pomocí standardních metod to totiž není možné.

---

<sup>6</sup> Pro snížení priority kódu se používá mechanismus zvaný DPC (Deferred Procedure Call), který umožňuje nejen snížit prioritu na DISPATCH\_LEVEL, ale i určit, na kterém procesoru bude příslušná rutina vykonávána.

O tom, že uživatel stiskl Ctrl+Alt+Del se „dozví“ jen ovladače klávesnice a proces WINLOGON.EXE, kde je informace pohlcena. Jiný proces se o ničem nedozví. Jak tedy tuto klávesovou zkratku odchytit, aniž bychom museli psát ovladač?

Bylo zjištěno, že informace o stisku Ctrl+Alt+Del je poslána do okna s názvem „SAS Window“, které patří procesu WINLOGON.EXE. Pokud chceme tuto informaci také zachytit, musíme tomuto oknu změnit rutinu obsluhující zprávy. Protože se nová obslužná rutina musí nacházet v adresovém prostoru procesu WINLOGON.EXE, je nutné spojit subclassing i se změnou paměti procesu.

Proces WINLOGON.EXE se nestará jen o výše popisovanou klávesovou zkratku, ale i o přihlašování a pravděpodobně i o odhlašování uživatelů a vypínání počítače. Jednoduchým nahrazením procedury okna může útočník například ukrást hesla uživatelů, kteří se na infikovaný počítač přihlásí.

Subclassing se pro svoje konkrétní zaměření na okna příliš nepoužívá, jak však bylo popsáno výše, i s touto technikou může útočník dosáhnout nevídaných výsledků, aniž by musel vynakládat přílišnou námahu na psaní škodlivého kódu, protože implementovat subclassing je velmi jednoduché a nevyžaduje příliš velké znalosti

## ***Vkládání DLL knihoven (DLL injection)***

Ovladače mají veškerou moc nad systémem, mohou jej mást a upravovat. Mohou zajistit naprosté skrytí procesu, souboru nebo klíče registru. Psát ovladač však není snadné. Je k tomu třeba spousty znalostí, trpělivosti a velké opatrnosti. Ne každý autor škodlivého softwaru má tyto vlastnosti. Takoví lidé si však našli cesty, jak si poradit i bez nich.

Bez ovladače není možné skrýt proces příliš dobře. Otázkou je, zda je nutné proces vůbec vytvářet. Škodlivý kód by přece mohl běžet v kontextu nějakého uživateli dobře známého procesu, nejlépe nějakého systémového. Tím bude škodlivý kód skryt všem zvědavým očím. Jediným problémem je donutit cílový proces, aby škodlivý kód vykonával. A tento problém právě řeší technika, jejíž název obsahuje nadpis této kapitoly.

DLL injection je postup, jehož výsledkem je vložení libovolné DLL knihovny do adresového prostoru běžícího procesu. Hned po načtení knihovny do paměti cílového procesu, dojde k volání inicializační rutiny. Kód této rutiny připraví knihovnu na škodlivou činnost. Knihovna může zahákovat důležité rutiny, které se vykonávají například při přihlašování uživatele a ukrást heslo. To samé platí i o internetovém bankovníctví. Hesla mohou být odesílána na email autora, který je pak zneužije. Pokud má uživatel nainstalovaný jen standardní firewall, nemusí si takové aktivity vůbec všimnout – knihovna se může ukrývat v procesu, který má povolený přístup k Internetu. I když uživatel zjistí, že nějaký systémový proces se snaží připojit k Internetu, nemusí mu to být podezřelé.

Vložit DLL knihovnu do běžícího procesu je jednoduché a příliš neomezuje škodlivý kód, který se v ní nachází. Příslušnou knihovnu však bude možné najít v seznamu modulů cílového procesu (pokud se neskryje). Pokud si útočník dá trochu záležet, DLL knihovnu vkládat nemusí – někde do paměti cílového procesu zapíše škodlivý kód, který následně spustí. Seznamu modulů procesu se neobjeví nic podezřelého. Škodlivý kód však musí splňovat dosti přísná kritéria. Musí být psán téměř výhradně v assembleru. Pokud se má použít některá funkce Windows API, musí si kód sám zjistit její adresu. Kód musí být naprosto nezávislý. Pokud si programátor nedá pozor, lze i kód, který splňuje výše zmíněné

podmínky, celkem snadno odhalit. Pokud si uživatel podívá do seznamu vláken procesu a zjistí, že jedno vlákno vykonává kód, který se nenachází v žádném z modulů, může pojmout podezření.

DLL injection je využívána zejména viry, červy a trojskými koňmi. Málomocný „dobrý“ program tuto techniku používá. Mezi tyto vzácné exempláře patří i Process Inspector.

## **Řetězce ovladačů**

Nyní se budeme zabývat technikou, kterou lze využít pouze v režimu jádra. Technika je založená na tom, že mnoho ovladačů je spojeno do jakýchsi řetězců, po nichž se přenáší informace. Pokud například stisknete klávesu, příslušný ovladač tuto událost zaznamená a informuje driver, který tvoří další článek řetězce. Informace se přenáší po řetězci, dokud nedosáhne jeho konce. Až poté se dostane k programům běžícím v uživatelském režimu.

Tyto řetězce nejsou neměnné. Může se do nich zapojit jakýkoliv ovladač. Rootkit se tedy do řetězce může zapojit a monitorovat události v systému, ale nejen to. Každý ovladač v řetězci může přenášené informace měnit. To může být velmi užitečné pokud rootkit chce skrýt svůj soubor. Při práci se souborovým systémem se také přenáší informace po řetězci. A i kdyby to tak nebylo, rootkit musí jenom vědět, který ovladač obsluhuje požadavky na souborový systém, a může se k němu připojit.

Tato metoda je výhodná, protože nedochází k žádnému hákování, jehož slabou stránkou je snadná detekce.

I soubory skryté touto technikou je možné nalézt. Jednou z možností je vytvořit si vlastní interpret souborového systému a všechny operace se soubory provádět přes čtení a zápis diskových sektorů. Požadavky na čtení a zápis sektorů budou samozřejmě zachyceny i rootkitem, ale můžeme doufat, že nebude schopen rozpoznat, co čtení a zápis diskových sektorů znamená.

## **DKOM**

DKOM je zkratka z anglických slov Direct Kernel Object Manipulation. Jedná se o jednu z nejúčinnějších technik, které může rootkit použít. Podstata spočívá v přepisování obsahu důležitých struktur jádra. K manipulaci se strukturami však nejsou používány žádné systémem zabudované rutiny, ale paměť, kde se struktury nachází, je přímo přepisována. Ačkoliv je velmi efektivní, má tato metoda některé značné nevýhody:

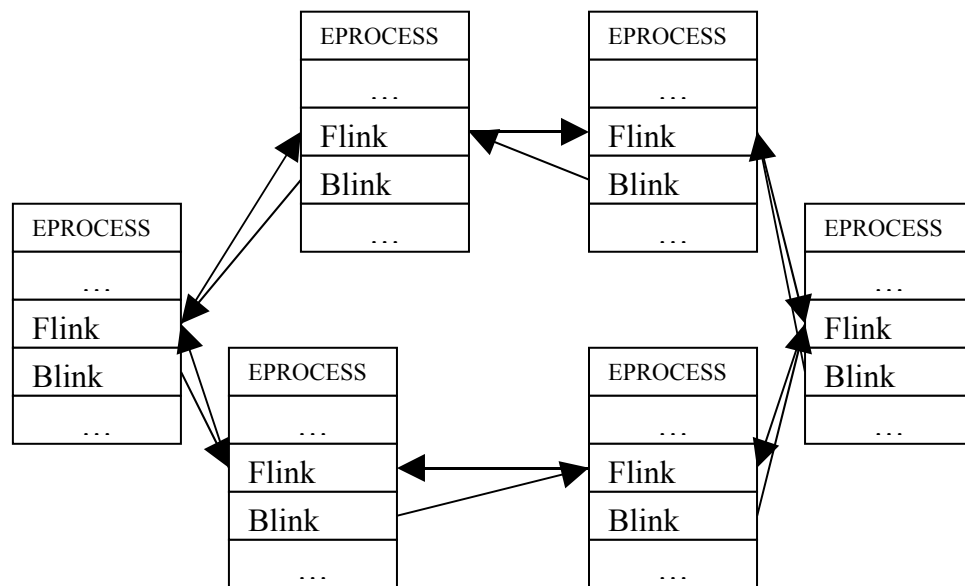
- Obsah důležitých struktur jádra se může měnit v závislosti na verzi operačního systému. Změnu může přinést třeba nainstalování Service Packu. Pokud chce autor rootkitu využít metodu DKOM, musí si na verzi systému dávat velký pozor.
- Využití je omezené. Pomocí DKOM lze skrýt proces nebo ovladač, ale ne soubor.
- Informací o interních strukturách jádra je velmi málo. Programátor si musí vše zjistit sám.

Z nevýhod vyplývá, že DKOM je určena především pro ty, kteří mají s programováním ovladačů velké zkušenosti. Je zřejmé, že drtivá většina čtenářů tohoto textu do této skupiny nepatří, protože jinak by tento text ani nečetla. Ukažme si tedy, jak jednoduše DKOM využít. Doufám, že z níže uvedených příkladů pochopíte, jak mocnou zbraní může tato technika být.

## Skrývání procesů a ovladačů

První veřejný rootkit používající DKOM má název FU. FU krom jiného skrývat procesy a ovladače. Způsob je velice elegantní, jednoduchý a efektivní, použijeme-li jej společně s dalšími technikami skrývání. Dříve, než vám prozradím, v čem spočívá, vysvětlím, jak pracují funkce Windows API, které získávají seznam běžících procesů.

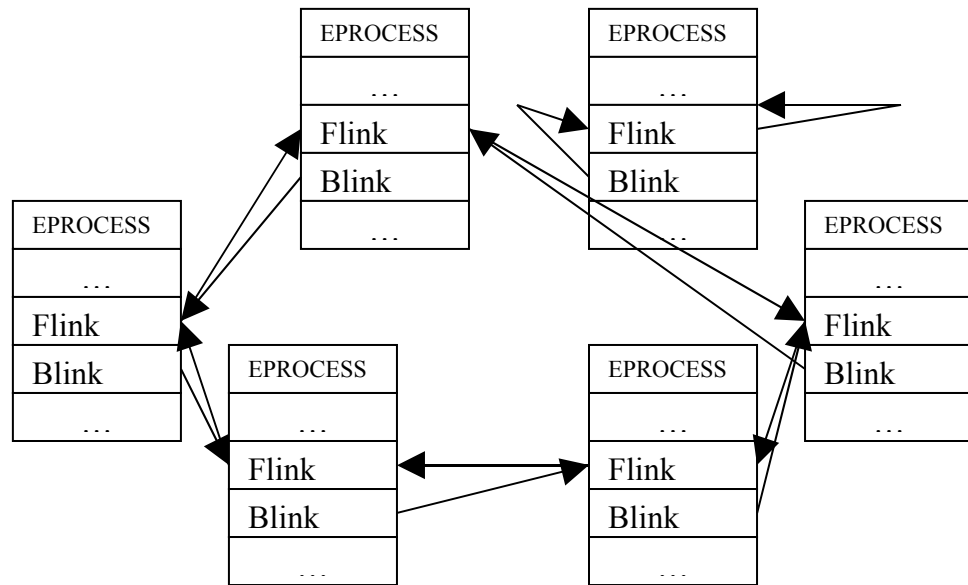
Všechny funkce Windows API, které zjišťují, jaké procesy běží právě v systému, interně volají rutinu NtQuerySystemInformation z knihovny ntdll.dll. V ntdll.dll dojde k překladu jména volané funkce (tedy NtQuerySystemInformation) na index do pole KiServiceTable a je zavolán příslušný podprogram. Tento podprogram zjišťuje informace o procesech ze struktur, které se nacházejí někde v paměti jádra. Tyto struktury se nazývají EPROCESS a každá z nich obsahuje informace o jednom z běžících procesů.



**Obrázek 2:** Propojení struktur EPROCESS na čistém systému

Z obrázku je patrné, že podprogram potřebuje nalézt jen jednu strukturu EPROCESS. Všechny struktury jsou totiž propojeny odkazy a tvoří obousměrný spojový seznam<sup>7</sup>. Položky Flink a Blink se nachází kdesi v hloubi struktury EPROCESS. Jejich poloha je závislá na verzi operačního systému, avšak je veřejně známá. Pokud víte, co to obousměrné spojové seznamy jsou, víte také, jak skrýt proces. Stačí jeho strukturu EPROCESS odejmout ze spojového seznamu. Stav po tomto kroku je vidět na obrázku.

<sup>7</sup> Jádro využívá spojových seznamů mnohem častěji než normální aplikace, protože se jednotlivé prvky seznamu mohou nacházet kdekoli v paměti. Nevýhodou pole je velký počet operací, který je nutné provést při přidání nebo odebrání prvku.



**Obrázek 3:** V systému běží proces skrytý technikou DKOM

Rootkit FU odejme proces z tohoto spojového seznamu a tím jej skryje. Krytí však není dokonalé a lze jej prolomit za využití prostředků Windows API. Autor rootkitu si toho byl vědom, a tak vytvořil pokračování s názvem FuTo. FuTo aplikuje DKOM na několik dalších struktur jádra, takže lze skryté procesy odhalit pouze za použití ovladače.

Seznam běžících ovladačů je rovněž uložen v obousměrném spojovém seznamu, kde každý prvek reprezentuje jeden ovladač. Ovladače lze tedy skrývat stejným způsobem jako procesy.